PAMS COMPLETE EXPLANATION DOCUMENTATION

Patient Appointment Management System (PAMS) - Complete Documentation

Table of Contents

Project Overview

The Patient Appointment Management System (PAMS) is a comprehensive web application built with Spring Boot that allows patients to book appointments with doctors, manage their appointments, and provides administrative functionality for system management.

Key Features

- Patient Registration & Authentication: Secure patient account creation and login
- Doctor Management: Admin can add doctors with availability schedules
- Appointment Booking: Real-time slot availability checking and booking
- Appointment Management: View, cancel, and track appointment status
- Admin Dashboard: System-wide appointment and doctor management
- Responsive Design: Modern, user-friendly interface

Technology Stack

Backend Technologies

- Java 17: Programming language
- Spring Boot 3.2.5: Application framework
- Spring MVC: Web framework
- Spring Data JPA: Data access layer
- Hibernate: ORM framework
- MySQL: Database management system

Frontend Technologies

- Thymeleaf: Server-side template engine
- HTML5: Markup language
- CSS3: Styling
- JavaScript: Client-side scripting

Build & Development Tools

- Maven: Build automation and dependency management
- Spring Boot DevTools: Development utilities
- Jackson: JSON processing

PROJECT STRUCTURE

```
pams/
├── pom.xml                    # Maven configuration
├── src/main/java/com/example/pams/
│   ├── PamsApplication.java          # Main Spring Boot application
│   ├── controller/                # Web controllers (MVC layer)
│   │   ├── AdminController.java
│   │   ├── AppointmentController.java
│   │   ├── DoctorController.java
│   │   ├── HomeController.java
│   │   └── PatientController.java
│   ├── entity/                  # JPA entities (Data model)
│ │  ├── Admin.java
│ │  ├── Appointment.java
│ │  ├── Doctor.java
│ │  └── Patient.java
│  ├── repository/                # Data access layer
│ │  ├── AdminRepository.java
│ │  ├── AppointmentRepository.java
│ │  ├── DoctorRepository.java
│ │  └── PatientRepository.java
│  └── service/                   # Business logic layer
│     ├── AdminService.java
│     ├── AppointmentService.java
│     ├── DoctorService.java
│     └── PatientService.java
└── src/main/resources/
   ├── application.properties       # Application configuration
   ├── schema.sql                 # Database initialization
   ├── static/css/                # CSS stylesheets
   └── templates/                 # Thymeleaf HTML templates
      ├── fragments/navbar.html
      ├── index.html
      ├── patient-*.html
      ├── admin-*.html
      └── appointment-*.html
```

Complete Code Analysis

1. Main Application Class

PamsApplication.java

```
package com.example.pams;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

/**

 * Main Spring Boot Application Class

 *

 * This is the entry point of the Patient Appointment Management System (PAMS).

 * The @SpringBootApplication annotation combines three important annotations:

 * - @Configuration: Marks this class as a configuration class

 * - @EnableAutoConfiguration: Enables Spring Boot's auto-configuration

 * - @ComponentScan: Enables component scanning in the package

 */

@SpringBootApplication

public class PamsApplication {


  /**

   * Main method - Entry point of the application

   *

   * @param args Command line arguments passed to the application

   *

   * This method starts the Spring Boot application by:

   * 1. Creating an ApplicationContext

   * 2. Starting the embedded Tomcat server

   * 3. Auto-configuring all Spring components

   * 4. Running the application on the configured port (8082)

   */

  public static void main(String[] args) {
    // SpringApplication.run() starts the Spring Boot application
    // It returns an ApplicationContext that manages all Spring beans
    SpringApplication.run(PamsApplication.class, args);
  }
}
```

2. Entity Classes (Data Model)

Patient.java

```
package com.example.pams.entity;

import jakarta.persistence.*;
import java.time.LocalDate;

/**
 * Patient Entity Class
 *
 * This class represents a Patient in the database.
 * It maps to the 'patient' table in MySQL database.
 *
 * Key Features:
 * - JPA annotations for database mapping
 * - Auto-generated primary key
 * - Unique email constraint
 * - Date of birth using LocalDate
 * - TEXT field for address to handle longer addresses
 */
@Entity  // Marks this class as a JPA entity
public class Patient {
```

```java
/**
 * Primary Key - Patient ID
 *
 * @GeneratedValue(strategy = GenerationType.IDENTITY):
 * - Uses database auto-increment for ID generation
 * - Each new patient gets a unique, sequential ID
 */
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer patientId;

/**
 * Patient's full name
 * Maps to 'name' column in database
 */
private String name;

/**
 * Patient's email address
 *
 * @Column(unique = true):
 * - Ensures no two patients can have the same email
 * - Database constraint prevents duplicate emails
 * - Used for login authentication
 */
@Column(unique = true)
private String email;
```

```
/**
 * Patient's phone number
 * Maps to 'phone' column in database
 */
private String phone;

/**
 * Patient's address
 *
 * @Column(columnDefinition = "TEXT"):
 * - Uses TEXT data type instead of VARCHAR
 * - Allows for longer address strings
 * - No length limitation like VARCHAR(255)
 */
@Column(columnDefinition = "TEXT")
private String address;

/**
 * Patient's date of birth
 *
 * LocalDate:
 * - Java 8+ date class (date only, no time)
 * - Maps to DATE column in database
 * - Used for age calculation and validation
 */
private LocalDate dob;

/**
```

```
 * Patient's password for login
 *
 * Note: In production, this should be encrypted using BCrypt
 * Currently stored as plain text (security improvement needed)
 */
private String password;

// ========== GETTER AND SETTER METHODS ==========

/**
 * Getter for Patient ID
 * @return Integer patientId
 */
public Integer getPatientId() {
   return patientId;
}

/**
 * Setter for Patient ID
 * @param id the patient ID to set
 */
public void setPatientId(Integer id) {
   this.patientId = id;
}

/**
 * Getter for Patient Name
 * @return String name
 */
public String getName() {
   return name;
}

/**
 * Setter for Patient Name
 * @param name the patient name to set
 */
public void setName(String name) {
   this.name = name;
}

/**
 * Getter for Patient Email
 * @return String email
 */
public String getEmail() {
   return email;
}

/**
 * Setter for Patient Email
 * @param email the patient email to set
 */
public void setEmail(String email) {
   this.email = email;
}
```

```
/**
 * Getter for Patient Phone
 * @return String phone
 */
public String getPhone() {
   return phone;
}

/**
 * Setter for Patient Phone
 * @param phone the patient phone to set
 */
public void setPhone(String phone) {
   this.phone = phone;
}

/**
 * Getter for Patient Address
 * @return String address
 */
public String getAddress() {
   return address;
}

/**
 * Setter for Patient Address
 * @param address the patient address to set
 */
public void setAddress(String address) {
    this.address = address;
}

/**
 * Getter for Patient Date of Birth
 * @return LocalDate dob
 */
public LocalDate getDob() {
    return dob;
}

/**
 * Setter for Patient Date of Birth
 * @param dob the patient date of birth to set
 */
public void setDob(LocalDate dob) {
    this.dob = dob;
}

/**
 * Getter for Patient Password
 * @return String password
 */
public String getPassword() {
    return password;
}
```

```
    /**
     * Setter for Patient Password
     * @param password the patient password to set
     */
    public void setPassword(String password) {
        this.password = password;
    }
}
```

Doctor.java

```
package com.example.pams.entity;

import jakarta.persistence.*;

/**
 * Doctor Entity Class
 *
 * This class represents a Doctor in the database.
 * It maps to the 'doctor' table in MySQL database.
 *
 * Key Features:
 * - Doctor specialization field
 * - JSON storage for availability schedule
 * - Contact information (email, phone)
 */
@Entity  // Marks this class as a JPA entity
public class Doctor {
```

```
/**
 * Primary Key - Doctor ID
 * Auto-generated using database identity
 */
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer doctorId;

/**
 * Doctor's full name
 * Maps to 'name' column in database
 */
private String name;

/**
 * Doctor's medical specialization
 * Examples: "Cardiology", "Neurology", "General Medicine"
 * Maps to 'specialization' column in database
 */
private String specialization;

/**
 * Doctor's email address
 * Used for contact and communication
 * Maps to 'email' column in database
 */
private String email;
```

```
/**
 * Doctor's phone number
 * Used for contact and emergency communication
 * Maps to 'phone' column in database
 */
private String phone;

/**
 * Doctor's availability schedule
 *
 * @Column(columnDefinition = "json"):
 * - Stores availability as JSON format
 * - Contains start and end times for working hours
 * - Example: {"start": "09:00", "end": "17:00"}
 * - Used to generate available time slots for appointments
 */
@Column(columnDefinition = "json")
private String availability;

// ========== GETTER AND SETTER METHODS ==========

/**
 * Getter for Doctor ID
 * @return Integer doctorId
 */
public Integer getDoctorId() {
  return doctorId;
}
```

```
/**
 * Setter for Doctor ID
 * @param id the doctor ID to set
 */
public void setDoctorId(Integer id) {
   this.doctorId = id;
}

/**
 * Getter for Doctor Name
 * @return String name
 */
public String getName() {
   return name;
}

/**
 * Setter for Doctor Name
 * @param name the doctor name to set
 */
public void setName(String name) {
   this.name = name;
}

/**
 * Getter for Doctor Specialization
 * @return String specialization
 */
public String getSpecialization() {
   return specialization;
}

/**
 * Setter for Doctor Specialization
 * @param s the doctor specialization to set
 */
public void setSpecialization(String s) {
   this.specialization = s;
}

/**
 * Getter for Doctor Email
 * @return String email
 */
public String getEmail() {
   return email;
}

/**
 * Setter for Doctor Email
 * @param email the doctor email to set
 */
public void setEmail(String email) {
   this.email = email;
}
```

```
/**
 * Getter for Doctor Phone
 * @return String phone
 */
public String getPhone() {
   return phone;
}

/**
 * Setter for Doctor Phone
 * @param phone the doctor phone to set
 */
public void setPhone(String phone) {
   this.phone = phone;
}

/**
 * Getter for Doctor Availability
 * @return String availability (JSON format)
 */
public String getAvailability() {
   return availability;
}

/**
 * Setter for Doctor Availability
 * @param availability the doctor availability to set (JSON format)
```

```
     */
    public void setAvailability(String availability) {
        this.availability = availability;
    }
}
```

Appointment.java

```
package com.example.pams.entity;


import jakarta.persistence.*;
import java.time.LocalDate;
import java.time.LocalTime;


/**
 * Appointment Entity Class
 *
 * This class represents an Appointment in the database.
 * It maps to the 'appointment' table in MySQL database.
 *
 * Key Features:
 * - Many-to-One relationship with Patient and Doctor
 * - Separate date and time fields for precise scheduling
 * - Status enum for appointment lifecycle management
 * - Default status is BOOKED when appointment is created
 */
@Entity  // Marks this class as a JPA entity
public class Appointment {


    /**
 * Primary Key - Appointment ID
 * Auto-generated using database identity
 */
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer appointmentId;

/**
 * Patient associated with this appointment
 *
 * @ManyToOne:
 * - Many appointments can belong to one patient
 * - Creates foreign key relationship to patient table
 * - JPA will automatically handle the relationship
 */
@ManyToOne
private Patient patient;

/**
 * Doctor associated with this appointment
 *
 * @ManyToOne:
 * - Many appointments can belong to one doctor
 * - Creates foreign key relationship to doctor table
 * - JPA will automatically handle the relationship
 */
@ManyToOne
private Doctor doctor;
```

```
/**
 * Date of the appointment
 *
 * LocalDate:
 * - Java 8+ date class (date only, no time)
 * - Maps to DATE column in database
 * - Used for scheduling and filtering appointments
 */
private LocalDate appointmentDate;

/**
 * Time slot of the appointment
 *
 * LocalTime:
 * - Java 8+ time class (time only, no date)
 * - Maps to TIME column in database
 * - Used for precise time scheduling
 * - Combined with date for unique appointment slots
 */
private LocalTime timeSlot;

/**
 * Status of the appointment
 *
 * @Enumerated(EnumType.STRING):
 * - Stores enum values as strings in database
 * - More readable than storing as numbers
```

```java
 * - Default value is BOOKED when appointment is created
 */
@Enumerated(EnumType.STRING)
private Status status = Status.BOOKED;

/**
 * Appointment Status Enumeration
 *
 * Defines the possible states of an appointment:
 * - BOOKED: Appointment is scheduled and confirmed
 * - CANCELED: Appointment has been canceled
 * - COMPLETED: Appointment has been completed
 */
public static enum Status {
   BOOKED,     // Initial status when appointment is created
   CANCELED,   // Status when appointment is canceled
   COMPLETED   // Status when appointment is finished
}

// ========== GETTER AND SETTER METHODS ==========

/**
 * Getter for Appointment ID
 * @return Integer appointmentId
 */
public Integer getAppointmentId() {
   return appointmentId;
}
```

```
/**
 * Setter for Appointment ID
 * @param id the appointment ID to set
 */
public void setAppointmentId(Integer id) {
   this.appointmentId = id;
}

/**
 * Getter for Patient
 * @return Patient patient
 */
public Patient getPatient() {
   return patient;
}

/**
 * Setter for Patient
 * @param p the patient to set
 */
public void setPatient(Patient p) {
   this.patient = p;
}

/**
 * Getter for Doctor
 * @return Doctor doctor
 */
public Doctor getDoctor() {
   return doctor;
}

/**
 * Setter for Doctor
 * @param d the doctor to set
 */
public void setDoctor(Doctor d) {
   this.doctor = d;
}

/**
 * Getter for Appointment Date
 * @return LocalDate appointmentDate
 */
public LocalDate getAppointmentDate() {
   return appointmentDate;
}

/**
 * Setter for Appointment Date
 * @param d the appointment date to set
 */
public void setAppointmentDate(LocalDate d) {
   this.appointmentDate = d;
}
```

```
/**
 * Getter for Time Slot
 * @return LocalTime timeSlot
 */
public LocalTime getTimeSlot() {
   return timeSlot;
}

/**
 * Setter for Time Slot
 * @param t the time slot to set
 */
public void setTimeSlot(LocalTime t) {
   this.timeSlot = t;
}

/**
 * Getter for Appointment Status
 * @return Status status
 */
public Status getStatus() {
   return status;
}

/**
 * Setter for Appointment Status
 * @param s the status to set
```

```
  */
  public void setStatus(Status s) {
    this.status = s;
  }
}
```

Admin.java

```
package com.example.pams.entity;

import jakarta.persistence.*;

/**
 * Admin Entity Class
 *
 * This class represents an Admin user in the database.
 * It maps to the 'Admin' table in MySQL database.
 *
 * Key Features:
 * - Role-based access control
 * - Unique email constraint
 * - Admin authentication for system management
 */
@Entity  // Marks this class as a JPA entity
@Table(name = "Admin")  // Explicitly specifies table name as "Admin"
public class Admin {

  /**
   * Primary Key - Admin ID
   * Auto-generated using database identity
 */
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Integer adminId;

/**
 * Admin's full name
 * Maps to 'name' column in database
 */
private String name;

/**
 * Admin's email address
 *
 * @Column(unique = true):
 * - Ensures no two admins can have the same email
 * - Database constraint prevents duplicate emails
 * - Used for login authentication
 */
@Column(unique = true)
private String email;

/**
 * Admin's role in the system
 *
 * Examples: "SUPERADMIN", "ADMIN", "MANAGER"
 * Used for role-based access control
 * Maps to 'role' column in database
 */
private String role;

/**
 * Admin's password for login
 *
 * Note: In production, this should be encrypted using BCrypt
 * Currently stored as plain text (security improvement needed)
 */
private String password;

// ========== GETTER AND SETTER METHODS ==========

/**
 * Getter for Admin ID
 * @return Integer adminId
 */
public Integer getAdminId() {
   return adminId;
}

/**
 * Setter for Admin ID
 * @param id the admin ID to set
 */
public void setAdminId(Integer id) {
   this.adminId = id;
}
```

```
/**
 * Getter for Admin Name
 * @return String name
 */
public String getName() {
   return name;
}

/**
 * Setter for Admin Name
 * @param name the admin name to set
 */
public void setName(String name) {
   this.name = name;
}

/**
 * Getter for Admin Email
 * @return String email
 */
public String getEmail() {
   return email;
}

/**
 * Setter for Admin Email
 * @param email the admin email to set
 */
public void setEmail(String email) {
   this.email = email;
}

/**
 * Getter for Admin Role
 * @return String role
 */
public String getRole() {
   return role;
}

/**
 * Setter for Admin Role
 * @param role the admin role to set
 */
public void setRole(String role) {
   this.role = role;
}

/**
 * Getter for Admin Password
 * @return String password
 */
public String getPassword() {
   return password;
}
```

```
    /**
     * Setter for Admin Password
     * @param password the admin password to set
     */
    public void setPassword(String password) {
        this.password = password;
    }
}
```

3. Repository Interfaces (Data Access Layer)

PatientRepository.java

```
package com.example.pams.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.pams.entity.Patient;

/**
 * Patient Repository Interface
 *
 * This interface extends JpaRepository to provide CRUD operations for Patient
entity.
 * Spring Data JPA automatically implements this interface at runtime.
 *
 * Key Features:
 * - Inherits basic CRUD operations from JpaRepository
 * - Custom query methods for specific business needs
 * - Method naming conventions for automatic query generation
 */
```

```
public interface PatientRepository extends JpaRepository<Patient, Integer> {

    /**
     * Find patient by email and password for authentication
     *
     * @param email Patient's email address
     * @param password Patient's password
     * @return Patient object if found, null otherwise
     *
     * This method is used for patient login authentication.
     * Spring Data JPA automatically generates the query:
     * SELECT * FROM patient WHERE email = ? AND password = ?
     */
    Patient findByEmailAndPassword(String email, String password);

    /**
     * Find patient by email address
     *
     * @param email Patient's email address
     * @return Patient object if found, null otherwise
     *
     * This method is used to:
     * - Check if email already exists during registration
     * - Find patient by email for various operations
     *
     * Spring Data JPA automatically generates the query:
     * SELECT * FROM patient WHERE email = ?
     */
    Patient findByEmail(String email);


    // Note: JpaRepository provides these methods automatically:
    // - save(Patient patient) - Save or update patient
    // - findById(Integer id) - Find patient by ID
    // - findAll() - Get all patients
    // - deleteById(Integer id) - Delete patient by ID
    // - count() - Count total patients
    // - existsById(Integer id) - Check if patient exists
}
```

DoctorRepository.java

```
package com.example.pams.repository;


import org.springframework.data.jpa.repository.JpaRepository;
import com.example.pams.entity.Doctor;


/**
 * Doctor Repository Interface
 *
 * This interface extends JpaRepository to provide CRUD operations for Doctor
entity.
 * Spring Data JPA automatically implements this interface at runtime.
 *
 * Key Features:
 * - Inherits basic CRUD operations from JpaRepository
 * - Currently no custom methods (uses only inherited methods)
 * - Can be extended with custom query methods as needed
 */
```

```java
public interface DoctorRepository extends JpaRepository<Doctor, Integer> {

   // Note: JpaRepository provides these methods automatically:
   // - save(Doctor doctor) - Save or update doctor
   // - findById(Integer id) - Find doctor by ID
   // - findAll() - Get all doctors
   // - deleteById(Integer id) - Delete doctor by ID
   // - count() - Count total doctors
   // - existsById(Integer id) - Check if doctor exists

   // Future custom methods could include:
   // - findBySpecialization(String specialization) - Find doctors by specialty
   // - findByNameContaining(String name) - Search doctors by name
   // - findByEmail(String email) - Find doctor by email
}
```

AppointmentRepository.java

```java
package com.example.pams.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.pams.entity.Appointment;
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;

/**
 * Appointment Repository Interface
 *
 * This interface extends JpaRepository to provide CRUD operations for
Appointment entity.
 * Spring Data JPA automatically implements this interface at runtime.
 *
 * Key Features:
 * - Inherits basic CRUD operations from JpaRepository
 * - Custom query methods for appointment-specific operations
 * - Complex queries using method naming conventions
 */
public interface AppointmentRepository extends JpaRepository<Appointment,
Integer> {

  /**
   * Find all appointments for a specific patient
   *
   * @param patientId The ID of the patient
   * @return List of appointments for the patient
   *
   * This method is used to:
   * - Display patient's appointment history
   * - Show patient's upcoming appointments
   * - Patient dashboard functionality
   *
   * Spring Data JPA automatically generates the query:
   * SELECT * FROM appointment WHERE patient_id = ?
   */
  List<Appointment> findByPatientPatientId(Integer patientId);

  /**
```

```
     * Find appointments by doctor, date, and time slot
     *
     * @param doctorId The ID of the doctor
     * @param d The appointment date
     * @param t The time slot
     * @return List of appointments matching the criteria
     *
     * This method is used to:
     * - Check if a time slot is already booked
     * - Prevent double booking of the same slot
     * - Validate appointment availability
     *
     * Spring Data JPA automatically generates the query:
     * SELECT * FROM appointment WHERE doctor_id = ? AND appointment_date = ?
AND time_slot = ?
     */
    List<Appointment>
findByDoctorDoctorIdAndAppointmentDateAndTimeSlot(Integer doctorId,
LocalDate d, LocalTime t);


    /**
     * Find appointments by status
     *
     * @param status The appointment status (BOOKED, CANCELED, COMPLETED)
     * @return List of appointments with the specified status
     *
     * This method is used to:
     * - Filter appointments by status
     * - Generate reports based on appointment status
     * - Admin dashboard statistics
     *
     * Spring Data JPA automatically generates the query:
     * SELECT * FROM appointment WHERE status = ?
     */
    List<Appointment> findByStatus(String status);

    // Note: JpaRepository provides these methods automatically:
    // - save(Appointment appointment) - Save or update appointment
    // - findById(Integer id) - Find appointment by ID
    // - findAll() - Get all appointments
    // - deleteById(Integer id) - Delete appointment by ID
    // - count() - Count total appointments
    // - existsById(Integer id) - Check if appointment exists
}
```

AdminRepository.java

```
package com.example.pams.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import com.example.pams.entity.Admin;

/**
 * Admin Repository Interface
 *
 * This interface extends JpaRepository to provide CRUD operations for Admin
entity.
 * Spring Data JPA automatically implements this interface at runtime.
 *
 * Key Features:
 * - Inherits basic CRUD operations from JpaRepository
 * - Custom authentication method for admin login
 * - Simple interface with minimal custom methods
 */
public interface AdminRepository extends JpaRepository<Admin, Integer> {

    /**
     * Find admin by email and password for authentication
     *
     * @param email Admin's email address
     * @param password Admin's password
     * @return Admin object if found, null otherwise
     *
     * This method is used for admin login authentication.
     * Spring Data JPA automatically generates the query:
     * SELECT * FROM admin WHERE email = ? AND password = ?
     */
    Admin findByEmailAndPassword(String email, String password);

    // Note: JpaRepository provides these methods automatically:
    // - save(Admin admin) - Save or update admin
    // - findById(Integer id) - Find admin by ID
    // - findAll() - Get all admins
    // - deleteById(Integer id) - Delete admin by ID
    // - count() - Count total admins
    // - existsById(Integer id) - Check if admin exists
    // Future custom methods could include:
    // - findByRole(String role) - Find admins by role
    // - findByEmail(String email) - Find admin by email
}
```

4. Service Classes (Business Logic Layer)

PatientService.java

```
package com.example.pams.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.pams.repository.PatientRepository;
import com.example.pams.entity.Patient;
import java.util.*;

/**
 * Patient Service Class
 *
 * This class contains the business logic for Patient operations.
 * It acts as an intermediary between the Controller and Repository layers.
 *
 * Key Features:
 * - @Service annotation marks it as a Spring service component
 * - @Autowired dependency injection for repository
 * - Business logic encapsulation
 * - Transaction management (automatic with Spring)
 */
@Service  // Marks this class as a Spring service component
```

```
public class PatientService {


  /**
   * Patient Repository - Injected by Spring
   *
   * @Autowired:
   * - Spring automatically injects the PatientRepository implementation
   * - No need to manually create repository instances
   * - Enables loose coupling between service and repository
   */
  @Autowired
  private PatientRepository patientRepo;


  /**
   * Authenticate patient using email and password
   *
   * @param email Patient's email address
   * @param password Patient's password
   * @return Patient object if authentication successful, null otherwise
   *
   * Business Logic:
   * - Validates patient credentials
   * - Used for login functionality
   * - Returns patient object for session management
   */
  public Patient findByEmailAndPassword(String email, String password) {
    return patientRepo.findByEmailAndPassword(email, password);
  }
```

```
/**
 * Find patient by email address
 *
 * @param email Patient's email address
 * @return Patient object if found, null otherwise
 *
 * Business Logic:
 * - Used to check if email already exists during registration
 * - Prevents duplicate patient registrations
 * - Email uniqueness validation
 */
public Patient findByEmail(String email) {
    return patientRepo.findByEmail(email);
}

/**
 * Get all patients
 *
 * @return List of all patients in the system
 *
 * Business Logic:
 * - Used for admin functionality
 * - Patient management operations
 * - System reporting
 */
public List<Patient> findAll() {
    return patientRepo.findAll();
}

/**
 * Find patient by ID
 *
 * @param id Patient ID
 * @return Optional<Patient> - may contain patient or be empty
 *
 * Business Logic:
 * - Used to retrieve specific patient information
 * - Optional return type prevents NullPointerException
 * - Safe patient lookup
 */
public Optional<Patient> findById(Integer id) {
   return patientRepo.findById(id);
}

/**
 * Save or update patient
 *
 * @param patient Patient object to save
 * @return Saved patient object with generated ID
 *
 * Business Logic:
 * - Handles both new patient creation and updates
 * - Automatic ID generation for new patients
 * - Transaction management (automatic rollback on failure)
 */
```

```
  public Patient save(Patient patient) {
    return patientRepo.save(patient);
  }

  /**
   * Delete patient by ID
   *
   * @param id Patient ID to delete
   *
   * Business Logic:
   * - Removes patient from the system
   * - Should check for existing appointments before deletion
   * - Transaction management (automatic rollback on failure)
   *
   * Note: This method should be enhanced to handle:
   * - Cascade deletion of related appointments
   * - Soft delete instead of hard delete
   * - Audit trail for deletions
   */
  public void deleteById(Integer id) {
    patientRepo.deleteById(id);
  }
}
```

DoctorService.java

```
package com.example.pams.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.pams.repository.DoctorRepository;
import com.example.pams.entity.Doctor;
import java.util.*;

/**
 * Doctor Service Class
 *
 * This class contains the business logic for Doctor operations.
 * It acts as an intermediary between the Controller and Repository layers.
 *
 * Key Features:
 * - @Service annotation marks it as a Spring service component
 * - @Autowired dependency injection for repository
 * - Business logic encapsulation
 * - Transaction management (automatic with Spring)
 */
@Service  // Marks this class as a Spring service component
public class DoctorService {

    /**
     * Doctor Repository - Injected by Spring
     *
     * @Autowired:
     * - Spring automatically injects the DoctorRepository implementation
     * - No need to manually create repository instances
     * - Enables loose coupling between service and repository
     */
    @Autowired
```

```
private DoctorRepository doctorRepo;

/**
 * Get all doctors
 *
 * @return List of all doctors in the system
 *
 * Business Logic:
 * - Used for displaying doctor list to patients
 * - Doctor selection for appointments
 * - Admin doctor management
 */
public List<Doctor> findAll() {
   return doctorRepo.findAll();
}

/**
 * Find doctor by ID
 *
 * @param id Doctor ID
 * @return Optional<Doctor> - may contain doctor or be empty
 *
 * Business Logic:
 * - Used to retrieve specific doctor information
 * - Optional return type prevents NullPointerException
 * - Safe doctor lookup for appointments
 */
public Optional<Doctor> findById(Integer id) {
    return doctorRepo.findById(id);
}

/**
 * Save or update doctor
 *
 * @param doctor Doctor object to save
 * @return Saved doctor object with generated ID
 *
 * Business Logic:
 * - Handles both new doctor creation and updates
 * - Automatic ID generation for new doctors
 * - Transaction management (automatic rollback on failure)
 * - Used by admin to add new doctors
 */
public Doctor save(Doctor doctor) {
    return doctorRepo.save(doctor);
}

/**
 * Delete doctor by ID
 *
 * @param id Doctor ID to delete
 *
 * Business Logic:
 * - Removes doctor from the system
 * - Should check for existing appointments before deletion
 * - Transaction management (automatic rollback on failure)
```

```
     *
     * Note: This method should be enhanced to handle:
     * - Cascade deletion of related appointments
     * - Soft delete instead of hard delete
     * - Audit trail for deletions
     * - Notify patients of doctor removal
     */
    public void deleteById(Integer id) {
        doctorRepo.deleteById(id);
    }
}
```

AppointmentService.java

```
package com.example.pams.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.pams.repository.AppointmentRepository;
import com.example.pams.entity.Appointment;
import java.time.LocalDate;
import java.time.LocalTime;
import java.util.*;

/**
 * Appointment Service Class
 *
 * This class contains the business logic for Appointment operations.
 * It acts as an intermediary between the Controller and Repository layers.
 *
 * Key Features:
 * - @Service annotation marks it as a Spring service component
 * - @Autowired dependency injection for repository
 * - Business logic encapsulation
 * - Transaction management (automatic with Spring)
 */
@Service  // Marks this class as a Spring service component
public class AppointmentService {

    /**
     * Appointment Repository - Injected by Spring
     *
     * @Autowired:
     * - Spring automatically injects the AppointmentRepository implementation
     * - No need to manually create repository instances
     * - Enables loose coupling between service and repository
     */
    @Autowired
    private AppointmentRepository appointmentRepo;

    /**
     * Get all appointments
     *
     * @return List of all appointments in the system
     *
     * Business Logic:
     * - Used for admin functionality
     * - System-wide appointment overview
     * - Reporting and analytics
     */
    public List<Appointment> findAll() {
       return appointmentRepo.findAll();
    }

    /**
     * Find appointment by ID
     *
     * @param id Appointment ID
     * @return Optional<Appointment> - may contain appointment or be empty
     *
     * Business Logic:
     * - Used to retrieve specific appointment information
     * - Optional return type prevents NullPointerException
     * - Safe appointment lookup for updates/cancellations
     */
    public Optional<Appointment> findById(Integer id) {
       return appointmentRepo.findById(id);
    }

    /**
     * Save or update appointment
     *
     * @param appointment Appointment object to save
     * @return Saved appointment object with generated ID
     *
     * Business Logic:
```

```
 * - Handles both new appointment creation and updates
 * - Automatic ID generation for new appointments
 * - Transaction management (automatic rollback on failure)
 * - Used for booking and status updates
 */
public Appointment save(Appointment appointment) {
   return appointmentRepo.save(appointment);
}

/**
 * Delete appointment by ID
 *
 * @param id Appointment ID to delete
 *
 * Business Logic:
 * - Removes appointment from the system
 * - Transaction management (automatic rollback on failure)
 *
 * Note: This method should be enhanced to handle:
 * - Soft delete instead of hard delete
 * - Audit trail for deletions
 * - Email notifications to patients
 */
public void deleteById(Integer id) {
   appointmentRepo.deleteById(id);
}

/**
```

```
 * Find appointments by patient ID
 *
 * @param patientId Patient ID
 * @return List of appointments for the specified patient
 *
 * Business Logic:
 * - Used for patient dashboard
 * - Patient appointment history
 * - Patient-specific appointment management
 */
public List<Appointment> findByPatientId(Integer patientId) {
   return appointmentRepo.findByPatientPatientId(patientId);
}

/**
 * Find appointments by doctor, date, and time
 *
 * @param doctorId Doctor ID
 * @param date Appointment date
 * @param time Time slot
 * @return List of appointments matching the criteria
 *
 * Business Logic:
 * - Used to check slot availability
 * - Prevents double booking
 * - Validates appointment conflicts
 * - Critical for appointment booking process
 */
```

```
  public List<Appointment> findByDoctorDateTime(Integer doctorId, LocalDate
date, LocalTime time) {
    return
appointmentRepo.findByDoctorDoctorIdAndAppointmentDateAndTimeSlot(doctor
Id, date, time);
  }

  /**
   * Find appointments by status
   *
   * @param status Appointment status (BOOKED, CANCELED, COMPLETED)
   * @return List of appointments with the specified status
   *
   * Business Logic:
   * - Used for filtering appointments
   * - Status-based reporting
   * - Admin dashboard statistics
   * - Appointment management workflows
   */
  public List<Appointment> findByStatus(String status) {
    return appointmentRepo.findByStatus(status);
  }
}
```

AdminService.java

```
package com.example.pams.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import com.example.pams.repository.AdminRepository;
```

```java
import com.example.pams.entity.Admin;

import java.util.*;


/**

 * Admin Service Class

 *

 * This class contains the business logic for Admin operations.

 * It acts as an intermediary between the Controller and Repository layers.

 *

 * Key Features:

 * - @Service annotation marks it as a Spring service component

 * - @Autowired dependency injection for repository

 * - Business logic encapsulation

 * - Transaction management (automatic with Spring)

 */

@Service  // Marks this class as a Spring service component

public class AdminService {


    /**

     * Admin Repository - Injected by Spring

     *

     * @Autowired:

     * - Spring automatically injects the AdminRepository implementation

     * - No need to manually create repository instances

     * - Enables loose coupling between service and repository

     */

    @Autowired

    private AdminRepository adminRepo;
```

```
/**
 * Authenticate admin using email and password
 *
 * @param email Admin's email address
 * @param password Admin's password
 * @return Admin object if authentication successful, null otherwise
 *
 * Business Logic:
 * - Validates admin credentials
 * - Used for admin login functionality
 * - Returns admin object for session management
 * - Role-based access control foundation
 */
public Admin findByEmailAndPassword(String email, String password) {
   return adminRepo.findByEmailAndPassword(email, password);
}

/**
 * Get all admins
 *
 * @return List of all admins in the system
 *
 * Business Logic:
 * - Used for admin management
 * - System administration
 * - Admin user listing
 */
```

```
public List<Admin> findAll() {
    return adminRepo.findAll();
}

/**
 * Find admin by ID
 *
 * @param id Admin ID
 * @return Optional<Admin> - may contain admin or be empty
 *
 * Business Logic:
 * - Used to retrieve specific admin information
 * - Optional return type prevents NullPointerException
 * - Safe admin lookup
 */
public Optional<Admin> findById(Integer id) {
    return adminRepo.findById(id);
}

/**
 * Save or update admin
 *
 * @param admin Admin object to save
 * @return Saved admin object with generated ID
 *
 * Business Logic:
 * - Handles both new admin creation and updates
 * - Automatic ID generation for new admins
```

```
     * - Transaction management (automatic rollback on failure)
     * - Used for admin user management
     */
    public Admin save(Admin admin) {
        return adminRepo.save(admin);
    }

    /**
     * Delete admin by ID
     *
     * @param id Admin ID to delete
     *
     * Business Logic:
     * - Removes admin from the system
     * - Transaction management (automatic rollback on failure)
     *
     * Note: This method should be enhanced to handle:
     * - Soft delete instead of hard delete
     * - Audit trail for deletions
     * - Prevent deletion of the last admin
     * - Role-based deletion permissions
     */
    public void deleteById(Integer id) {
        adminRepo.deleteById(id);
    }
}
```

5. Controller Classes (Web Layer)

HomeController.java

```
package com.example.pams.controller;


import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;


/**

 * Home Controller Class

 *

 * This controller handles the main landing page and home-related requests.

 * It's the entry point for users visiting the application.

 *

 * Key Features:

 * - @Controller annotation marks it as a Spring MVC controller

 * - Handles GET requests for home page

 * - Returns view names for Thymeleaf template resolution

 */

@Controller  // Marks this class as a Spring MVC controller

public class HomeController {


   /**

    * Handle home page requests

    *

    * @GetMapping({ "/", "/home" }):

    * - Maps both "/" and "/home" URLs to this method

    * - Handles GET requests only

    * - "/" is the root URL of the application

    * - "/home" is an alternative URL for the home page
    *
    * @return String "index" - Thymeleaf template name
    *
    * Business Logic:
    * - Returns the main landing page
    * - No model attributes needed for basic home page
    * - Thymeleaf
```

PatientController.java

```
package com.example.pams.controller;

import com.example.pams.entity.Patient;
import com.example.pams.service.PatientService;
import jakarta.servlet.http.HttpSession;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

/**
 * Patient Controller Class
 *
 * This controller handles all patient-related web requests.
 * It manages patient registration, login, dashboard, and logout functionality.
 *
 * Key Features:
 * - @Controller annotation marks it as a Spring MVC controller
 * - @RequestMapping("/patients") sets base URL path
 * - Session management for user authentication
 * - Model attributes for Thymeleaf templates
 */
@Controller
@RequestMapping("/patients")  // Base URL path for all patient-related endpoints
public class PatientController {

  /**
   * Patient Service - Injected by Spring
   *
   * @Autowired:
   * - Spring automatically injects the PatientService implementation
   * - Enables loose coupling between controller and service
   * - Provides access to business logic
   */
  @Autowired
  private PatientService patientService;

  /**
   * Show patient registration form
   *
   * @GetMapping("/register"):
   * - Maps to "/patients/register" URL
   * - Handles GET requests only
   *
   * @param model Spring Model object for passing data to view
   * @return String "patient-register" - Thymeleaf template name
   *
   * Business Logic:
```

```
 * - Creates a new Patient object for form binding
 * - Passes empty patient object to registration form
 * - Thymeleaf will bind form fields to patient object
 */
@GetMapping("/register")
public String showRegister(Model model) {
  model.addAttribute("patient", new Patient());  // Empty patient for form binding
  return "patient-register";  // Returns "patient-register.html" template
}

/**
 * Process patient registration
 *
 * @PostMapping("/register"):
 * - Maps to "/patients/register" URL
 * - Handles POST requests only
 *
 * @param patient Patient object populated from form data
 * @return String redirect URL
 *
 * Business Logic:
 * - Validates email uniqueness
 * - Saves new patient if email is unique
 * - Redirects to login page on success
 * - Redirects to registration page with error on failure
 */
@PostMapping("/register")
public String register(@ModelAttribute Patient patient) {
        // Check if email already exists
        if (patientService.findByEmail(patient.getEmail()) != null) {
            return "redirect:/patients/register?error";  // Redirect with error parameter
        }

        // Save new patient
        patientService.save(patient);

        // Redirect to login page
        return "redirect:/patients/login";
    }

    /**
     * Show patient login form
     *
     * @GetMapping("/login"):
     * - Maps to "/patients/login" URL
     * - Handles GET requests only
     *
     * @return String "patient-login" - Thymeleaf template name
     *
     * Business Logic:
     * - Returns login form template
     * - No model attributes needed for basic login form
     */
    @GetMapping("/login")
    public String showLogin() {
        return "patient-login";  // Returns "patient-login.html" template
```

```
}

/**
 * Process patient login
 *
 * @PostMapping("/login"):
 * - Maps to "/patients/login" URL
 * - Handles POST requests only
 *
 * @param email Patient's email from form
 * @param password Patient's password from form
 * @param session HttpSession for storing user state
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Authenticates patient credentials
 * - Stores patient ID in session on successful login
 * - Redirects to dashboard on success
 * - Shows error message on failure
 */
@PostMapping("/login")
public String login(@RequestParam String email,
          @RequestParam String password,
          HttpSession session,
          Model model) {

  // Authenticate patient
```

```
    Patient p = patientService.findByEmailAndPassword(email, password);

    if (p != null) {
        // Login successful - store patient ID in session
        session.setAttribute("patientId", p.getPatientId());
        return "redirect:/patients/dashboard";  // Redirect to dashboard
    }

    // Login failed - show error message
    model.addAttribute("error", "Invalid credentials");
    return "patient-login";  // Return to login page with error
}

/**
 * Show patient dashboard
 *
 * @GetMapping("/dashboard"):
 * - Maps to "/patients/dashboard" URL
 * - Handles GET requests only
 *
 * @param session HttpSession for checking user authentication
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Checks if patient is logged in
 * - Retrieves patient information for dashboard
 * - Redirects to login if not authenticated
```

```
     */
    @GetMapping("/dashboard")
    public String dashboard(HttpSession session, Model model) {
      // Check if patient is logged in
      Object pid = session.getAttribute("patientId");
      if (pid == null) {
        return "redirect:/patients/login";  // Redirect to login if not authenticated
      }

      // Get patient information for dashboard
      model.addAttribute("patient", patientService.findById((Integer)
  pid).orElse(null));
      return "patient-dashboard";  // Returns "patient-dashboard.html" template
    }

    /**
     * Process patient logout
     *
     * @GetMapping("/logout"):
     * - Maps to "/patients/logout" URL
     * - Handles GET requests only
     *
     * @param session HttpSession to invalidate
     * @return String redirect URL
     *
     * Business Logic:
     * - Invalidates user session
     * - Clears all session data
     * - Redirects to home page
     */
    @GetMapping("/logout")
    public String logout(HttpSession session) {
        session.invalidate();  // Clear all session data
        return "redirect:/";  // Redirect to home page
    }
}
```

DoctorController.java

```
package com.example.pams.controller;

import java.time.LocalDate;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

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.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.example.pams.entity.Doctor;
import com.example.pams.service.AppointmentService;
import com.example.pams.service.DoctorService;

import com.fasterxml.jackson.databind.ObjectMapper;


/**

 * Doctor Controller Class

 *

 * This controller handles all doctor-related web requests.

 * It manages doctor listing and availability checking functionality.

 *

 * Key Features:

 * - @Controller annotation marks it as a Spring MVC controller

 * - @RequestMapping("/doctors") sets base URL path

 * - JSON processing for doctor availability

 * - Time slot generation and booking status checking

 */

@Controller

@RequestMapping("/doctors")  // Base URL path for all doctor-related endpoints

public class DoctorController {


   /**

    * Doctor Service - Injected by Spring

    *

    * @Autowired:

    * - Spring automatically injects the DoctorService implementation

    * - Provides access to doctor business logic

    */

   @Autowired

   private DoctorService doctorService;
```

```
/**
 * Appointment Service - Injected by Spring
 *
 * @Autowired:
 * - Spring automatically injects the AppointmentService implementation
 * - Provides access to appointment business logic
 * - Used for checking slot availability
 */
@Autowired
private AppointmentService appointmentService;

/**
 * JSON Object Mapper
 *
 * Used for parsing doctor availability JSON data
 * Converts JSON string to Map for processing
 */
private ObjectMapper mapper = new ObjectMapper();

/**
 * Show list of all doctors
 *
 * @GetMapping:
 * - Maps to "/doctors" URL
 * - Handles GET requests only
 *
 * @param model Spring Model object for passing data to view
 * @return String "doctors" - Thymeleaf template name
 *
 * Business Logic:
 * - Retrieves all doctors from database
 * - Passes doctor list to view for display
 * - Used for doctor selection in appointment booking
 */
@GetMapping
public String list(Model model) {
    model.addAttribute("doctors", doctorService.findAll());  // Get all doctors
    return "doctors";  // Returns "doctors.html" template
}

/**
 * Show doctor availability and time slots
 *
 * @GetMapping("/availability/{id}"):
 * - Maps to "/doctors/availability/{id}" URL
 * - {id} is a path variable for doctor ID
 * - Handles GET requests only
 *
 * @param id Doctor ID from URL path
 * @param date Optional date parameter for availability check
 * @param model Spring Model object for passing data to view
 * @return String "doctor-availability" - Thymeleaf template name
 *
 * Business Logic:
 * - Retrieves doctor information
 * - Parses doctor availability JSON
 * - Generates time slots based on availability
 * - Checks booking status for each slot
 * - Handles date selection (defaults to today)
 */
@GetMapping("/availability/{id}")
public String availability(@PathVariable Integer id,
                    @RequestParam(required = false) String date,
                    Model model) {

  // Get doctor information
  Doctor d = doctorService.findById(id).orElse(null);
  model.addAttribute("doctor", d);

  // List to store time slots with booking status
  List<Map<String, Object>> slots = new ArrayList<>();

  try {
    // Check if doctor exists and has availability data
    if (d != null && d.getAvailability() != null && !d.getAvailability().isBlank()) {

      // Parse availability JSON
      Map m = mapper.readValue(d.getAvailability(), Map.class);
      String start = (String) m.get("start");  // Start time (e.g., "09:00")
      String end = (String) m.get("end");     // End time (e.g., "17:00")

      // Parse times
      LocalTime st = LocalTime.parse(start);
```

```
            LocalTime en = LocalTime.parse(end);

            // Determine date (default to today if not provided)
            LocalDate day = date == null ? LocalDate.now() : LocalDate.parse(date);

            // Generate time slots (15-minute intervals)
            for (LocalTime t = st; !t.isAfter(en.minusMinutes(15)); t = t.plusMinutes(15)) {
              LocalDate dt = day;

              // Check if slot is already booked
              boolean booked =
!appointmentService.findByDoctorDateTime(d.getDoctorId(), dt, t).isEmpty();

              // Create slot information map
              Map<String, Object> s = new HashMap<>();
              s.put("time", t.toString());     // Time slot (e.g., "09:00")
              s.put("date", dt.toString());    // Date (e.g., "2025-01-15")
              s.put("booked", booked);         // Booking status

              slots.add(s);
            }
          }
        } catch (Exception e) {
          // Handle JSON parsing errors
          e.printStackTrace();
        }

        // Pass slots to view
        model.addAttribute("slots", slots);
        return "doctor-availability";  // Returns "doctor-availability.html" template
    }
}
```

AppointmentController.java

```
package com.example.pams.controller;

import java.time.LocalDate;
import java.time.LocalTime;
import java.util.List;
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.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.example.pams.entity.Appointment;
import com.example.pams.entity.Doctor;
import com.example.pams.entity.Patient;
import com.example.pams.service.AppointmentService;
import com.example.pams.service.DoctorService;
import com.example.pams.service.PatientService;
import jakarta.servlet.http.HttpSession;

/**
```

```
 * Appointment Controller Class
 *
 * This controller handles all appointment-related web requests.
 * It manages appointment booking, viewing, and cancellation functionality.
 *
 * Key Features:
 * - @Controller annotation marks it as a Spring MVC controller
 * - @RequestMapping("/appointments") sets base URL path
 * - Session management for patient authentication
 * - Conflict prevention for appointment booking
 * - Status management for appointments
 */
@Controller
@RequestMapping("/appointments")  // Base URL path for all appointment-related
endpoints
public class AppointmentController {

  /**
   * Appointment Service - Injected by Spring
   *
   * @Autowired:
   * - Spring automatically injects the AppointmentService implementation
   * - Provides access to appointment business logic
   */
  @Autowired
  private AppointmentService appointmentService;

  /**
   * Patient Service - Injected by Spring
   *
   * @Autowired:
   * - Spring automatically injects the PatientService implementation
   * - Provides access to patient business logic
   */
  @Autowired
  private PatientService patientService;

  /**
   * Doctor Service - Injected by Spring
   *
   * @Autowired:
   * - Spring automatically injects the DoctorService implementation
   * - Provides access to doctor business logic
   */
  @Autowired
  private DoctorService doctorService;

  /**
   * Show appointment booking page
   *
   * @GetMapping("/book"):
   * - Maps to "/appointments/book" URL
   * - Handles GET requests only
   *
   * @param session HttpSession for checking user authentication
   * @param model Spring Model object for passing data to view
```

```java
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Checks if patient is logged in
 * - Retrieves all doctors for selection
 * - Redirects to login if not authenticated
 */
@GetMapping("/book")
public String bookPage(HttpSession session, Model model) {
  // Check if patient is logged in
  if (session.getAttribute("patientId") == null) {
    return "redirect:/patients/login";  // Redirect to login if not authenticated
  }

  // Get all doctors for selection
  model.addAttribute("doctors", doctorService.findAll());
  return "appointment-book";  // Returns "appointment-book.html" template
}

/**
 * Show available time slots for a doctor on a specific date
 *
 * @GetMapping("/slots"):
 * - Maps to "/appointments/slots" URL
 * - Handles GET requests only
 *
 * @param doctorId Doctor ID from request parameter
 * @param date Date from request parameter
 * @param model Spring Model object for passing data to view
 * @return String "appointment-slots" - Thymeleaf template name
 *
 * Business Logic:
 * - Retrieves doctor information
 * - Parses doctor availability JSON
 * - Generates time slots based on availability
 * - Checks booking status for each slot
 * - Handles JSON parsing errors gracefully
 */
@GetMapping("/slots")
public String slots(@RequestParam Integer doctorId,
             @RequestParam String date,
             Model model) {

  // Get doctor information
  Doctor d = doctorService.findById(doctorId).orElse(null);
  model.addAttribute("doctor", d);

  // List to store time slots with booking status
  List<java.util.Map<String, Object>> show = new java.util.ArrayList<>();

  try {
    // Check if doctor exists and has availability data
    if (d != null && d.getAvailability() != null && !d.getAvailability().isBlank()) {

      // Parse availability JSON
```

```
            com.fasterxml.jackson.databind.ObjectMapper mapper = new
com.fasterxml.jackson.databind.ObjectMapper();
            java.util.Map m = mapper.readValue(d.getAvailability(), java.util.Map.class);

            // Parse start and end times
            java.time.LocalTime st = java.time.LocalTime.parse((String) m.get("start"));
            java.time.LocalTime en = java.time.LocalTime.parse((String) m.get("end"));
            java.time.LocalDate day = java.time.LocalDate.parse(date);

            // Generate time slots (15-minute intervals)
            for (java.time.LocalTime t = st; !t.isAfter(en.minusMinutes(15)); t =
t.plusMinutes(15)) {

              // Check if slot is already booked
              boolean booked = !appointmentService.findByDoctorDateTime(doctorId,
day, t).isEmpty();

              // Create slot information map
              java.util.Map<String, Object> map = new java.util.HashMap<>();
              map.put("date", day.toString());    // Date (e.g., "2025-01-15")
              map.put("time", t.toString());      // Time slot (e.g., "09:00")
              map.put("booked", booked);         // Booking status

              show.add(map);
            }
          }
      } catch (Exception e) {
          // Handle JSON parsing errors
          e.printStackTrace();
```

```
    }

    // Pass slots to view
    model.addAttribute("slots", show);
    return "appointment-slots";  // Returns "appointment-slots.html" template
}

/**
 * Save new appointment
 *
 * @PostMapping("/save"):
 * - Maps to "/appointments/save" URL
 * - Handles POST requests only
 *
 * @param doctorId Doctor ID from form
 * @param date Appointment date from form
 * @param time Time slot from form
 * @param session HttpSession for getting patient ID
 * @return String redirect URL
 *
 * Business Logic:
 * - Checks if patient is logged in
 * - Validates appointment date and time
 * - Checks for existing appointments to prevent conflicts
 * - Creates new appointment if slot is available
 * - Redirects to appointment list on success
 */
@PostMapping("/save")
```

```
  public String save(@RequestParam Integer doctorId,
           @RequestParam String date,
           @RequestParam String time,
           HttpSession session) {

    // Check if patient is logged in
    Object pid = session.getAttribute("patientId");
    if (pid == null) {
      return "redirect:/patients/login";  // Redirect to login if not authenticated
    }

    // Parse date and time
    LocalDate dt = LocalDate.parse(date);
    LocalTime tm = LocalTime.parse(time);

    // Check for existing appointments to prevent conflicts
    List<Appointment> existing =
appointmentService.findByDoctorDateTime(doctorId, dt, tm);
    boolean booked = false;

    // Check if any existing appointment is still booked (not canceled)
    for (Appointment a : existing) {
      if (a.getStatus() == Appointment.Status.BOOKED) {
        booked = true;
        break;
      }
    }
```

```
        // If slot is already booked, redirect with error
        if (booked) {
            return "redirect:/appointments/book?error";
        }

        // Create new appointment
        Appointment a = new Appointment();

        // Get patient and doctor information
        Patient p = patientService.findById((Integer) pid).orElse(null);
        Doctor d = doctorService.findById(doctorId).orElse(null);

        // Set appointment details
        a.setPatient(p);
        a.setDoctor(d);
        a.setAppointmentDate(dt);
        a.setTimeSlot(tm);
        a.setStatus(Appointment.Status.BOOKED);  // Default status is BOOKED

        // Save appointment
        appointmentService.save(a);

        // Redirect to patient's appointment list
        return "redirect:/appointments/my";
    }

    /**
     * Show patient's appointments
```

```
 *
 * @GetMapping("/my"):
 * - Maps to "/appointments/my" URL
 * - Handles GET requests only
 *
 * @param session HttpSession for checking user authentication
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Checks if patient is logged in
 * - Retrieves all appointments for the logged-in patient
 * - Redirects to login if not authenticated
 */
@GetMapping("/my")
public String myAppointments(HttpSession session, Model model) {
    // Check if patient is logged in
    Object pid = session.getAttribute("patientId");
    if (pid == null) {
        return "redirect:/patients/login";  // Redirect to login if not authenticated
    }

    // Get all appointments for the patient
    List<Appointment> list = appointmentService.findByPatientId((Integer) pid);
    model.addAttribute("appointments", list);

    return "appointment-list";  // Returns "appointment-list.html" template
}
```

```
/**
 * Cancel an appointment
 *
 * @GetMapping("/cancel/{id}"):
 * - Maps to "/appointments/cancel/{id}" URL
 * - {id} is a path variable for appointment ID
 * - Handles GET requests only
 *
 * @param id Appointment ID from URL path
 * @param session HttpSession for checking user authentication
 * @return String redirect URL
 *
 * Business Logic:
 * - Checks if patient is logged in
 * - Finds the appointment by ID
 * - Changes appointment status to CANCELED
 * - Saves the updated appointment
 * - Redirects to appointment list
 */
@GetMapping("/cancel/{id}")
public String cancel(@PathVariable Integer id, HttpSession session) {
  // Check if patient is logged in
  Object pid = session.getAttribute("patientId");
  if (pid == null) {
    return "redirect:/patients/login";  // Redirect to login if not authenticated
  }
```

```
        // Find appointment by ID
        Appointment a = appointmentService.findById(id).orElse(null);

        if (a != null) {
            // Change status to CANCELED
            a.setStatus(Appointment.Status.CANCELED);

            // Save updated appointment
            appointmentService.save(a);
        }

        // Redirect to appointment list
        return "redirect:/appointments/my";
    }
}
```

AdminController.java

```
package com.example.pams.controller;

import java.util.HashMap;
import java.util.Map;

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.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;


import com.example.pams.entity.Admin;

import com.example.pams.entity.Appointment;

import com.example.pams.entity.Doctor;

import com.example.pams.service.AdminService;

import com.example.pams.service.AppointmentService;

import com.example.pams.service.DoctorService;

import com.fasterxml.jackson.databind.ObjectMapper;


import jakarta.servlet.http.HttpSession;


/**

 * Admin Controller Class

 *

 * This controller handles all admin-related web requests.

 * It manages admin authentication, doctor management, and system
administration.

 *

 * Key Features:

 * - @Controller annotation marks it as a Spring MVC controller

 * - @RequestMapping("/admin") sets base URL path

 * - Session management for admin authentication

 * - JSON processing for doctor availability

 * - System-wide appointment management

 */

@Controller

@RequestMapping("/admin")  // Base URL path for all admin-related endpoints
```

```
public class AdminController {

    /**
     * Admin Service - Injected by Spring
     *
     * @Autowired:
     * - Spring automatically injects the AdminService implementation
     * - Provides access to admin business logic
     */
    @Autowired
    private AdminService adminService;

    /**
     * Doctor Service - Injected by Spring
     *
     * @Autowired:
     * - Spring automatically injects the DoctorService implementation
     * - Provides access to doctor business logic
     */
    @Autowired
    private DoctorService doctorService;

    /**
     * Appointment Service - Injected by Spring
     *
     * @Autowired:
     * - Spring automatically injects the AppointmentService implementation
     * - Provides access to appointment business logic
```

```
 */
@Autowired
private AppointmentService appointmentService;

/**
 * JSON Object Mapper
 *
 * Used for creating doctor availability JSON data
 * Converts Map to JSON string for storage
 */
private ObjectMapper mapper = new ObjectMapper();

/**
 * Show admin login form
 *
 * @GetMapping("/login"):
 * - Maps to "/admin/login" URL
 * - Handles GET requests only
 *
 * @return String "admin-login" - Thymeleaf template name
 *
 * Business Logic:
 * - Returns admin login form template
 * - No model attributes needed for basic login form
 */
@GetMapping("/login")
public String loginPage() {
  return "admin-login";  // Returns "admin-login.html" template
```

```
}

/**
 * Process admin login
 *
 * @PostMapping("/login"):
 * - Maps to "/admin/login" URL
 * - Handles POST requests only
 *
 * @param email Admin's email from form
 * @param password Admin's password from form
 * @param session HttpSession for storing admin state
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Authenticates admin credentials
 * - Stores admin ID in session on successful login
 * - Redirects to dashboard on success
 * - Shows error message on failure
 */
@PostMapping("/login")
public String login(@RequestParam String email,
            @RequestParam String password,
            HttpSession session,
            Model model) {

  // Authenticate admin
```

```
    Admin a = adminService.findByEmailAndPassword(email, password);

    if (a != null) {
        // Login successful - store admin ID in session
        session.setAttribute("adminId", a.getAdminId());
        return "redirect:/admin/dashboard";  // Redirect to dashboard
    }

    // Login failed - show error message
    model.addAttribute("error", "Invalid credentials");
    return "admin-login";  // Return to login page with error
}

/**
 * Show admin dashboard
 *
 * @GetMapping("/dashboard"):
 * - Maps to "/admin/dashboard" URL
 * - Handles GET requests only
 *
 * @param session HttpSession for checking admin authentication
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Checks if admin is logged in
 * - Retrieves all doctors for management
 * - Provides empty doctor object for adding new doctors
```

```
   * - Redirects to login if not authenticated
   */
  @GetMapping("/dashboard")
  public String dashboard(HttpSession session, Model model) {
    // Check if admin is logged in
    if (session.getAttribute("adminId") == null) {
      return "redirect:/admin/login";  // Redirect to login if not authenticated
    }

    // Prepare data for dashboard
    model.addAttribute("doctor", new Doctor());       // Empty doctor for adding new
doctors
    model.addAttribute("doctors", doctorService.findAll());  // All doctors for
management

    return "admin-dashboard";  // Returns "admin-dashboard.html" template
  }

  /**
   * Add new doctor
   *
   * @PostMapping("/addDoctor"):
   * - Maps to "/admin/addDoctor" URL
   * - Handles POST requests only
   *
   * @param name Doctor's name from form
   * @param specialization Doctor's specialization from form
   * @param email Doctor's email from form
   * @param phone Doctor's phone from form
```

```
 * @param start Doctor's start time from form
 * @param end Doctor's end time from form
 * @param session HttpSession for checking admin authentication
 * @return String redirect URL
 *
 * Business Logic:
 * - Checks if admin is logged in
 * - Creates new doctor object
 * - Converts availability times to JSON format
 * - Saves doctor to database
 * - Redirects to dashboard
 */
@PostMapping("/addDoctor")
public String addDoctor(@RequestParam String name,
                        @RequestParam String specialization,
                        @RequestParam String email,
                        @RequestParam String phone,
                        @RequestParam String start,
                        @RequestParam String end,
                        HttpSession session) {

  // Check if admin is logged in
  if (session.getAttribute("adminId") == null) {
    return "redirect:/admin/login";  // Redirect to login if not authenticated
  }

  try {
    // Create new doctor object
```

```
        Doctor d = new Doctor();

        d.setName(name);

        d.setSpecialization(specialization);

        d.setEmail(email);

        d.setPhone(phone);


        // Create availability JSON

        Map<String, String> m = new HashMap<>();

        m.put("start", start);  // Start time (e.g., "09:00")

        m.put("end", end);     // End time (e.g., "17:00")


        // Convert to JSON string

        d.setAvailability(mapper.writeValueAsString(m));


        // Save doctor

        doctorService.save(d);


    } catch (Exception e) {

        // Handle JSON conversion errors

        e.printStackTrace();

    }


    // Redirect to dashboard

    return "redirect:/admin/dashboard";

}


/**

 * Show all appointments (admin view)
 *
 * @GetMapping("/appointments"):
 * - Maps to "/admin/appointments" URL
 * - Handles GET requests only
 *
 * @param session HttpSession for checking admin authentication
 * @param model Spring Model object for passing data to view
 * @return String view name or redirect URL
 *
 * Business Logic:
 * - Checks if admin is logged in
 * - Retrieves all appointments in the system
 * - Provides system-wide appointment overview
 * - Redirects to login if not authenticated
 */
@GetMapping("/appointments")
public String allAppointments(HttpSession session, Model model) {
  // Check if admin is logged in
  if (session.getAttribute("adminId") == null) {
    return "redirect:/admin/login";  // Redirect to login if not authenticated
  }

  // Get all appointments
  model.addAttribute("appointments", appointmentService.findAll());

  return "admin-appointments";  // Returns "admin-appointments.html"
template
}
```

```
/**
 * Cancel an appointment (admin action)
 *
 * @GetMapping("/cancel/{id}"):
 * - Maps to "/admin/cancel/{id}" URL
 * - {id} is a path variable for appointment ID
 * - Handles GET requests only
 *
 * @param id Appointment ID from URL path
 * @param session HttpSession for checking admin authentication
 * @return String redirect URL
 *
 * Business Logic:
 * - Checks if admin is logged in
 * - Finds the appointment by ID
 * - Changes appointment status to CANCELED
 * - Saves the updated appointment
 * - Redirects to admin appointments list
 */
@GetMapping("/cancel/{id}")
public String cancel(@PathVariable Integer id, HttpSession session) {
  // Check if admin is logged in
  if (session.getAttribute("adminId") == null) {
    return "redirect:/admin/login";  // Redirect to login if not authenticated
  }

  // Find appointment by ID
```

```
    Appointment a = appointmentService.findById(id).orElse(null);

    if (a != null) {
      // Change status to CANCELED
      a.setStatus(Appointment.Status.CANCELED);

      // Save updated appointment
      appointmentService.save(a);
    }

    // Redirect to admin appointments list
    return "redirect:/admin/appointments";
  }

  /**
   * Process admin logout
   *
   * @GetMapping("/logout"):
   * - Maps to "/admin/logout" URL
   * - Handles GET requests only
   *
   * @param session HttpSession to invalidate
   * @return String redirect URL
   *
   * Business Logic:
   * - Invalidates admin session
   * - Clears all session data
   * - Redirects to admin login page
```

```
     */
    @GetMapping("/logout")
    public String logout(HttpSession session) {
        session.invalidate();  // Clear all session data
        return "redirect:/admin/login";  // Redirect to admin login page
    }
}
```