

# Configuring Hibernate in Spring Boot



Estimated time: 45 minutes

## Overview

In this lab, you will learn:

- **What ORM (Object-Relational Mapping)** is and how it simplifies database interactions.
- **Why Hibernate is a leading ORM framework** for Java applications.
- **Key components of Hibernate architecture** and how they interact.
- **How to use Hibernate annotations** for mapping Java objects to database tables.
- **Real-world use cases** where Hibernate is beneficial.

By the end of this lab, you will be able to configure Hibernate and use it for database operations efficiently.

## Learning Objectives

After completing this lab, you will be able to:

- Understand **Hibernate's architecture** and core components.
- Use **Hibernate annotations** to map Java classes to database tables.
- Perform **CRUD operations** using Hibernate.
- Manage **transactions and entity lifecycle** in Hibernate.

## Understanding ORM

### What is ORM?

- **Object-Relational Mapping (ORM)** is a technique that maps object-oriented programming models to relational database tables.
- ORM allows developers to interact with databases using Java objects rather than writing raw SQL queries.
- **Advantages of ORM:**
  - Reduces boilerplate SQL code.
  - Simplifies data persistence and retrieval.
  - Automates object-to-table mapping.

### Challenges with JDBC (Before ORM)

- Manual SQL queries were required for every database operation.
- Complex result set mapping to Java objects.
- Error-prone relationship management between database tables.

## Introduction to Hibernate

### What is Hibernate?

- Hibernate is an **open-source ORM framework** for Java.
- It provides tools to persist and retrieve Java objects from relational databases.
- Key benefits:
  - Automates SQL query generation.
  - Supports lazy and eager loading.
  - Handles **One-to-Many, Many-to-Many relationships**.
  - Supports **caching** for better performance.

### Key Components of Hibernate

Component	Description
SessionFactory	Creates Hibernate sessions and manages configurations.
Session	Handles database operations for entities.
Transaction	Ensures atomicity of database changes.
Query	Executes HQL (Hibernate Query Language) or SQL queries.
Entity Classes	POJOs (Plain Old Java Objects) mapped to database tables.
Configuration	Loads Hibernate settings from <code>hibernate.cfg.xml</code> .

## Getting Started

## Step 1: Create the initial code

### Initial settings

Go to <https://start.spring.io/index.html>

- In **Project**, select **Maven**.
- In **Language**, select **Java**.
- In **Group**, enter `com.project`.
- In **Artifact**, enter `code`.

Please just leave the other settings as the default.

### Add dependencies

Click **Add dependencies** on the right and add the following dependencies:

- Spring Web
- MySQL Driver

### Download code

Click **Generate** to download the `code.zip` file.

## Step 2: Move the file to the working directory in VS Code

1. Click the explorer icon in the left-hand panel. Drag the `code.zip` file under the `Projects` directory.

File Edit Selection



EXPLORER

> OPEN EDITORS

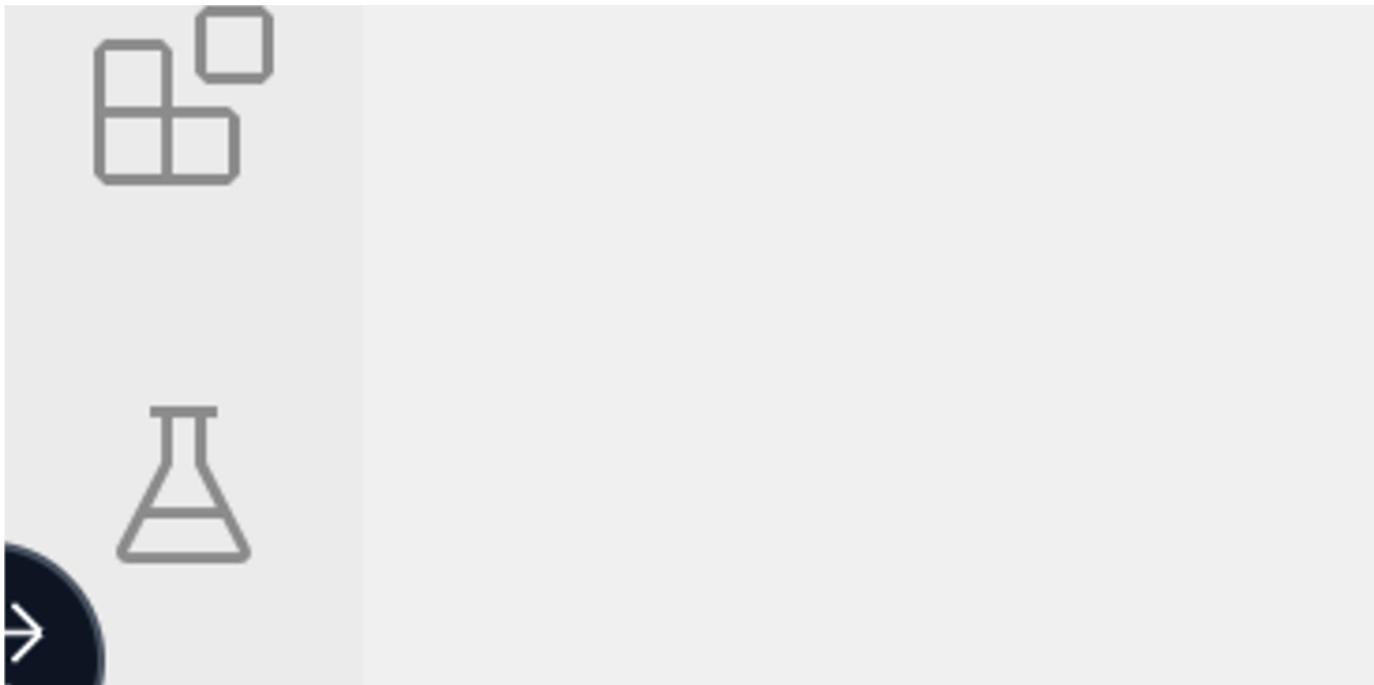


▼ PROJECT

▼ □ .theia

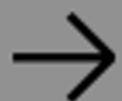
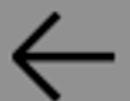
■ settings.json





2. The `code.zip` file appears in the Projects directory. A message box asks if you want to open the file. Click the `No` button.

File Edit Selection



EXPLORER

...

> OPEN EDITORS

▼ PROJECT

▼ .theia



settings.json



code.zip



The file is either binary

Opening it might take s  
anyway?



3. In the terminal window, unzip the file by running the following command:

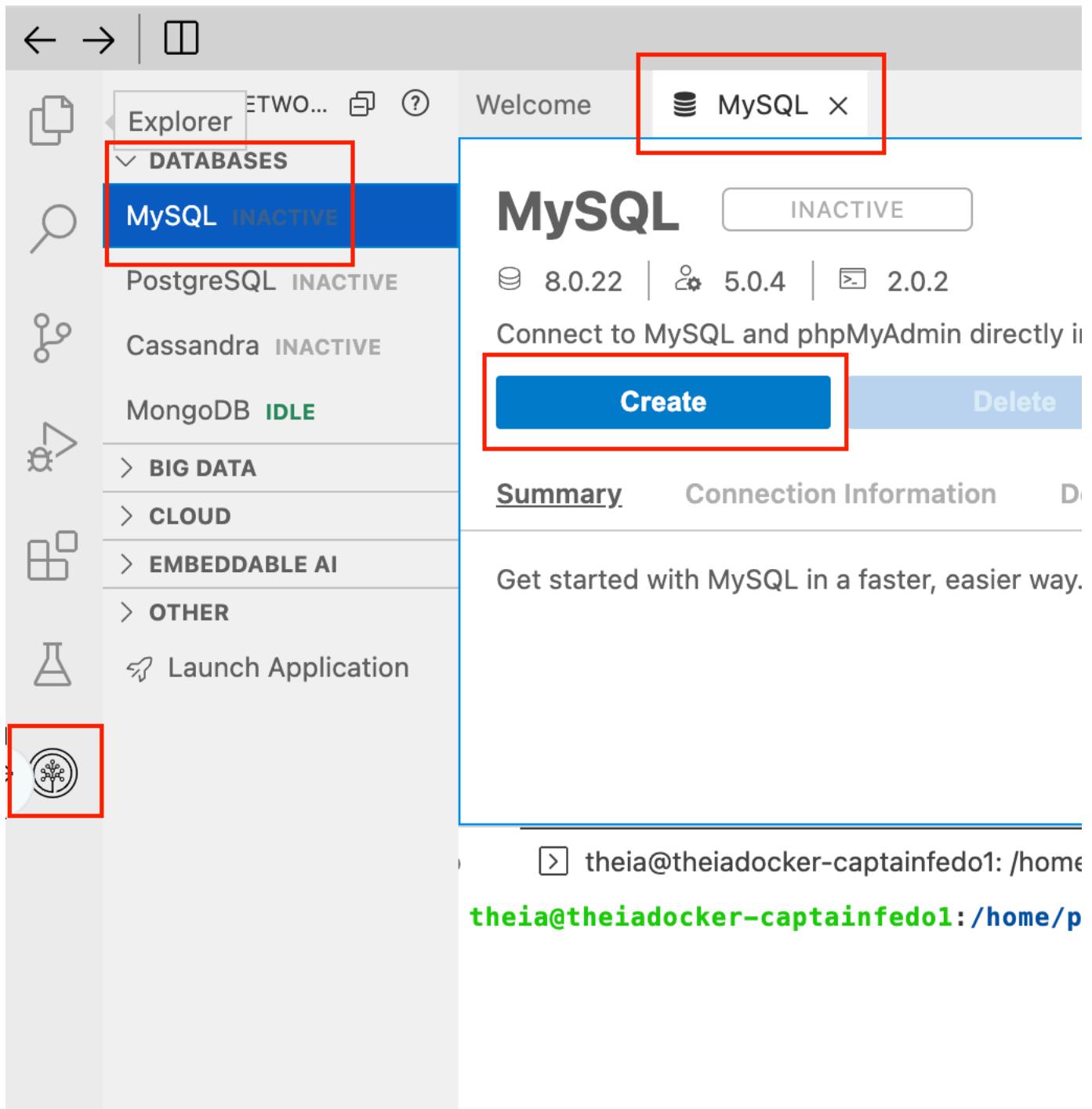
```
unzip code.zip
```

4. Remove the zip file using the following command:

```
rm -r code.zip
```

### Step 3: Create a MySQL instance

1. The first step is to create the MySQL instance in the IDE environment. Click the toolbox icon at the bottom of the left-hand panel. This will bring up another panel on the left. Select MySQL in the databases category. This will open a new tab. Click the Create button.



2. The MySQL Database will start in a few minutes. Once it has started, click the `MySQL CLI` button in the `Summary` tab. This will open the MySQL terminal at the bottom of the screen.

Welcome MySQL X

# MySQL

ACTIVE

8.0.22 | 5.0.4 | 2.0.2

Connect to MySQL and phpMyAdmin directly in your Skills Network Labs environment.

[Create](#) [Delete](#)

[Summary](#) [Connection Information](#) [Details](#)

Your database and phpMyAdmin server are now ready to use and available. Learn how to navigate MySQL, please check out the Details section.

You can manage MySQL via:

[phpMyAdmin](#) [Open in new tab](#)

Or to interact with the database in the terminal, select one of these options:

[MySQL CLI](#) [New Terminal](#)

```
theia@theiadocker-captainfedo1:/home/project$ mysql --host=127.0.0.1 -u root -p
d=c4hXuJ5KGkhLMSPvGB9P9hE
mysql: [Warning] Using a password on the command line interface may be insecure!
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1181
Server version: 8.0.37 MySQL Community Server - GPL

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

3. You can now enter all the **MySQL-based** commands in this terminal. This example shows the `show databases;` command to list all the databases.

# MySQL

ACTIVE

8.0.22 | 5.0.4 | 2.0.2

Connect to MySQL and phpMyAdmin directly in your Skills Network Labs environment.

Create

Delete

Summary

Connection Information

Details

MYSQL\_USERNAME:

root



MYSQL\_HOST:

172.21.226.108



theia@theiadocker-captainfedo1: /home/project

theia@theiadocke

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.00 sec)
```

```
mysql> 
```

4. You can find the username, password, and connection URL information in the Connection Information tab if you need to connect to this database outside the IDE environment or if the information is required in a script or file.

The screenshot shows the MySQL configuration page in the Skills Network interface. At the top, there are navigation icons for back, forward, and search. The main header includes 'Welcome' and 'MySQL' with an 'ACTIVE' status indicator. Below the header, version numbers are listed: MySQL 8.0.22, PHP 5.0.4, and PHPMyAdmin 2.0.2. A descriptive text says 'Connect to MySQL and phpMyAdmin directly in your Skills Network Labs environment'. There are 'Create' and 'Delete' buttons. The 'Connection Information' tab is selected and highlighted with a red box. It displays the following details:

- MYSQL\_USERNAME:** root
- MYSQL\_HOST:** 172.21.226.108
- MYSQL\_PORT:** 3306
- URL:** https://labs-mysql-short-red-napkin.mysql.data.tls

#### Step 4: Create a bookstore database

1. In the MySQL CLI terminal, create a database called bookstore in MySQL using the following command:

```
create database bookstore;
```

2. The output will look similar to the following:

```
mysql> create database bookstore;
Query OK, 1 row affected (0.01 sec)
```

# Setting Up Hibernate

## Add Dependencies in pom.xml

[Open pom.xml in IDE](#)

Before Hibernate can work in our project, we need to include it as a dependency. Hibernate will manage object-relational mapping and facilitate database operations through configuration and annotated classes.

1. Add the following code to the file at the end of the last dependency within the dependencies section:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.2.0.Final</version>
</dependency>
```

## Configure Hibernate in hibernate.cfg.xml

1. Click the button below to create a hibernate.cfg.xml file in /src/main/resources/.

[Open hibernate.cfg.xml in IDE](#)

This configuration file defines how Hibernate connects to the database, what dialect it uses, and how it handles session management and schema updates.

2. Add the following code to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name = "hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property name = "hibernate.connection.driver_class">
            com.mysql.cj.jdbc.Driver
        </property>
        <property name = "hibernate.connection.url">
            jdbc:mysql://{MYSQL_HOST}/bookstore
        </property>
        <property name = "hibernate.connection.username">
            {MYSQL_USERNAME}
        </property>
        <!-- Enter your correct password here -->
        <property name = "hibernate.connection.password">
            {MYSQL_PASSWORD}
        </property>
        <!-- Echo all executed SQL to stdout -->
        <property name = "show_sql">
            true
        </property>
        <!-- Enable Hibernate's automatic session context management -->
        <property name = "current_session_context_class">thread</property>
        <!-- Enable Hibernate to update/create table in mysql -->
        <property name="hibernate.hbm2ddl.auto">update</property>
    </session-factory>
</hibernate-configuration>
```

## Explanation:

- **hibernate.dialect** – Tells Hibernate how to generate SQL for a specific database.
- **hibernate.connection.driver\_class** – JDBC driver class used for MySQL.

- `hibernate.connection.url` – JDBC URL to connect to the MySQL bookstore database.
- `hibernate.connection.username/password` – Credentials to connect to the database.
- `show_sql` – Enables logging of SQL queries in the console.
- `current_session_context_class` – Ensures sessions are managed per thread.
- `hibernate.hbm2ddl.auto=update` – Automatically updates the database schema.

3. To establish the connection, ensure you replace the `{MYSQL_HOST}`, `{MYSQL_USERNAME}`, and `{MYSQL_PASSWORD}` parts with the actual database details. You will find all these details in the MySQL Connection Information tab.

## Creating an Entity Class

### Create Book.java Entity Class in /src/main/java/com/project/code

In this section, we'll define our entity class which will be mapped to a table in the database. Hibernate will use this class to perform persistence operations on the corresponding table.

1. Click the button below to create a `Book.java` file in `src/main/java/com/project/code/`:

[Open Book.java in IDE](#)

Alternatively, you can also use the terminal.

```
touch /home/project/code/src/main/java/com/project/code/Book.java
```

2. Add the following code to the file:

```
package com.project.code;
import jakarta.persistence.*;
@Entity
@Table(name = "books")
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(name = "title", nullable = false, length = 100)
    private String title;
    @Column(name = "price", nullable = false)
    private double price;
    // Getters and setters
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}
```

### Explanation:

- `@Entity` – Marks this class as a Hibernate entity.
- `@Table(name = "books")` – Maps the entity to a database table.
- `@Id` and `@GeneratedValue` – Define the primary key.
- `@Column` – Maps fields to database columns.

## Managing Hibernate Sessions

### Create HibernateUtil.java in /src/main/java/com/project/code

This utility class centralizes the setup of Hibernate's SessionFactory and provides methods to access and shut it down when needed.

1. Click the button below to create a `HibernateUtil.java` file in `src/main/java/com/project/code/`:

[Open `HibernateUtil.java` in IDE](#)

Alternatively, you can also use the terminal.

```
touch /home/project/code/src/main/java/com/project/code/HibernateUtil.java
```

2. Add the following code to the file:

```
package com.project.code;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;
public class HibernateUtil {
    private static final SessionFactory sessionFactory = new Configuration().configure("hibernate.cfg.xml").addAnnotatedClass(Book.class).buildSessionFactory();
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

## Explanation:

- Loads `hibernate.cfg.xml` and initializes `SessionFactory`.
- `getSession()` method opens a session for database interactions.

## Creating Services

### Create `BookService.java` in `/src/main/java/com/project/code`

1. Click the button below to create a `BookService.java` file in `src/main/java/com/project/code/`:

[Open `BookService.java` in IDE](#)

Alternatively, you can also use the terminal.

```
touch /home/project/code/src/main/java/com/project/code/BookService.java
```

This class contains the business logic to handle CRUD operations using Hibernate's Session and Transaction. You'll use this service in your controller to interact with the database.

2. Add the following code to the file:

```
package com.project.code;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.springframework.stereotype.Service;
import java.util.List;
@Service
public class BookService {
    // Save book
    public void saveBook(Book book) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction transaction = session.beginTransaction();
        session.save(book);
        transaction.commit();
    }
}
```

```

    }
    // Get book by id
    public Book getBookById(Long id) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction transaction = session.beginTransaction();
        Book book = session.get(Book.class, id);
        transaction.commit();
        return book;
    }
    // Get all books
    public List<Book> getAllBooks() {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction transaction = session.beginTransaction();
        List<Book> books = session.createQuery("FROM Book", Book.class).getResultList();
        transaction.commit();
        return books;
    }
    // Update book
    public void updateBook(Book book) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction transaction = session.beginTransaction();
        session.update(book);
        transaction.commit();
    }
    // Delete book
    public void deleteBook(Long id) {
        Session session = HibernateUtil.getSessionFactory().getCurrentSession();
        Transaction transaction = session.beginTransaction();
        Book book = session.get(Book.class, id);
        if (book != null) {
            session.delete(book);
        }
        transaction.commit();
    }
}

```

Explanation:

- **@Service** – Marks the class as a Spring service component.
- **saveBook(Book book)** – Opens a Hibernate session and saves the book entity.
- **getBookById(Long id)** – Retrieves a book by its primary key.
- **getAllBooks()** – Queries all records from the books table.
- **updateBook(Book book)** – Updates an existing book.
- **deleteBook(Long id)** – Deletes the book by ID if it exists.
- **Transaction Management** – Each method begins and commits a transaction, ensuring changes are consistent and atomic.

## Creating Controllers

Create BookController.java in /src/main/java/com/project/code

1. Click the button below to create a BookController.java file in src/main/java/com/project/code/:

[Open BookController.java in IDE](#)

Alternatively, you can also use the terminal.

```
touch /home/project/code/src/main/java/com/project/code/BookController.java
```

This controller class handles HTTP requests to create, retrieve, update, and delete books. It connects the frontend/API endpoints to the BookService layer.

2. Add the following code to the file:

```

package com.project.code;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
public class BookController {
    @Autowired
    private BookService bookService;
    // Create a new book
    @PostMapping("/saveBook")
    public String saveBook(@RequestBody Book book) {
        bookService.saveBook(book);
    }
}

```

```

        return "Book saved with ID: " + book.getId();
    }
    // Get all books
    @GetMapping("/getBooks")
    public List<Book> getAllBooks() {
        return bookService.getAllBooks();
    }
    // Get book by ID
    @GetMapping("/getBook/{id}")
    public Book getBookById(@PathVariable Long id) {
        return bookService.getBookById(id);
    }
    // Update a book by ID
    @PutMapping("/updateBook/{id}")
    public String updateBook(@PathVariable Long id, @RequestBody Book book) {
        book.setId(id);
        bookService.updateBook(book);
        return "Book updated with ID: " + id;
    }
    // Delete a book by ID
    @DeleteMapping("/deleteBook/{id}")
    public String deleteBook(@PathVariable Long id) {
        bookService.deleteBook(id);
        return "Book deleted with ID: " + id;
    }
}

```

## Explanation:

- `@RestController` – Indicates this is a controller where every method returns a response body.
- `@Autowired` – Automatically injects the BookService bean.
- `@PostMapping("/saveBook")` – Maps HTTP POST requests to save a new book.
- `@GetMapping("/getBooks")` – Maps HTTP GET requests to fetch all books.
- `@GetMapping("/getBook/{id}")` – Maps to fetch a specific book using its ID.
- `@PutMapping("/updateBook/{id}")` – Updates an existing book by ID.
- `@DeleteMapping("/deleteBook/{id}")` – Deletes a book from the database by ID.

## Application and Test Endpoints

### Start the Spring Boot application

1. Open a new terminal by using the top menu Terminal -> New Terminal and run the command below to start the Spring Boot application.

```
cd /home/project/code && mvn clean install && mvn spring-boot:run
```

The command does four things:

- Changes into the project directory
- Runs the Maven clean command
- Installs all the dependencies
- Runs the application

```

...
...
2025-03-25T01:35:19.167-04:00  INFO 3485 --- [code] [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: ini
2025-03-25T01:35:19.508-04:00  INFO 3485 --- [code] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port 8080 (ht
2025-03-25T01:35:19.515-04:00  INFO 3485 --- [code] [           main] com.project.code.CodeApplication      : Started CodeApplication in 1.77

```

## Updating the database

1. Open a new terminal by using the menu Terminal -> New Terminal.

## Add Book

2. Now, hit the api endpoint /addBook using the command below:

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "Spring Boot Essentials",
    "author": "John Doe",
    "price": 19.99
}'
```

You should see the output Book saved.

```
Book saved with ID: 1
```

3. Let's add a few more books, by running each of the commands below in sequence.

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "Java Basics",
    "author": "Alice Johnson",
    "price": 15.50
}'
```

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "Hibernate in Depth",
    "author": "Michael Brown",
    "price": 22.00
}'
```

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "REST APIs with Spring",
    "author": "Samantha Lee",
    "price": 18.25
}'
```

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "Effective Java",
}'
```

```
"author": "Joshua Bloch",
"price": 28.99
}'
```

```
curl --location 'http://localhost:8080/saveBook' \
--header 'Content-Type: application/json' \
--data '{
    "title": "Clean Code",
    "author": "Robert C. Martin",
    "price": 25.75
}'
```

## List All books

4. To list the books, run the following command:

```
curl --location 'http://localhost:8080/getBooks'
```

You'll get output similar to the following:

```
[{"id":1,"title":"Spring Boot Essentials","price":19.99}, {"id":2,"title":"Java Basics","price":15.5}, {"id":3,"title":"Hibernate in Depth","price":25.75}]
```

You can use the util jq to get a more formatted output:

5. Run this command:

```
curl --location 'http://localhost:8080/getBooks' | jq
```

You'll see this output:

```
[
  {
    "id": 1,
    "title": "Spring Boot Essentials",
    "price": 19.99
  },
  {
    "id": 2,
    "title": "Java Basics",
    "price": 15.5
  },
  {
    "id": 3,
```

```
        "title": "Hibernate in Depth",
        "price": 22
    },
    {
        "id": 4,
        "title": "REST APIs with Spring",
        "price": 18.25
    },
    {
        "id": 5,
        "title": "Effective Java",
        "price": 28.99
    },
    {
        "id": 6,
        "title": "Clean Code",
        "price": 25.75
    }
]
```

## Get Book by id

- Run the curl command below to get book data by Id:

```
curl --location 'http://localhost:8080/getBook/1'
```

**Note:** You can change the id to any id but it should be present in the database. Otherwise it will not return any output.

```
{"id":1,"title":"Spring Boot Essentials","price":19.99}
```

## Update Book by id

- Run the curl command below to update the book data:

```
curl --location --request PUT 'http://localhost:8080/updateBook/1' \
--header 'Content-Type: application/json' \
--data '{"title":"SpringBoot","author": "John", "price":"10"}'
```

You'll see the following output:

```
Book updated with ID: 1
```

## Delete book by id

8. Run the below command to delete a record in the table using its Id:

```
curl --location --request DELETE 'http://localhost:8080/deleteBook/4'
```

You'll see the following output:

```
Book deleted with ID: 4
```

9. You can view all the books again to see the changes with the jq util.

```
curl --location 'http://localhost:8080/getBooks' | jq
```

```
[  
  {  
    "id": 1,  
    "title": "SpringBoot",  
    "price": 10  
  },  
  {  
    "id": 2,  
    "title": "Java Basics",  
    "price": 15.5  
  },  
  {  
    "id": 3,  
    "title": "Hibernate in Depth",  
    "price": 22  
  },  
  {  
    "id": 5,  
    "title": "Effective Java",  
    "price": 28.99  
  },  
  {  
    "id": 6,  
    "title": "Clean Code",  
    "price": 25.75  
  }]
```

## Validating the Database

Now, let's check if the changes executed are reflected in the database.

1. Switch back to the MySQL CLI terminal, and run the command below:

```
use bookstore;
```

2. Now run the following command:

```
select * from books;
```

It will show the data added in the table.

```
mysql> select * from books;
+----+-----+-----+
| id | price | title |
+----+-----+-----+
| 1  | 10   | SpringBoot |
| 2  | 15.5 | Java Basics |
| 3  | 22   | Hibernate in Depth |
| 5  | 28.99 | Effective Java |
| 6  | 25.75 | Clean Code |
+----+-----+-----+
5 rows in set (0.01 sec)
```

### Conclusion:

- As you can see, we updated the book with id **1** and we can see the changes in the database as well.
- We also deleted the book with id **4** and we can see after the **3** the next number in the ID column is **5**.

Additionally, you can also see the changes in the terminal as well where the Spring Boot app was running.

```
2025-03-12T21:59:33.316+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.pooling : HHH10001005: Loaded JDBC driver
2025-03-12T21:59:33.317+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.pooling : HHH10001012: Connecting with J
2025-03-12T21:59:33.317+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.pooling : HHH10001001: Connection proper
2025-03-12T21:59:33.317+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.pooling : HHH10001003: Autocommit mode:
2025-03-12T21:59:33.317+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.pooling : HHH10001115: Connection pool s
2025-03-12T21:59:33.472+05:30 INFO 21013 --- [code] [nio-8080-exec-1] SQL dialect : HHH000400: Using dialect: org.
2025-03-12T21:59:33.531+05:30 INFO 21013 --- [code] [nio-8080-exec-1] o.h.b.i.BytecodeProviderInitiator : HHH000021: Bytecode provider n
2025-03-12T21:59:33.764+05:30 INFO 21013 --- [code] [nio-8080-exec-1] org.hibernate.orm.connections.access : HHH10001501: Connection obtain
Hibernate: create table books (id bigint not null auto_increment, price float(53) not null, title varchar(100) not null, primary key (id)) engine
Hibernate: insert into books (price,title) values (?,?)
Hibernate: select b1_0.id,b1_0.price,b1_0.title from books b1_0
Hibernate: select b1_0.id,b1_0.price,b1_0.title from books b1_0
Hibernate: select b1_0.id,b1_0.price,b1_0.title from books b1_0 where b1_0.id=?
Hibernate: update books set price=?,title=? where id=?
Hibernate: select b1_0.id,b1_0.price,b1_0.title from books b1_0 where b1_0.id=?
Hibernate: delete from books where id=?
```

## Conclusion and Next Steps

Congratulations! You have successfully learned how to use Hibernate for ORM in Java applications.

### Next Steps

- Investigate Hibernate Query Language (HQL) and Criteria API for advanced querying.
- Integrate Hibernate with Spring Data JPA for further simplification of data access.
- Learn about transaction management in more depth.

**Author(s)**

Upkar Lidder