🎓 Case Study: Library Management System

🧾 Objective:

Design a Library Management System where:

- Readers can borrow books
- Books belong to categories
- Authors can write multiple books

📦 Entities:

1. Reader

- Each reader has a name and email.
- One reader can borrow many books.

2. Book

- Each book has a title and publish date.
- One book can be borrowed by one reader at a time.
- One book belongs to one category.
- One book is written by one author.

3. Category

- Each category has a name (e.g., Fiction, Technology).
- One category can have many books.

4. Author

- Each author has a name.
- One author can write multiple books.

🔄 Relationships Between Entities

Entity	Relationship	Description
Reader	One-to-Many with Book	A reader can borrow multiple books

Book	Many-to-One with Reader	A book can be borrowed by one reader
Book	Many-to-One with Category	A book belongs to one category
Book	Many-to-One with Author	A book is written by one author
Author	One-to-Many with Book	One author can write many books
Category	One-to-Many with Book	One category can contain many books

🗂️ Folder Structure

```
src/main/java
└── com.example.library
    ├── controller
    │   └── LibraryController.java
    ├── entity
    │   ├── Reader.java
    │   ├── Book.java
    │   ├── Author.java
    │   └── Category.java
    ├── repository
    │   ├── ReaderRepository.java
    │   ├── BookRepository.java
    │   ├── AuthorRepository.java
    │   └── CategoryRepository.java
    ├── LibraryManagementApplication.java
```

⚙ application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/library_db
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

🛠️ Step-by-Step Flow:

Step 1: Setup Project

- Use Spring Initializr
- Dependencies: Spring Web, Spring Data JPA, MySQL Driver, Lombok

Step 2: Create MySQL Database

```
CREATE DATABASE library_db;
```

🧪 Sample API Use (Postman Testing)

➕ Add a Category

POST `http://localhost:8080/api/categories`

```
{
  "name": "Fiction"
}
```

➕ Add an Author

POST `http://localhost:8080/api/authors`

```
{
  "name": "George Orwell"
}
```

➕ Add a Reader

POST `http://localhost:8080/api/readers`

```
{
  "name": "Alice",
  "email": "alice@gmail.com"
}
```

➕ Add a Book

POST `http://localhost:8080/api/books`

```
{
  "title": "1984",
  "publishDate": "1949-06-08",
  "reader": { "id": 1 },
  "category": { "id": 1 },
  "author": { "id": 1 }
}
```

🔄 Actions (CRUD)

Entity	POST (Add)	GET (View All)	PUT (Update)	DELETE (Remove)

Reader	/api/readers	/api/readers	/api/readers/{id}	/api/readers/{id}
Book	/api/books	/api/books	/api/books/{id}	/api/books/{id}
Author	/api/authors	/api/authors	/api/authors/{id}	/api/authors/{id}
Category	/api/categories	/api/categories	/api/categories/{id}	/api/categories/{id}

✅ Case Study Title: Hospital Management System using Spring Boot and Spring Data JPA

🏥 1. Overview

The Hospital Management System helps manage patients, doctors, appointments, and medical records. It allows hospital staff to:

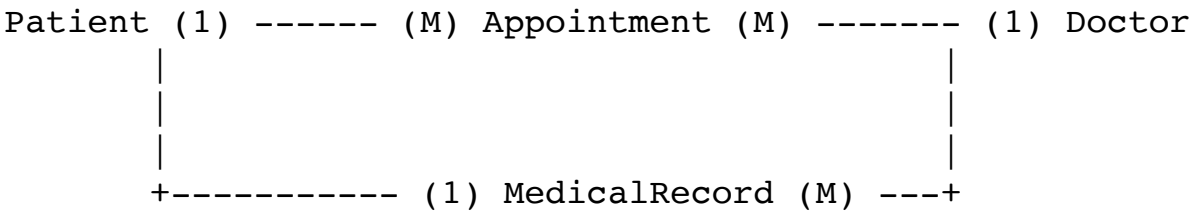
- Add/update patient and doctor records
- Schedule appointments
- Track medical history

📦 2. Final Entity Classes

We will use **4 primary entities** with proper **JPA relationships**:

Entity	Description	Key Relationships
`Patient`	Stores patient details	One-to-Many with `Appointment`, One-to-Many with `MedicalRecord`
`Doctor`	Stores doctor details	One-to-Many with `Appointment`
`Appointm`	Connects patients and doctors	Many-to-One with `Patient`, Many-to-One with
`MedicalRecord`	Tracks the patient's diagnosis & prescriptions	Many-to-One with `Patient`

🧱 3. Entity Relationship Diagram (ERD)



📦 4. JPA Entity Class Summary

1. `Patient`

- `id`, `name`, `age`, `gender`, `address`
- **Mapped** to `Appointment` and `MedicalRecord` (One-to-Many)

2. `Doctor`

- `id`, `name`, `specialization`, `email`, `phone`
- **Mapped** to `Appointment` (One-to-Many)

3. `Appointment`

- `id`, `date`, `time`, `notes`
- References both `Patient` and `Doctor` (Many-to-One)

4. `MedicalRecord`

- `id`, `diagnosis`, `treatment`, `date`
- Linked to a `Patient` (Many-to-One)

🔌 5. Spring Boot and JDBC Connectivity

Spring Boot provides automatic JPA configuration for JDBC. You'll just need:

```
spring.datasource.url=jdbc:mysql://localhost:3306/hospitaldb
spring.datasource.username=root
spring.datasource.password=yourpassword
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
```

Use **Spring Data JPA Repositories** for each entity:

```
public interface PatientRepository extends
JpaRepository<Patient, Long> {}
public interface DoctorRepository extends
JpaRepository<Doctor, Long> {}
public interface AppointmentRepository extends
JpaRepository<Appointment, Long> {}
```

```
public interface MedicalRecordRepository extends
JpaRepository<MedicalRecord, Long> {}
```

📬 6. API Flow (Sample)

Action	HTTP Method	Endpoint
Add Patient	POST	`/api/patients`
List all Patients	GET	`/api/patients`
Add Doctor	POST	`/api/doctors`
Book Appointment	POST	`/api/appointments`
View Appointments	GET	`/api/appointments`
Add Medical Record	POST	`/api/medical-records`
View Patient History	GET	`/api/patients/{id}/records`

🧪 7. Sample Postman Data (JSON)

Create Patient:

```
{
  "name": "John Doe",
  "age": 35,
  "gender": "Male",
  "address": "123 Main Street"
}
```

Create Doctor:

```
{
  "name": "Dr. Smith",
  "specialization": "Cardiologist",
  "email": "drsmith@example.com",
  "phone": "9876543210"
}
```