**Case Study: Hospital Management System using Spring Boot (Monolithic Architecture)**

**Overview**

The **Hospital Management System** (HMS) is a monolithic backend service built using **Spring Boot**. The system allows patients to book appointments with doctors. It follows best practices like using **DTOs (Data Transfer Objects)**, **ModelMapper for data mapping**, **Spring Validation**, and **Global Exception Handling**.

**Technologies Used**

* **Spring Boot** (Framework)
* **Spring Data JPA** (Persistence Layer)
* **Spring Validation** (Input Validation)
* **ModelMapper** (Data Mapping)
* **H2/PostgreSQL** (Database)
* **Lombok** (Boilerplate Code Reduction)
* **Spring Web** (REST API Development)
* **Spring Boot Starter Validation** (Validation)

**System Components**

1. **Patient Management**
2. **Doctor Management**
3. **Appointment Booking**

**Project Structure**

HospitalManagementSystem/

|-- src/main/java/com/hms/

|-- controller/

|-- service/

|-- impl/

|-- repository/

|-- entity/

|-- dto/

|-- exception/

|-- config/

|-- pom.xml

**Step 1: Define Entities**

**Patient Entity**

@Entity

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Patient {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "Patient name is required")

private String name;

@Email(message = "Invalid email format")

private String email;

@NotBlank(message = "Contact number is required")

private String contactNumber;

}

**Doctor Entity**

@Entity

@Data

@NoArgsConstructor

@AllArgsConstructor

public class Doctor {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

@NotBlank(message = "Doctor name is required")

private String name;

@NotBlank(message = "Specialization is required")

private String specialization;

}

**Step 2: Define DTOs**

**Patient DTOs**

@Data

@NoArgsConstructor

@AllArgsConstructor

public class PatientRequestDTO {

private String name;

private String email;

private String contactNumber;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

public class PatientResponseDTO {

private Long id;

private String name;

private String email;

private String contactNumber;

}

**Doctor DTOs**

@Data

@NoArgsConstructor

@AllArgsConstructor

public class DoctorRequestDTO {

private String name;

private String specialization;

}

@Data

@NoArgsConstructor

@AllArgsConstructor

public class DoctorResponseDTO {

private Long id;

private String name;

private String specialization;

}

**Step 3: Define Service Interfaces**

public interface PatientService {

PatientResponseDTO addPatient(PatientRequestDTO patientRequestDTO);

PatientResponseDTO getPatientById(Long id);

List<PatientResponseDTO> getAllPatients();

PatientResponseDTO updatePatient(Long id, PatientRequestDTO patientRequestDTO);

void deletePatient(Long id);

}

public interface DoctorService {

DoctorResponseDTO addDoctor(DoctorRequestDTO doctorRequestDTO);

DoctorResponseDTO getDoctorById(Long id);

List<DoctorResponseDTO> getAllDoctors();

DoctorResponseDTO updateDoctor(Long id, DoctorRequestDTO doctorRequestDTO);

void deleteDoctor(Long id);

}

**Step 4: Implement Service Classes**

**Patient Service Implementation**

@Service

@RequiredArgsConstructor

public class PatientServiceImpl implements PatientService {

private final PatientRepository patientRepository;

private final ModelMapper modelMapper;

@Override

public PatientResponseDTO addPatient(PatientRequestDTO patientRequestDTO) {

Patient patient = modelMapper.map(patientRequestDTO, Patient.class);

patient = patientRepository.save(patient);

return modelMapper.map(patient, PatientResponseDTO.class);

}

@Override

public PatientResponseDTO getPatientById(Long id) {

Patient patient = patientRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Patient not found"));

return modelMapper.map(patient, PatientResponseDTO.class);

}

@Override

public List<PatientResponseDTO> getAllPatients() {

return patientRepository.findAll().stream()

.map(patient -> modelMapper.map(patient, PatientResponseDTO.class))

.collect(Collectors.toList());

}

@Override

public PatientResponseDTO updatePatient(Long id, PatientRequestDTO patientRequestDTO) {

Patient patient = patientRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Patient not found"));

modelMapper.map(patientRequestDTO, patient);

patient = patientRepository.save(patient);

return modelMapper.map(patient, PatientResponseDTO.class);

}

@Override

public void deletePatient(Long id) {

patientRepository.deleteById(id);

}

}

**Doctor Service Implementation**

@Service

@RequiredArgsConstructor

public class DoctorServiceImpl implements DoctorService {

private final DoctorRepository doctorRepository;

private final ModelMapper modelMapper;

@Override

public DoctorResponseDTO addDoctor(DoctorRequestDTO doctorRequestDTO) {

Doctor doctor = modelMapper.map(doctorRequestDTO, Doctor.class);

doctor = doctorRepository.save(doctor);

return modelMapper.map(doctor, DoctorResponseDTO.class);

}

@Override

public DoctorResponseDTO getDoctorById(Long id) {

Doctor doctor = doctorRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Doctor not found"));

return modelMapper.map(doctor, DoctorResponseDTO.class);

}

@Override

public List<DoctorResponseDTO> getAllDoctors() {

return doctorRepository.findAll().stream()

.map(doctor -> modelMapper.map(doctor, DoctorResponseDTO.class))

.collect(Collectors.toList());

}

@Override

public DoctorResponseDTO updateDoctor(Long id, DoctorRequestDTO doctorRequestDTO) {

Doctor doctor = doctorRepository.findById(id)

.orElseThrow(() -> new RuntimeException("Doctor not found"));

modelMapper.map(doctorRequestDTO, doctor);

doctor = doctorRepository.save(doctor);

return modelMapper.map(doctor, DoctorResponseDTO.class);

}

@Override

public void deleteDoctor(Long id) {

doctorRepository.deleteById(id);

}

}

**Conclusion**

This **monolithic Spring Boot application** provides a backend service for hospital management, focusing on **DTO usage, ModelMapper for mapping, validation, and exception handling**. This structured approach ensures scalability, maintainability, and robustness.