

# JPA and Hibernate One To Many Bidirectional Mapping in Spring Boot REST APIs

Last modified @ 28 November 2020

# JPA and Hibernate

# Spring Boot

JPA and Hibernate provide `@ManyToOne` and `@OneToMany` as the two primary annotations for mapping One to Many unidirectional and bidirectional relationship

A bidirectional relationship provides navigation access to both sides while a unidirectional relationship provides navigation access to one side only

This tutorial will walk you through the steps of using `@OneToMany` and `@ManyToOne` to do a bidirectional mapping for a JPA and Hibernate One to Many relationship, and writing CRUD REST APIs to expose the relationship for accessing the database in Spring Boot, Spring Data JPA, and MySQL

There are a convenient benefit and also a performance tradeoff when using `@OneToMany`. We will address them and find the approach

Let's get started!

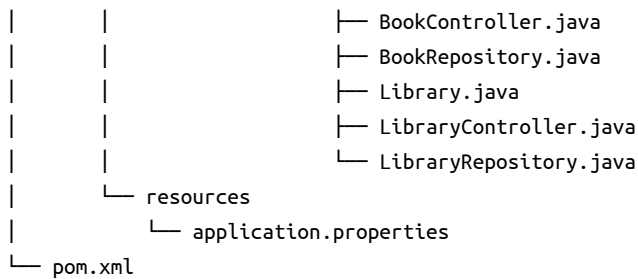
## What you'll need

- JDK 8+ or OpenJDK 8+
- Maven 3+
- MySQL Server 5+
- Your favorite IDE

## Project structure

Following is the final project structure with all the files we would create

```
├─ src
│   └─ main
│       └─ java
│           └─ com
│               └─ hellokoding
│                   └─ jpa
│                       └─ bidirectional
│                           └─ Application.java
│                           └─ Book.java
```



## Create a sample Spring Boot application

Create a new Spring Boot application with Spring Initializr via web UI or a command-line tool such as cURL or HTTPie, you can find the guide [at here](#)

### Example with the cURL command-line

```
curl https://start.spring.io/starter.zip \
  -d dependencies=jpa,mysql,web \
  -d javaVersion=1.8 \
  -d packageName=com.hellokoding.jpa \
  -d groupId=com.hellokoding.jpa \
  -o hk-demo-jpa.zip
```

Unzip the `hk-demo-jpa.zip` file and import the sample project into your IDE.

## Project dependencies

We will need the following dependencies on the pom.xml file

- `spring-boot-starter-data-jpa` to work with JPA and Hibernate
- `mysql-connector-java` to work with MySQL. The scope `runtime` indicates that the dependency is not required for compilation, but for execution
- `spring-boot-starter-web` for defining the CRUD REST APIs for the one-to-many relationship mapping

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>

<dependency>
```

```

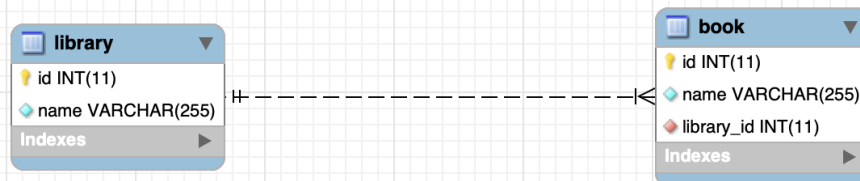
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>

```

## The One to Many Relationship

One-to-many refers to the relationship between two tables A and B in which one row of A may be linked with many rows of B, but one row of B is linked to only one row of A

We will use the relationship between the library and books to implement for this example. One library may have many books, one book can only be managed by one library. The relationship is enforced via the `library_id` foreign key column placed on the `book` table (the Many side)



## Define JPA and Hibernate Entities

Create `Library` and `Book` JPA Entities corresponding to `library` and `book` tables in the database.

```

package com.hellokoding.jpa.bidirectional;

import javax.persistence.*;
import javax.validation.constraints.NotNull;
import java.util.HashSet;
import java.util.Set;

@Entity
public class Library {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @NotNull
    private String name;

    @OneToMany(mappedBy = "library", cascade = CascadeType.ALL)
    private Set<Book> books = new HashSet<>();

    public int getId() {
        return id;
    }

```

```

    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Set<Book> getBooks() {
        return books;
    }

    public void setBooks(Set<Book> books) {
        this.books = books;

        for(Book b : books) {
            b.setLibrary(this);
        }
    }
}

package com.hellokoding.jpa.bidirectional;

import com.fasterxml.jackson.annotation.JsonProperty;

import javax.persistence.*;
import javax.validation.constraints.NotNull;

@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @NotNull
    private String name;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "library_id")
    @JsonProperty(access = JsonProperty.Access.WRITE_ONLY)
    private Library library;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}

```

```

    }

    public void setName(String name) {
        this.name = name;
    }

    public Library getLibrary() {
        return library;
    }

    public void setLibrary(Library library) {
        this.library = library;
    }
}

```

@Entity annotation is required to specify a JPA and Hibernate entity

@Id annotation is required to specify the identifier property of the entity

@OneToMany and @ManyToOne defines a one-to-many and many-to-one relationship between 2 entities.

@JoinColumn specifies the foreign key column. mappedBy indicates the entity is the inverse of the relationship

@OneToMany should be placed on the parent entity (the One side), and @ManyToOne should be placed on the child entity (the Many side)

The default fetchType of @ManyToOne is EAGER which can cause performance issue, so here we change it to LAZY

@JsonProperty(access = JsonProperty.Access.WRITE\_ONLY) is for the REST APIs section below to ignore the property when serializing it to JSON string, due to library is a LAZY association which can throw LazyInitializationException if it is uninitialized in a non-transactional context

CascadeType.ALL is for propagating the CRUD operations on the parent entity to the child entities.

CascadeType.ALL should be used for small child collection only as it can cause performance issue, we will dig more into this in the later part

## Extend Spring Data JPA Repository Interfaces

Spring Data JPA provides a collection of repository interfaces that help reducing boilerplate code required to implement the data access layer for various databases

Let's create LibraryRepository and BookRepository interfaces, then extend them from JpaRepository

```

package com.hellokoding.jpa.library;

import org.springframework.data.jpa.repository.JpaRepository;

public interface LibraryRepository extends JpaRepository<Library, Integer>{
}

```

```

package com.hellokoding.jpa.unidirectional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Modifying;
import org.springframework.data.jpa.repository.Query;

import javax.transaction.Transactional;

public interface BookRepository extends JpaRepository<Book, Integer>{
}

```

## Create the One-to-Many CRUD REST APIs to access database

Create LibraryController and BookController to define CRUD REST APIs for accessing the database via the one to many relationship mapping

```

package com.hellokoding.jpa.bidirectional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;
import java.util.Optional;

@RestController
@RequestMapping("/api/v1/libraries")
public class LibraryController {
    private final LibraryRepository libraryRepository;
    private final BookRepository bookRepository;

    @Autowired
    public LibraryController(LibraryRepository libraryRepository, BookRepository bookRepository) {
        this.libraryRepository = libraryRepository;
        this.bookRepository = bookRepository;
    }

    @PostMapping
    public ResponseEntity<Library> create(@Valid @RequestBody Library library) {
        Library savedLibrary = libraryRepository.save(library);
        URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
            .buildAndExpand(savedLibrary.getId()).toUri();

        return ResponseEntity.created(location).body(savedLibrary);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Library> update(@PathVariable Integer id, @Valid @RequestBody Library library) {
        Optional<Library> optionalLibrary = libraryRepository.findById(id);
    }
}

```

```

        if (!optionalLibrary.isPresent()) {
            return ResponseEntity.unprocessableEntity().build();
        }

        library.setId(optionalLibrary.get().getId());
        libraryRepository.save(library);

        return ResponseEntity.noContent().build();
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Library> delete(@PathVariable Integer id) {
        Optional<Library> optionalLibrary = libraryRepository.findById(id);
        if (!optionalLibrary.isPresent()) {
            return ResponseEntity.unprocessableEntity().build();
        }

        libraryRepository.delete(optionalLibrary.get());

        return ResponseEntity.noContent().build();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Library> getById(@PathVariable Integer id) {
        Optional<Library> optionalLibrary = libraryRepository.findById(id);
        if (!optionalLibrary.isPresent()) {
            return ResponseEntity.unprocessableEntity().build();
        }

        return ResponseEntity.ok(optionalLibrary.get());
    }

    @GetMapping
    public ResponseEntity<Page<Library>> getAll(Pageable pageable) {
        return ResponseEntity.ok(libraryRepository.findAll(pageable));
    }
}

```

```

package com.hellokoding.jpa.bidirectional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;
import java.util.Optional;

@RestController
@RequestMapping("/api/v1/books")
public class BookController {
    private final BookRepository bookRepository;
    private final LibraryRepository libraryRepository;

    @Autowired

```

```

public BookController(BookRepository bookRepository, LibraryRepository libraryRepository) {
    this.bookRepository = bookRepository;
    this.libraryRepository = libraryRepository;
}

@PostMapping
public ResponseEntity<Book> create(@RequestBody @Valid Book book) {
    Optional<Library> optionalLibrary = libraryRepository.findById(book.getLibrary().getId());
    if (!optionalLibrary.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    book.setLibrary(optionalLibrary.get());

    Book savedBook = bookRepository.save(book);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
        .buildAndExpand(savedBook.getId()).toUri();

    return ResponseEntity.created(location).body(savedBook);
}

@PutMapping("/{id}")
public ResponseEntity<Book> update(@RequestBody @Valid Book book, @PathVariable Integer id) {
    Optional<Library> optionalLibrary = libraryRepository.findById(book.getLibrary().getId());
    if (!optionalLibrary.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    Optional<Book> optionalBook = bookRepository.findById(id);
    if (!optionalBook.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    book.setLibrary(optionalLibrary.get());
    book.setId(optionalBook.get().getId());
    bookRepository.save(book);

    return ResponseEntity.noContent().build();
}

@DeleteMapping("/{id}")
public ResponseEntity<Book> delete(@PathVariable Integer id) {
    Optional<Book> optionalBook = bookRepository.findById(id);
    if (!optionalBook.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    bookRepository.delete(optionalBook.get());

    return ResponseEntity.noContent().build();
}

@GetMapping
public ResponseEntity<Page<Book>> getAll(Pageable pageable) {
    return ResponseEntity.ok(bookRepository.findAll(pageable));
}

@GetMapping("/{id}")
public ResponseEntity<Book> getById(@PathVariable Integer id) {

```



```

    Optional<Book> optionalBook = bookRepository.findById(id);
    if (!optionalBook.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    return ResponseEntity.ok(optionalBook.get());
}
}

```

## Application Configurations

Configure the Spring Datasource JDBC URL, user name, and password of your local MySQL server in `application.properties`

```

spring.datasource.url=jdbc:mysql://localhost:3306/test?useSSL=false&allowPublicKeyRetrieval=true
spring.datasource.username=root
spring.datasource.password=hellokoding

spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.hibernate.ddl-auto=create
spring.jpa.show-sql=true

```

Create the test database in your local MySQL server if not exists

We don't have to create table schemas, the `ddl-auto=create` config allows JPA and Hibernate does that based on the entity-relationship mappings. In practice, consider to use `ddl-auto=none` (default) and use a migration tool such as [Flyway](#) for better database management

`spring.jpa.show-sql=true` for showing generated SQL queries in the application logs, consider to disable it on production environment

## Run the application

We use `@SpringBootApplication` to launch the application

```

package com.hellokoding.jpa.bidirectional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}

```

Type the mvn command at the project root directory to start the application

```
./mvnw clean spring-boot:run -Dstart-class=com.hellokoding.jpa.bidirectional.Application
```

Access to your local MySQL Server to query the table schemas created by JPA and Hibernate based on your entity mapping

```
[mysql> describe library;
+-----+-----+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id    | int(11)       | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255)  | NO   |     | NULL    |                |
+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)

[mysql> describe book;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | NO   | PRI | NULL    | auto_increment |
| name       | varchar(255)  | NO   |     | NULL    |                |
| library_id | int(11)       | NO   | MUL | NULL    |                |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.01 sec)
```

## The pros of @OneToMany

Let's test the one to many REST APIs with Postman

Thanks to the CascadeType.ALL setting with @OneToMany mapping on the parent entity

```
@OneToMany(mappedBy = "library", cascade = CascadeType.ALL)
private Set<Book> books = new HashSet<>();
```

You can cascade the CRUD operations on the parent to child collection by using a single line of code

1) Create a list of new child entities when creating a new parent via `libraryRepository.save(library);` in the Create Library API

```
@PostMapping
public ResponseEntity<Library> create(@Valid @RequestBody Library library) {
    Library savedLibrary = libraryRepository.save(library);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
        .buildAndExpand(savedLibrary.getId()).toUri();

    return ResponseEntity.created(location).body(savedLibrary);
}
```

► Library Create

POST localhost:8080/api/v1/libraries


Params Authorization Headers (8) **Body** Pre-request Script Tests Settings


none form-data x-www-form-urlencoded **raw** binary GraphQL JSON ▼

```

1 {
2   "name": "HelloKoding",
3   "books": [
4     {
5       "name": "Spring Boot In Practice"
6     },
7     {
8       "name": "Java Tutorials"
9     }
10  ]
11 }

```

Body Cookies Headers (6) Test Results  Status: 201 Created

Pretty Raw Preview Visualize JSON ▼ 

```

1 {
2   "id": 1,
3   "name": "HelloKoding",
4   "books": [
5     {
6       "id": 1,
7       "name": "Spring Boot In Practice"
8     },
9     {
10      "id": 2,
11      "name": "Java Tutorials"
12    }
13  ]
14 }

```

2) Create a list of new child entities when updating an existing parent via `libraryRepository.save(library);` in the Update Library API

```

@PutMapping("/{id}")
public ResponseEntity<Library> update(@PathVariable Integer id, @Valid @RequestBody Library library) {
    Optional<Library> optionalLibrary = libraryRepository.findById(id);
    if (!optionalLibrary.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    library.setId(optionalLibrary.get().getId());
    libraryRepository.save(library);

    return ResponseEntity.noContent().build();
}

```

3) Retrieve a list of child entities when retrieving a parent entity via `libraryRepository.findById(id);` in the Get Library API

```

@GetMapping("/{id}")
public ResponseEntity<Library> getById(@PathVariable Integer id) {
    Optional<Library> optionalLibrary = libraryRepository.findById(id);
    if (!optionalLibrary.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    return ResponseEntity.ok(optionalLibrary.get());
}

```

4) Delete a library by ID and all books belong to it via

`libraryRepository.delete(optionalLibrary.get());` in the Delete API

```

@DeleteMapping("/{id}")
public ResponseEntity<Library> delete(@PathVariable Integer id) {
    Optional<Library> optionalLibrary = libraryRepository.findById(id);
    if (!optionalLibrary.isPresent()) {
        return ResponseEntity.unprocessableEntity().build();
    }

    libraryRepository.delete(optionalLibrary.get());

    return ResponseEntity.noContent().build();
}

```

## The cons of @OneToMany

Everything is a tradeoff, and indeed the @OneToMany sugar syntax comes with performance issues

1) It is not possible to limit the size of the @OneToMany collection directly from the database. So all the APIs that retrieving the @OneToMany collection may hit a performance issue when the collection size grows

2) Using CascadeType.ALL and CascadeType.REMOVE with @OneToMany can cause a performance issue

The above Delete API works fine on the surface, but if look into the console log, we would see the following

```

Hibernate: select library0_.id as id1_1_0_, library0_.name as name2_1_0_ from library library0_ where library0_.id=
Hibernate: select books0_.library_id as library_3_0_0_, books0_.id as id1_0_0_, books0_.id as id1_0_1_, books0_.lib
Hibernate: delete from book where id=?
Hibernate: delete from book where id=?
Hibernate: delete from library where id=?

```

To do the REMOVE cascading operations, Hibernate has to generate 1 SQL to get all child entities and N+1 DELETE queries. Obviously, to delete the child collection, we only need 1 DELETE query and we don't have to fetch the entire collection

# A compromise approach for @OneToMany

We can fix the performance issues of @OneToMany, but have to trade the convenient cascading and navigating operations

1) Update @OneToMany settings to use cascade = {CascadeType.PERSIST,CascadeType.MERGE} instead of CascadeType.ALL, and remove the getChildCollection() method from the parent entity

```
@Entity
public class Library {
    ...

    @OneToMany(mappedBy = "library", cascade = {CascadeType.PERSIST,CascadeType.MERGE})
    private Set<Book> books = new HashSet<>();

    ...

    // public Set<Book> getBooks() {
    //     return books;
    // }
}
```

2) Add custom delete and select queries into the BookRepository

```
public interface BookRepository extends JpaRepository<Book, Integer>{
    Page<Book> findByLibraryId(Integer libraryId, Pageable pageable);

    @Modifying
    @Transactional
    @Query("DELETE FROM Book b WHERE b.library.id = ?1")
    void deleteByLibraryId(Integer libraryId);
}
```

3) Update LibraryController to use the new methods from BookRepository

```
public class LibraryController {
    ...

    @DeleteMapping("/{id}")
    public ResponseEntity<Library> delete(@PathVariable Integer id) {
        Optional<Library> optionalLibrary = libraryRepository.findById(id);
        if (!optionalLibrary.isPresent()) {
            return ResponseEntity.unprocessableEntity().build();
        }

        deleteLibraryInTransaction(optionalLibrary.get());

        return ResponseEntity.noContent().build();
    }

    @Transactional
    public void deleteLibraryInTransaction(Library library) {
```

```

        bookRepository.deleteByLibraryId(library.getId());
        libraryRepository.delete(library);
    }

    ...

}

```

4) Add a new API into BookController for retrieving the collection association by a parent entity id

```

public class BookController {

    ...

    @GetMapping("/library/{libraryId}")
    public ResponseEntity<Page<Book>> getByLibraryId(@PathVariable Integer libraryId, Pageable pageable) {
        return ResponseEntity.ok(bookRepository.findByLibraryId(libraryId, pageable));
    }

    ...

}

```

Restart the application and test the Delete Library API with Postman again you would see that to delete the child collection, Hibernate only generates 1 DELETE query

Hibernate: delete from book where library\_id=?

As we remove the `getBooks()` method from the Library entity, the `create`, `update`, and `getById` APIs won't return the list books anymore but we can fetch it via the `getByLibraryId` API or if you would like to include it in the response of those APIs, consider to use the DTO design pattern like the suggestion in the later part

## When to use @OneToMany

In summary, you can use `@OneToMany` if the child collection size is limited, otherwise, if the child collection can grow to a lot of items, consider to

- Don't retrieve `@OneToMany` child collection directly from the parent, you can retrieve via custom queries on the repositories like the step 2 above
- Don't use `CascadeType.REMOVE` or `CascadeType.ALL` with `@OneToMany`

## Unidirectional mapping with the only @ManyToOne

You can use [the only @ManyToOne to do the mapping](#) for the One to Many unidirectional relationship. You may have to do more with the only @ManyToOne but it can help you worry less about the potential issues of @OneToMany

## Convert JPA and Hibernate entities to DTO objects

In practice, one JPA and Hibernate entity model can not fit the various needs of clients. So instead of exposing directly, you may like [converting JPA and Hibernate entities to DTO objects](#) for providing custom API response data to the client

## Conclusion

In this tutorial, we learned about bidirectional mapping the One-To-Many relationship with @OneToMany and @ManyToOne and expose it through REST APIs in Spring Boot and Spring Data JPA to do CRUD operations against a MySQL database. We also had a look at the pros and cons of using @OneToMany. The source code is available on [Github](#)

You may also like the other tutorials about entity relationship mapping in JPA and Hibernate

- [Composite Primary Key](#)
- [One to One with Foreign Key](#)
- [One to One with Shared Primary Key](#)
- [One to Many Unidirectional Mapping](#)
- [Many to Many without Extra Columns](#)
- [Many to Many with Extra Columns](#)

[# JPA and Hibernate](#)[# Spring Boot](#)

---

### Share to social

[Twitter](#)[Facebook](#)

Giau Ngo

Giau Ngo is a software engineer, creator of HelloKoding. He loves coding, blogging, and traveling. You may find him on [GitHub](#) and [LinkedIn](#)

---

## Comments

---









Recent Articles

JPA and Hibernate Many To Many Mapping without Joined Entity in Spring Boot

---

JPA and Hibernate Many To Many Unidirectional Mapping with Joined Entity and Single Primary Key in Spring Boot

[i](#) [x](#)

BINANCE NFT

# Buy & Sell NFTs on Binance

Get started now with  
your first NFT

Open

## HelloKoding - Practical Coding Guides, Tutorials and Examples Series

[Guides](#)

[Spring Boot](#)

[JPA and Hibernate](#)

[REST with Spring](#)

[Security with Spring](#)

[Java](#)

[Data Structure](#)

[Algorithm](#)

[GitHub](#)

[Facebook](#)

[Twitter](#)

[LinkedIn](#)

---

© 2022 HelloKoding - Practical Coding Guides, Tutorials and Examples Series

Content on this site is available under the CC-BY-4.0 license