# Spring Data JPA: A Quick Example

## Introduction

This document provides a complete, runnable example of using **Spring Data JPA** to handle database operations in a Spring Boot application. The example creates a simple library management system that can save and retrieve Book objects from an **H2 in-memory database**.

The key benefit of Spring Data JPA is that it automatically generates the data access layer (repository) code at runtime, significantly reducing boilerplate code.

## Step 1: Configure Project Dependencies (pom.xml)

First, ensure your Maven project's pom.xml includes the necessary dependencies for Spring Web, Spring Data JPA, and the H2 Database.

**File:** pom.xml

xml

<dependencies> *<!-- For building web applications and REST APIs -->* <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-web</artifactId> </dependency> *<!-- The main dependency for Spring Data JPA -->* <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-data-jpa</artifactId> </dependency> *<!-- H2 In-Memory Database Driver -->* <dependency> <groupId>com.h2database</groupId> <artifactId>h2</artifactId> <scope>runtime</scope> </dependency></dependencies>

## Step 2: Define the Data Model (The Entity)

An entity is a Plain Old Java Object (POJO) that is mapped to a table in the database. The @Entity annotation marks the class as a JPA entity.

**File:** src/main/java/com/library/model/Book.java

java

**package** com.library.model;**import** jakarta.persistence.Entity;**import** jakarta.persistence.GeneratedValue;**import** jakarta.persistence.Id;@Entity *// Marks this class as a database table***public** **class** Book { @Id *// Marks this field as the primary key* @GeneratedValue *// Configures the ID to be auto-generated by the database* **private** Long id; **private** String title; **private** String author; *// A no-argument constructor is required by the JPA specification* **protected** Book() {} **public** Book(String title, String author) { **this**.title = title; **this**.author = author; } *// Standard getters for all fields* **public** Long getId() { **return** id; } **public** String getTitle() { **return** title; } **public** String getAuthor() { **return** author; } @Override **public** String toString() { **return** String.format("Book[id=%d, title='%s', author='%s']", id, title, author); }}

## Step 3: Create the Repository Interface

This is the core of Spring Data JPA. You only need to define an interface that extends JpaRepository. Spring automatically provides a full implementation with standard CRUD (Create, Read, Update, Delete) methods. You can also define custom query methods just by declaring their signature.

**File:** src/main/java/com/library/repository/BookRepository.java

java

**package** com.library.repository;**import** com.library.model.Book;**import** org.springframework.data.jpa.repository.JpaRepository;**import** java.util.List;*// The generic types are the entity class (Book) and its primary key type (Long).***public** **interface** BookRepository **extends** JpaRepository<Book, Long> { */\*\* \* Spring Data JPA creates this query automatically from the method name. \* It understands "FindByAuthor" and generates a query to find all books \* where the 'author' field matches the provided string. \*/* List<Book> findByAuthor(String author);}

## Step 4: Configure the Database Connection

Configure the H2 in-memory database connection in the application.properties file.

**File:** src/main/resources/application.properties

text

# H2 Database Settingsspring.datasource.url=jdbc:h2:mem:librarydbspring.datasource.driverClassName=org.h2.Driverspring.datasource.username=saspring.datasource.password=# Automatically create the database schema on startup and drop it on shutdown.# This is great for development and testing.spring.jpa.hibernate.ddl-auto=create-drop# Enable the H2 console, a web interface to view the database.spring.h2.console.enabled=true

## Step 5: Use the Repository in the Main Application

To demonstrate the repository, we use a CommandLineRunner. This special bean executes its code once the Spring application has started. We inject the BookRepository and use it to perform database operations.

**File:** src/main/java/com/library/LibraryManagementApplication.java

java

**package** com.library;**import** com.library.model.Book;**import** com.library.repository.BookRepository;**import** org.springframework.boot.CommandLineRunner;**import** org.springframework.boot.SpringApplication;**import** org.springframework.boot.autoconfigure.SpringBootApplication;**import** org.springframework.context.annotation.Bean;@SpringBootApplication**public** **class** LibraryManagementApplication { **public** **static** **void** main(String[] args) { SpringApplication.run(LibraryManagementApplication.**class**, args); } *// This bean will be executed on application startup.* *// Spring automatically injects our BookRepository implementation.* @Bean **public** CommandLineRunner demo(BookRepository repository) { **return** (args) -> { *// 1. Save a few books to the database* System.out.println("--- Saving new books ---"); repository.save(**new** Book("A Game of Thrones", "George R. R. Martin")); repository.save(**new** Book("The Name of the Wind", "Patrick Rothfuss")); repository.save(**new** Book("The Way of Kings", "Brandon Sanderson")); repository.save(**new** Book("Mistborn", "Brandon Sanderson")); System.out.println("------------------------\n"); *// 2. Fetch all books* System.out.println("--- All books found with findAll(): ---"); repository.findAll().forEach(System.out::println); System.out.println("--------------------------------------\n"); *// 3. Fetch an individual book by its ID* System.out.println("--- Book found with findById(1L): ---"); repository.findById(1L).ifPresent(System.out::println); System.out.println("------------------------------------\n"); *// 4. Fetch books using our custom findByAuthor method* System.out.println("--- Books by 'Brandon Sanderson' found with findByAuthor(): ---"); repository.findByAuthor("Brandon Sanderson").forEach(System.out::println); System.out.println("------------------------------------------------------------\n"); }; }}

## Step 6: Expected Output

When you run the LibraryManagementApplication, you will see the following output in the console, confirming that all database operations were successful.

text

--- Saving new books ------------------------------ All books found with findAll(): ---Book[id=1, title='A Game of Thrones', author='George R. R. Martin']Book[id=2, title='The Name of the Wind', author='Patrick Rothfuss']Book[id=3, title='The Way of Kings', author='Brandon Sanderson']Book[id=4, title='Mistborn', author='Brandon Sanderson']----------------------------------------- Book found with findById(1L): ---Book[id=1, title='A Game of Thrones', author='George R. R. Martin']--------------------------------------- Books by 'Brandon Sanderson' found with findByAuthor(): ---Book[id=3, title='The Way of Kings', author='Brandon Sanderson']Book[id=4, title='Mistborn', author='Brandon Sanderson']------------------------------------------------------------