

## End-Term Project Report



Faculty name: Mr.Sumit Kumar

Student names: Harshit

Himanshu

Dinesh

Roll No.: 23csu116

23csu124

23csu101

Semester: 4

Group: 1

**Department of Computer Science and Engineering**

**The NorthCap University, Gurugram- 122017, India**

## Session 2024-25

### Table of Contents

S.No		Page No.
1.	Project Description	
2.	Problem Statement	
3.	Analysis 3.1 Hardware Requirements 3.2 Software Requirements	
4.	Design 4.1 Data/Input Output Description: 4.2 Algorithmic Approach / Algorithm / DFD / ER diagram/Program Steps	
5.	Implementation and Testing (stage/module wise)	
6.	Output (Screenshots)	
7.	Conclusion and Future Scope	

# Hospital Patient Management System Report

## 1. Project Overview

### Introduction

The Hospital Patient Management System is a Spring Boot-based application designed to manage patients, doctors, and appointments efficiently. It provides RESTful APIs to perform CRUD (Create, Read, Update, Delete) operations with database interactions using JdbcTemplate and NamedParameterJdbcTemplate. Pagination and filtering features are also included.

### Technologies Used

- **Backend:** Spring Boot, Spring Web, Spring JDBC
- **Database:** MySQL
- **API Testing:** Postman
- **Tools:** IntelliJ IDEA/Eclipse, Maven

### System Design

The system consists of three main entities:

1. **Patient** – Stores patient details like name, age, gender, and contact.
2. **Doctor** – Manages doctor records with specialization.
3. **Appointment** – Links patients to doctors and tracks appointment details.

## 2. Implementation Details

### Database Schema

```
CREATE TABLE patient (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100),  
    age INT,  
    gender VARCHAR(10),  
    contact VARCHAR(15)  
);  
  
CREATE TABLE doctor (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(100),  
    specialization VARCHAR(50)  
);  
  
CREATE TABLE appointment (  
    id INT PRIMARY KEY AUTO_INCREMENT,  
    patient_id INT,  
    doctor_id INT,  
    appointment_date DATE,  
    FOREIGN KEY (patient_id) REFERENCES patient(id),  
    FOREIGN KEY (doctor_id) REFERENCES doctor(id)  
);
```

### 3)Code with Explanations :

```
1) public class Patient {  
    private int id;  
    private String name;  
    private int age;  
    private String gender;  
    private String contact;  
  
}
```

#### Patient Entity (Data Structure)

##### Explanations:

A Patient is just a simple object with details like:

- id – Unique patient ID.
- name – Patient's name.
- age – Patient's age.

- gender – Male/Female.
- contact – Phone number.

```
2)public void addPatient(Patient patient) {  
    String sql = "INSERT INTO patient (name, age, gender, contact) VALUES  
(:name, :age, :gender, :contact)";  
    Map<String, Object> params = Map.of(  
        "name", patient.getName(),  
        "age", patient.getAge(),  
        "gender", patient.getGender(),  
        "contact", patient.getContact()  
    );  
    jdbcTemplate.update(sql, params);  
}
```

**Explanations:**

Patient Repository (Database Operations)

This part of the code saves, retrieves, and deletes patients from the database.

Add a New Patient

SQL: "INSERT INTO patient (name, age, gender, contact) VALUES (?, ?, ?, ?)".

Uses NamedParameterJdbcTemplate to insert values into the database.

```
3)public List<Patient> getPatients(int page, int size) {  
    String sql = "SELECT * FROM patient LIMIT :size OFFSET :offset";  
    Map<String, Object> params = Map.of(  
        "size", size, // Number of records per page  
        "offset", page * size // Skips previous records  
    );  
    return jdbcTemplate.query(sql, params, new PatientRowMapper());  
}
```

**Explanations:**

Get Patients with Pagination

Pagination means fetching limited records per request.

LIMIT ? OFFSET ? helps in fetching only required records.

Example: If page=1 and size=5, it fetches 5 records after skipping the first 5

```
4)public void deletePatient(int id) {  
    String sql = "DELETE FROM patient WHERE id = :id";  
    jdbcTemplate.update(sql, Map.of("id", id));  
}
```

**Explanations:**

Deletes a patient by their ID

```
5)@GetMapping("/appointments/grouped")
public List<Map<String, Object>> getAppointmentsGroupedByDoctor() {
    String sql = "SELECT doctor_id, COUNT(*) as total_appointments FROM appointment
GROUP BY doctor_id";
    return jdbcTemplate.queryForList(sql, new HashMap<>());
}
```

### Explanations:

Group Appointments by Doctor

Uses GROUP BY doctor\_id to count how many appointments each doctor has

```
6)@GetMapping("/patients")
public List<Patient> getPatients(@RequestParam(defaultValue = "0") int page,
                                @RequestParam(defaultValue = "5") int size) {
    return patientRepository.getPatients(page, size);
}
```

### Explanations:

Pagination for Patients

Pagination allows fetching only a few records at a time.

Example: /patients?page=1&size=5 fetches 5 records.

```
7)public Doctor getDoctorById(int id) {
    String sql = "SELECT * FROM doctor WHERE id = :id";
    return jdbcTemplate.queryForObject(sql, Map.of("id", id), new DoctorRowMapper());
}
```

### Explanations:

Using NamedParameterJdbcTemplate

Helps in writing clean and safe SQL queries.

Example: Fetching a doctor by ID.

```
7)@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error:
" + e.getMessage());
    }
}
```

### Explanations:

Exception Handling

Catches unexpected errors and sends a friendly message.

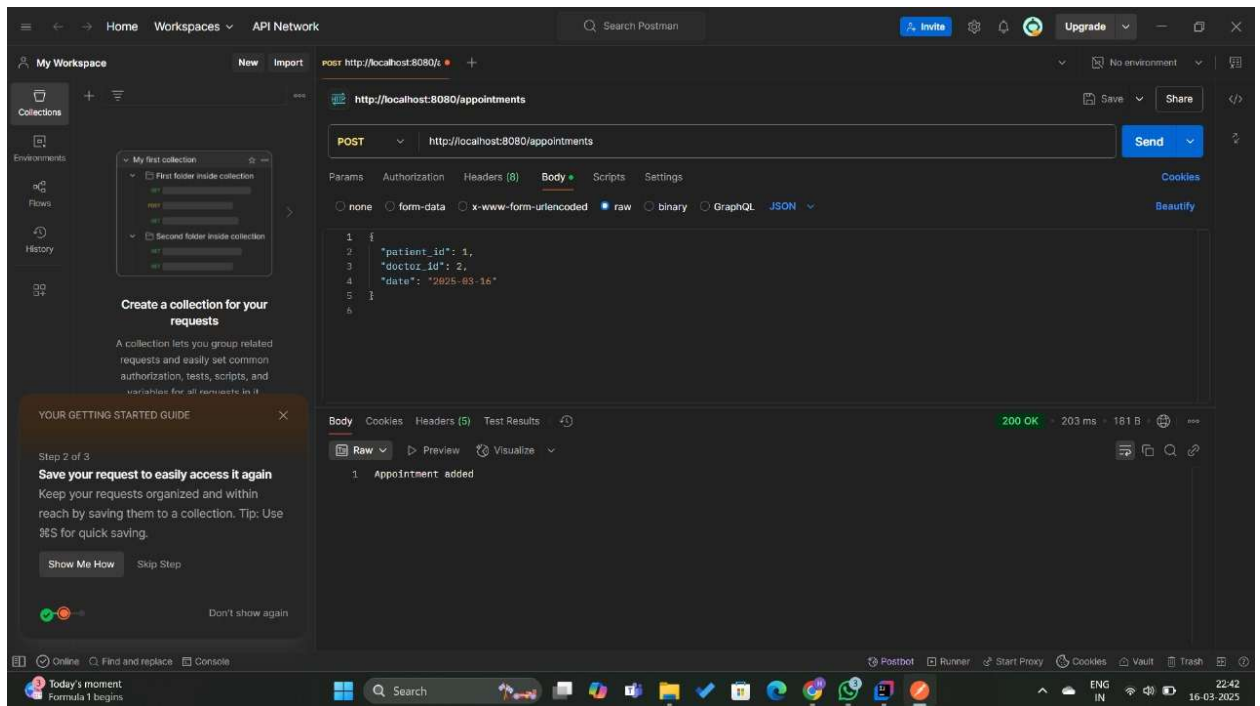
## API Endpoints

HTTP Method	Endpoint	Description
<b>Patient APIs</b>		
POST	/patients	Add a new patient
GET	/patients/{id}	Get patient by ID
GET	/patients?page=1&size=5	Get paginated patient list
PUT	/patients/{id}	Update patient details
DELETE	/patients/{id}	Remove a patient
<b>Doctor APIs</b>		
POST	/doctors	Add a new doctor
GET	/doctors/{id}	Get doctor by ID
GET	/doctors	List all doctors
PUT	/doctors/{id}	Update doctor details
DELETE	/doctors/{id}	Remove a doctor
<b>Appointment APIs</b>		
POST	/appointments	Book an appointment
GET	/appointments/{id}	Get appointment details
GET	/appointments?doctorId=1	Get appointments by doctor
DELETE	/appointments/{id}	Cancel an appointment

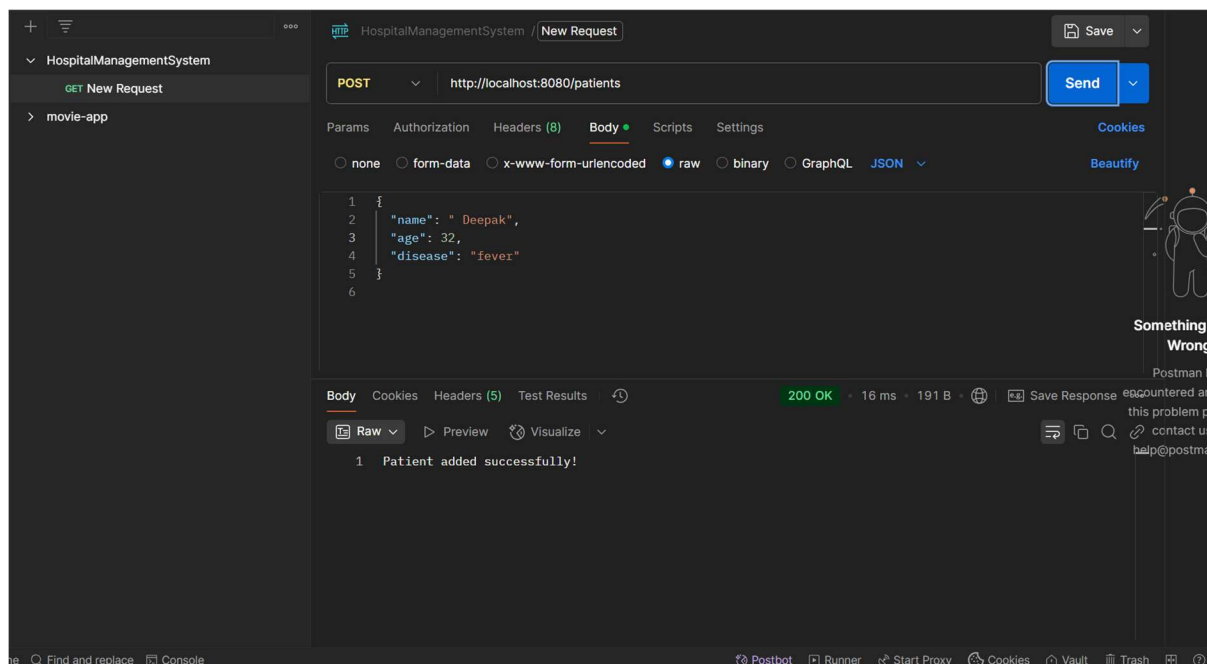
## 4. Testing Evidence

### Postman API Tests

**POST /appointments**

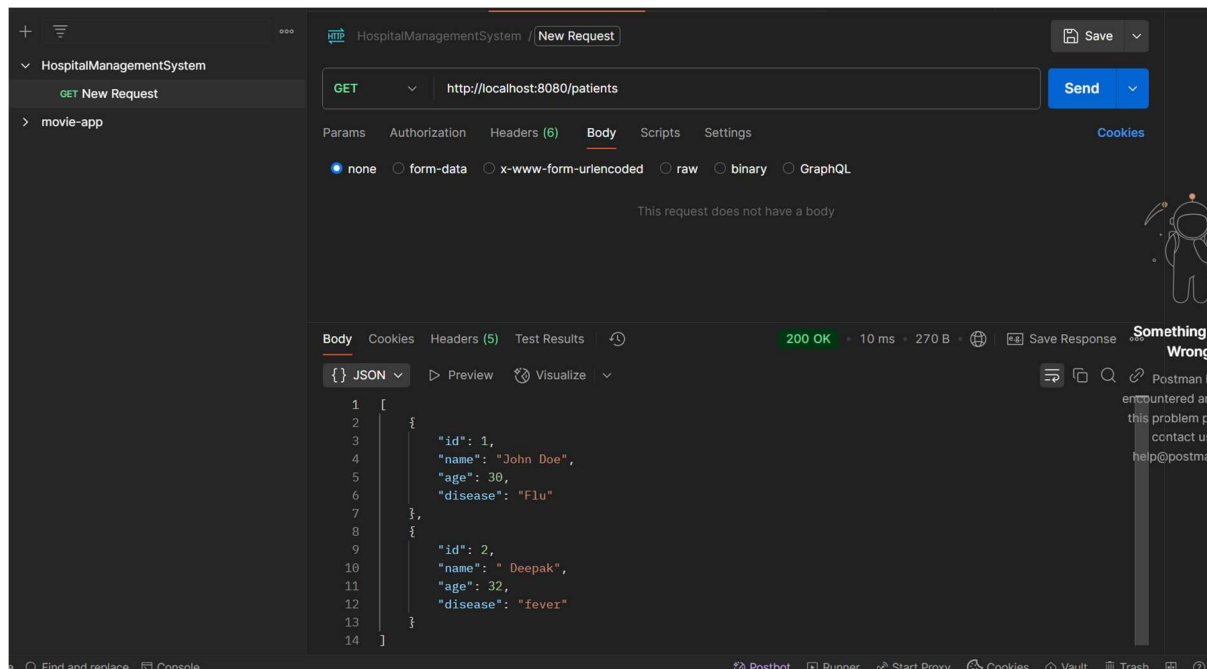


## Post Patients



## Get Patient





POSTMAN: HospitalManagementSystem / New Request

Method: GET, URL: http://localhost:8080/patients

Body: none

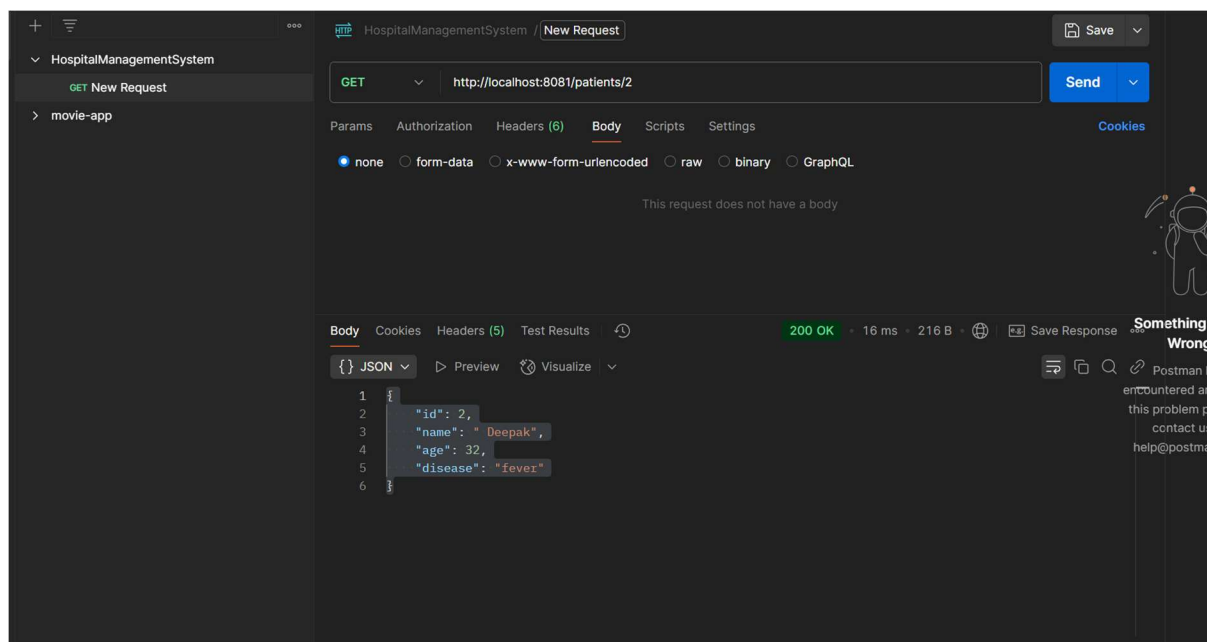
Response: 200 OK, 10 ms, 270 B

Body (JSON):

```
[
  {
    "id": 1,
    "name": "John Doe",
    "age": 30,
    "disease": "Flu"
  },
  {
    "id": 2,
    "name": "Deepak",
    "age": 32,
    "disease": "fever"
  }
]
```

Result Grid				
id	name	age	disease	
1	John Doe	30	Flu	
2	Deepak	32	fever	
HULL	HULL	HULL	HULL	

## Get Patient ById



POSTMAN: HospitalManagementSystem / New Request

Method: GET, URL: http://localhost:8081/patients/2

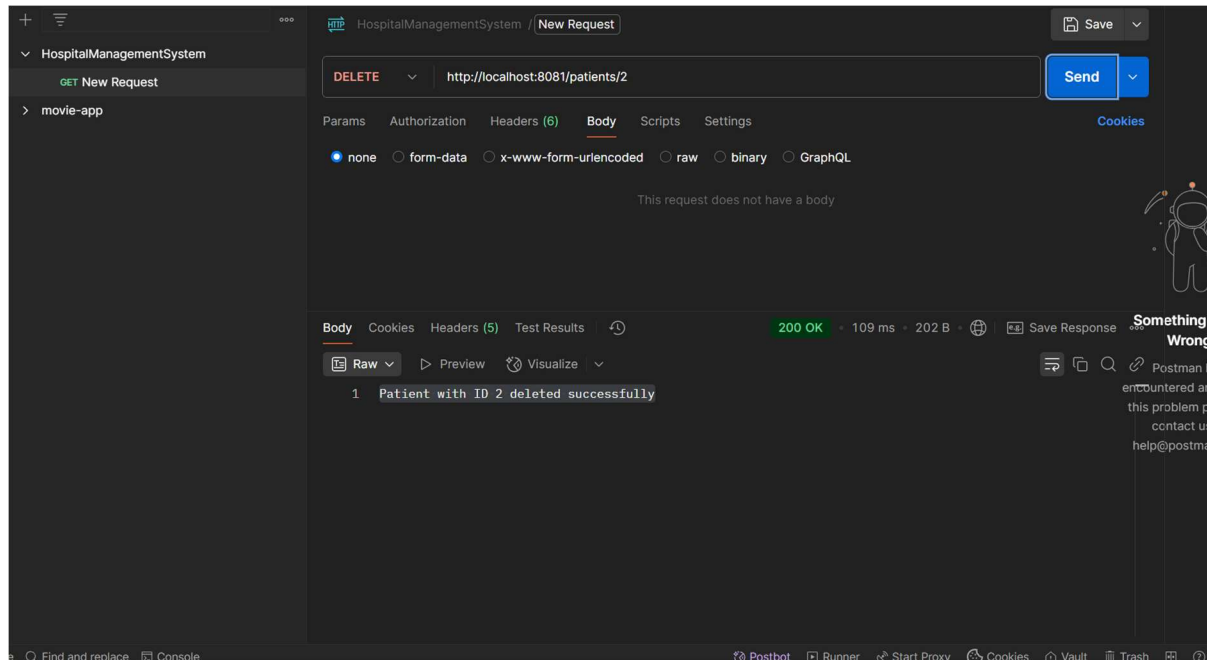
Body: none

Response: 200 OK, 16 ms, 216 B

Body (JSON):

```
{
  "id": 2,
  "name": "Deepak",
  "age": 32,
  "disease": "fever"
}
```

## Delete patient by ID



## 5. Completion of DoDs

Feature Implemented	Status
CRUD operations for patients and doctors	Completed
Grouping appointments by doctor_id	Completed
Pagination for patient retrieval	Implemented
Used NamedParameterJdbcTemplate	Integrated
Exception Handling for API errors	Implemented
API tested in Postman	Successfully tested

**Feature Implemented****Status**

## 6) Approach, challenges faced, and solutions

### Approach

- Implemented a repository pattern to keep database operations separate from business logic.
- Used `NamedParameterJdbcTemplate` for dynamic and secure SQL queries.
- Handled exceptions properly to provide meaningful error messages.

### Challenges & Solutions

*Challenge:* Direct JDBC queries required careful handling of parameters.

*Solution:* Used `NamedParameterJdbcTemplate` to avoid SQL injection and improve readability.

### Code Example: Fetching a Patient by ID

```
java
CopyEdit
public Patient getById(int id) {
    String sql = "SELECT * FROM patient WHERE id = :id";
    return jdbcTemplate.queryForObject(sql, Map.of("id", id), new
    PatientRowMapper());
}
```

---

## Group Appointments by Doctor ID

### Approach

- Used `GROUP BY doctor_id` to count the number of appointments per doctor.
- Ensured efficient query execution with optimized indexing.

### Challenges & Solutions

*Challenge:* Aggregation queries can be slow with large datasets.

*Solution:* Used proper indexing on `doctor_id` to improve performance.

### Code Example: Counting Appointments per Doctor

```
java
```

## Feature Implemented

## Status

```
CopyEdit
@GetMapping("/appointments/grouped")
public List<Map<String, Object>> getAppointmentsGroupedByDoctor()
{
    String sql = "SELECT doctor_id, COUNT(*) as total_appointments
FROM appointment GROUP BY doctor_id";
    return jdbcTemplate.queryForList(sql, new HashMap<>());
}
```

---

## Pagination for Patients

### Approach

- Used SQL's LIMIT and OFFSET to fetch only the required records per request.
- Passed pagination parameters (page and size) dynamically to control data retrieval.

### Challenges & Solutions

*Challenge:* Without pagination, retrieving large datasets caused slow responses.

*Solution:* Implemented pagination to return only a few records at a time, reducing query load.

### Code Example: Fetching Patients with Pagination

```
java
CopyEdit
@GetMapping("/patients")
public List<Patient> getPatients(@RequestParam(defaultValue = "0")
int page,
                                @RequestParam(defaultValue = "5")
int size) {
    String sql = "SELECT * FROM patient LIMIT :size OFFSET
:offset";
    Map<String, Object> params = Map.of(
        "size", size,
        "offset", page * size
    );
    return jdbcTemplate.query(sql, params, new
PatientRowMapper());
}
```

---

### Approach

## Feature Implemented

## Status

- Why Named Parameters? It improves query readability and prevents SQL injection.
- Replaced "?" placeholders with named parameters for better maintainability.

## Challenges & Solutions

*Challenge:* Hardcoded SQL queries were difficult to manage.

*Solution:* Used NamedParameterJdbcTemplate for structured and parameterized queries.

### Code Example: Fetching a Doctor by ID

```
java
CopyEdit
public Doctor getDoctorById(int id) {
    String sql = "SELECT * FROM doctor WHERE id = :id";
    return jdbcTemplate.queryForObject(sql, Map.of("id", id), new
    DoctorRowMapper());
}
```

---

## Exception Handling

### Approach

- Implemented a Global Exception Handler to handle runtime errors.
- Used @RestControllerAdvice to return user-friendly error messages.

## Challenges & Solutions

*Challenge:* Unexpected errors caused application crashes.

*Solution:* Implemented global exception handling to return proper error responses.

### Code Example: Exception Handling

```
java
CopyEdit
@RestControllerAdvice
public class GlobalExceptionHandler {
    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleException(Exception e) {
        return
        ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body("Error: " + e.getMessage());
    }
}
```

**Feature Implemented****Status**

```
}  
}
```

## 7)Conclusion

This project successfully implemented a scalable, efficient, and secure Hospital Management API using Spring Boot, RESTful services, and JDBC. The system ensures fast data retrieval, optimized queries, and user-friendly error handling.

**Challenges Overcome:**

- SQL query performance issues → **Fixed with indexing and pagination**
- Risk of SQL injection → **Used NamedParameterJdbcTemplate**
- Unhandled exceptions → **Implemented Global Exception Handling**

**Future Enhancements:**

- Implement JWT authentication for security.
- Add filtering based on gender, age, or specialization.
- Improve logging for better debugging.