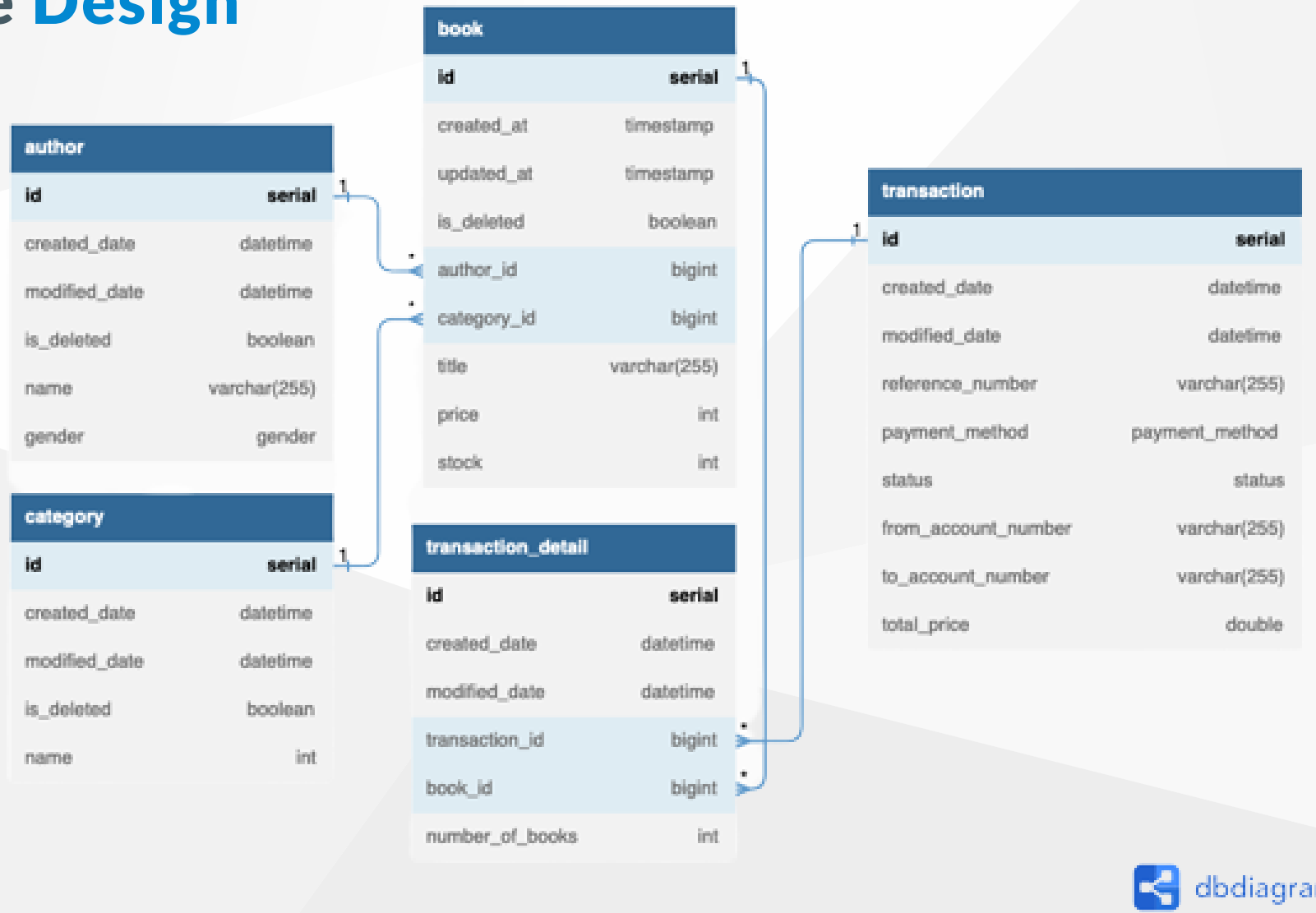
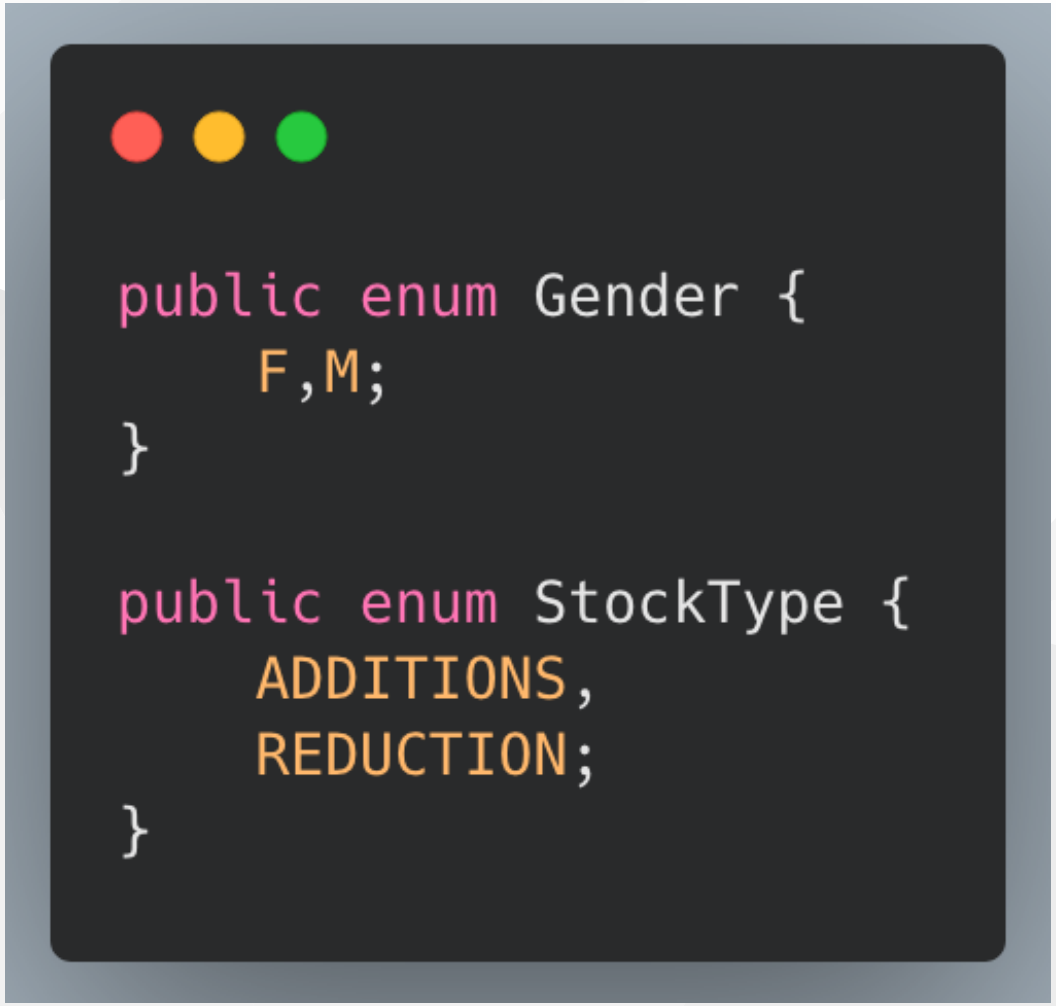
Spring Boot JPA Relationship

Rawlabs Academy

Database Design



Constant Enum



```
public enum Gender {
    F,M;
}

public enum StockType {
    ADDITIONS,
    REDUCTION;
}
```

Book DAO

In the previous material, we have created a `Book` model and then adjust it as in the example

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "book")
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Schema(description = "Generated ID", requiredMode = Schema.RequiredMode.REQUIRED, example = "1")
    private Long id;

    @Column(name = "created_date")
    @Schema(description = "Created date", requiredMode = Schema.RequiredMode.REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime createdDate;

    @Column(name = "modified_date")
    @Schema(description = "Modified date", requiredMode = Schema.RequiredMode.NOT_REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime modifiedDate;

    @Column(name = "isDeleted")
    @Schema(description = "Is Deleted",requiredMode = Schema.RequiredMode.REQUIRED, example = "false")
    private Boolean isDeleted;

    @Column(name = "title", nullable = false)
    @Schema(description = "Book title", requiredMode = Schema.RequiredMode.REQUIRED, example = "Mastering Spring
Boot")
    private String title;

    @Column(name = "price", nullable = false)
    @Schema(description = "Book price", requiredMode = Schema.RequiredMode.REQUIRED, example = "1500000")
    private Integer price;

    @Column(name = "stock", nullable = false)
    @Schema(description = "Stock value", requiredMode = Schema.RequiredMode.REQUIRED, example = "100")
    private Integer stock;

}
```

Author DAO

- `@JsonIgnore` used for ignore field from json response
- `@OneToMany` based on DB design that mean the **Author** can have **many books**.
- `cascade` When we perform some action on the target entity, the same action will be applied to the associated entity
- `fetchType` how to fetch the data, `LAZY` is fetch **when needed** and `EAGER` fetch **immediatelly**

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "author")
public class Author {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Schema(description = "Generated ID", requiredMode = Schema.RequiredMode.REQUIRED, example = "1")
    private Long id;

    @Column(name = "created_date")
    @Schema(description = "Created date", requiredMode = Schema.RequiredMode.REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime createdDate;

    @Column(name = "modified_date")
    @Schema(description = "Modified date", requiredMode = Schema.RequiredMode.NOT_REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime modifiedDate;

    @Column(name = "isDeleted")
    @Schema(description = "Is Deleted",requiredMode = Schema.RequiredMode.REQUIRED, example = "false")
    private Boolean isDeleted;

    @Column(name = "name", nullable = false)
    @Schema(description = "Author name", requiredMode = Schema.RequiredMode.REQUIRED, example = "John Doe")
    private String name;

    @Column(name = "gender", nullable = false)
    @Schema(description = "Author gender", requiredMode = Schema.RequiredMode.REQUIRED, example = "M")
    @Enumerated(value = EnumType.STRING)
    private Gender gender;

    @JsonIgnore
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "author")
    private List<Book> books;

}
```

Category DAO

- `@OneToMany` based on DB design that mean the **Category** can have **many books**.
- `fetchType` how to fetch the data, `LAZY` is fetch **when needed** and `EAGER` fetch **immediatelly**
- `mappedBy` to be used for mapping attribute on the `Book` **DAO**

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Schema(description = "Generated ID", requiredMode = Schema.RequiredMode.REQUIRED, example = "1")
    private Long id;

    @Column(name = "created_date")
    @Schema(description = "Created date", requiredMode = Schema.RequiredMode.REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime createdDate;

    @Column(name = "modified_date")
    @Schema(description = "Modified date", requiredMode = Schema.RequiredMode.NOT_REQUIRED, pattern = "yyyy-MM-
ddTHH:mm:ss.XXXZ")
    private LocalDateTime modifiedDate;

    @Column(name = "isDeleted")
    @Schema(description = "Is Deleted",requiredMode = Schema.RequiredMode.REQUIRED, example = "false")
    private Boolean isDeleted;

    @Column(name = "name", nullable = false)
    @Schema(description = "Category name", requiredMode = Schema.RequiredMode.REQUIRED, example = "Programming")
    private String name;

    @JsonIgnore
    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "category")
    private List<Book> books;

}
```

Update the Book DAO

- `@ManyToOne` based on DB design that mean `Book` DAO have foreign key `author` and `category` then be mapped on each associated entity
- Foreign key will be generated automatically from `author` into `author_id` and `category` into `category_id` on database.

```
@ManyToOne
@Schema(
        description = "Book's Author'",
        requiredMode = Schema.RequiredMode.REQUIRED
)
private Author author;

@ManyToOne
@Schema(
        description = "Book's category",
        requiredMode = Schema.RequiredMode.REQUIRED
)
private Category category;
```

Data Transfer Object

(DTO)

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthorDto {

    @Schema(description = "Author name", requiredMode = Schema.RequiredMode.REQUIRED,
            example = "John")
    private String name;

    @Schema(description = "Author's gender", requiredMode = Schema.RequiredMode.REQUIRED,
            example = "M")
    private Gender gender;

}
```

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class CategoryDto {

    @Schema(description = "Category name", requiredMode = Schema.RequiredMode.REQUIRED,
            example = "Programming")
    private String name;

}
```

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class BookDto {

    @Schema(description = "Book title", requiredMode = Schema.RequiredMode.REQUIRED, example = "Mastering Spring Boot")
    private String title;

    @Schema(description = "Book price", requiredMode = Schema.RequiredMode.REQUIRED, example = "1500000")
    private Integer price;

    @Schema(description = "Author's ID", requiredMode = Schema.RequiredMode.REQUIRED, example = "1")
    private Long authorId;

    @Schema(description = "Category's ID", requiredMode = Schema.RequiredMode.REQUIRED, example = "1")
    private Long categoryId;

}
```

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class StockDto {

    @Schema(description = "Book update stock", requiredMode = Schema.RequiredMode.REQUIRED,
            example = "100")
    private Integer value;

    @Schema(description = "Stock type operation", requiredMode = Schema.RequiredMode.REQUIRED,
            example = "ADDITIONS")
    private StockType type;

}
```

Repository

(JPA Repository)

```java
@Repository
public interface AuthorRepository extends JpaRepository<Author, Long> {

    Author findAuthorByNameIgnoreCase(String name);

}

@Repository
public interface BookRepository extends JpaRepository<Book, Long> {

    @Query(value = "select b from Book b where upper(b.category.name) like upper(:category) and
upper(b.author.name) like upper(:author)")
    List<Book> findAllByCategoryAndAuthor(String category, String author);

}

@Repository
public interface CategoryRepository extends JpaRepository<Category, Long> {

    Category findCategoryByNameIgnoreCase(String name);

}
```

`BookRepository` Explanation

The query **"select b from Book b where upper(b.category.name) like upper(:category) and upper(b.author.name) like upper(:author)"** is Java Persistence Query Language **(JPQL)**

It will be translated into SQL by **"select b.* from book b join category c on c.id = b.category_id join author a on a.id = b.author_id where upper(c.name) like upper(:category) and upper(a.name) like upper(:author)"** on native query language.

Service

Author Service

The `AuthorService` will be used to create new author of the book.

```java
@Service
public class AuthorService {

    private final AuthorRepository authorRepository;

    @Autowired
    public AuthorService(AuthorRepository authorRepository) {
        this.authorRepository = authorRepository;
    }

    public Author createAuthor(AuthorDto request) {
        Author author = Author.builder()
                .createdDate(LocalDateTime.now())
                .isDeleted(Boolean.FALSE)
                .name(request.getName())
                .gender(request.getGender())
                .build();
        return authorRepository.save(author);
    }

}
```

Category Service

The `CategoryService` will be used to create new category of the book.

```java
@Service
public class CategoryService {

    private final CategoryRepository categoryRepository;

    @Autowired
    public CategoryService(CategoryRepository categoryRepository) {
        this.categoryRepository = categoryRepository;
    }

    public Category saveCategory(CategoryDto request) {
        Category category = Category.builder()
                .createdDate(LocalDateTime.now())
                .isDeleted(Boolean.FALSE)
                .name(request.getName())
                .build();

        return categoryRepository.save(category);
    }

}
```

Book Service Dependency Injection

```
private final AuthorRepository authorRepository;
private final BookRepository bookRepository;
private final CategoryRepository categoryRepository;

@Autowired
public BookService(AuthorRepository authorRepository, BookRepository bookRepository,
                   CategoryRepository categoryRepository) {
    this.authorRepository = authorRepository;
    this.bookRepository = bookRepository;
    this.categoryRepository = categoryRepository;
}
```

Save Book

`BookService`

- The flow is, find the `author` and `category` first because the `book` have relation to `author` and `category`
- Line **18 - 19** is set attribute **association**

```
1  public Book save(BookDto request) {
2      Optional<Author> author = authorRepository.findById(request.getAuthorId());
3      if (author.isEmpty()) {
4          throw new RuntimeException("Author not found");
5      }
6
7      Optional<Category> category = categoryRepository.findById(request.getCategoryId());
8      if (category.isEmpty()) {
9          throw new RuntimeException("Category not found");
10     }
11
12     Book book = Book.builder()
13             .title(request.getTitle())
14             .price(request.getPrice())
15             .createdDate(LocalDateTime.now())
16             .isDeleted(Boolean.FALSE)
17             .stock(0)
18             .author(author.get())
19             .category(category.get())
20             .build();
21     return bookRepository.save(book);
22 }
```

Get List Book

`Book Service`

Why not direct use book repository?

`Author` and `Category` have `OneToMany` relation. It can be used to get many book where have foreign key of each **Author** or **Category**

```java
public List<Book> getBooksByCategory(String categoryName) {
    Category category = categoryRepository.findCategoryByNameIgnoreCase(categoryName);
    if (category != null) return category.getBooks();

    return new ArrayList<>();
}

public List<Book> getBooksByAuthor(String authorName) {
    Author author = authorRepository.findAuthorByNameIgnoreCase(authorName);
    if (author != null) return author.getBooks();
    return new ArrayList<>();
}

public List<Book> getBooksByCategoryAndAuthor(String category, String author) {
    return bookRepository.findAllByCategoryAndAuthor(category, author);
}
```

Update Stock

`BookService`

The flow is find book by id first if book is empty will return an exception.

If **StockType** is `ADDITIONS` that mean stock will be **added** by request value.

or if **StockType** is `REDUCTION` that mean stock will be **reducted** by request value

```java
public Book updateStock(Long bookId, StockDto request) {
    Optional<Book> bookOptional = bookRepository.findById(bookId);
    if (bookOptional.isEmpty()) {
        // Book is empty
        throw new RuntimeException("Book not found");
    }

    Book book = bookOptional.get();
    Integer stock = book.getStock();

    if (StockType.ADDITIONS.equals(request.getType())) {
        stock = stock + request.getValue();
    } else {
        stock = stock - request.getValue();
    }

    book.setStock(stock);
    return bookRepository.save(book);
}
```

Controller

Author Controller

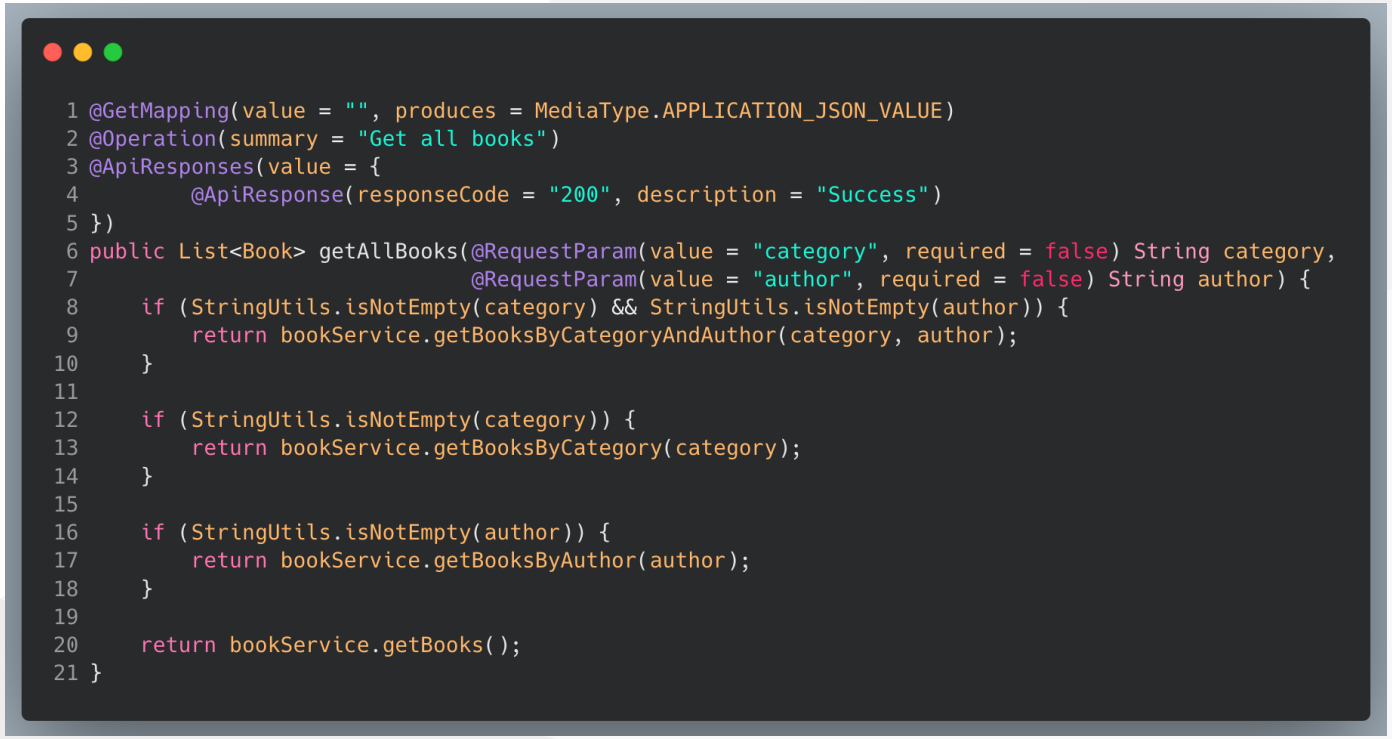
```java
@RestController
@RequestMapping(value = "/author")
public class AuthorController {

    private final AuthorService authorService;

    @Autowired
    public AuthorController(AuthorService authorService) {
        this.authorService = authorService;
    }

    @PostMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
    @Operation(summary = "Save new author")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Success")
    })
    public Author saveAuthor(@RequestBody AuthorDto request) {
        return authorService.createAuthor(request);
    }

}
```

Category Controller

```java
@RestController
@RequestMapping(value = "/category")
public class CategoryController {

    private final CategoryService categoryService;

    @Autowired
    public CategoryController(CategoryService categoryService) {
        this.categoryService = categoryService;
    }

    @PostMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
    @Operation(summary = "Save new category")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "200", description = "Success")
    })
    public Category saveCategory(@RequestBody CategoryDto request) {
        return categoryService.saveCategory(request);
    }

}
```

Get List Book

- If request param `author` and `category` is not empty will be get list book by author and category
- if request param only `author` is not empty will be get list book by author
- if request param only `category` is not empty will be get list book by category
- Or else get all books



```java
@GetMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
@Operation(summary = "Get all books")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Success")
})
public List<Book> getAllBooks(@RequestParam(value = "category", required = false) String category,
                              @RequestParam(value = "author", required = false) String author) {
    if (StringUtils.isNotEmpty(category) && StringUtils.isNotEmpty(author)) {
        return bookService.getBooksByCategoryAndAuthor(category, author);
    }

    if (StringUtils.isNotEmpty(category)) {
        return bookService.getBooksByCategory(category);
    }

    if (StringUtils.isNotEmpty(author)) {
        return bookService.getBooksByAuthor(author);
    }

    return bookService.getBooks();
}
```

Update Book Stock

```java
@PatchMapping(value = "/{id}/update-stock", produces = MediaType.APPLICATION_JSON_VALUE)
@Operation(summary = "Update book stock")
@ApiResponses(value = {
        @ApiResponse(responseCode = "200", description = "Success")
})
public Book updateStock(@PathVariable(value = "id") Long id,
                        @RequestBody StockDto request) {
    return bookService.updateStock(id, request);
}
```

Soft Deletes

What is Soft Deletes?

Deleting data permanently from a table is a common requirement when interacting with database. But, sometimes there are business requirements to **not permanently delete** data from the database.

The solution is we just **hide that data** so that can't be accessed from the front-end.

Implementation

By default, the delete command in the JPA repository will run a **SQL delete query**. So, let's first add some **annotation** to Entity class.

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "book")
@SQLDelete(sql = "update book set is_deleted = true where id = ?")
@Where(clause = "is_deleted = false")
public class Book {
    ...
}
```

JPA Query LIKE Example

```
1 @Repository
2 public interface BookRepository extends JpaRepository<Book, Long> {
3
4     List<Book> findBooksByTitleContaining(String title);
5
6     List<Book> findBooksByTitleContains(String title);
7
8     List<Book> findBooksByTitleIsContaining(String title);
9
10 }
```

JPA Query Count Example

```java
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {

    Integer countBooksByPriceBetween(Integer startPrice, Integer endPrice);

}
```

JPA Query Min and Max Example

```java
@Repository
public interface BookRepository extends JpaRepository<Book, Long> {

    @Query(value = "select min(b.stock) from Book b")
    Integer getMinimumStock();

    @Query(value = "select max(b.stock) from Book b")
    Integer getMaximumStock();

}
```


THANK YOU