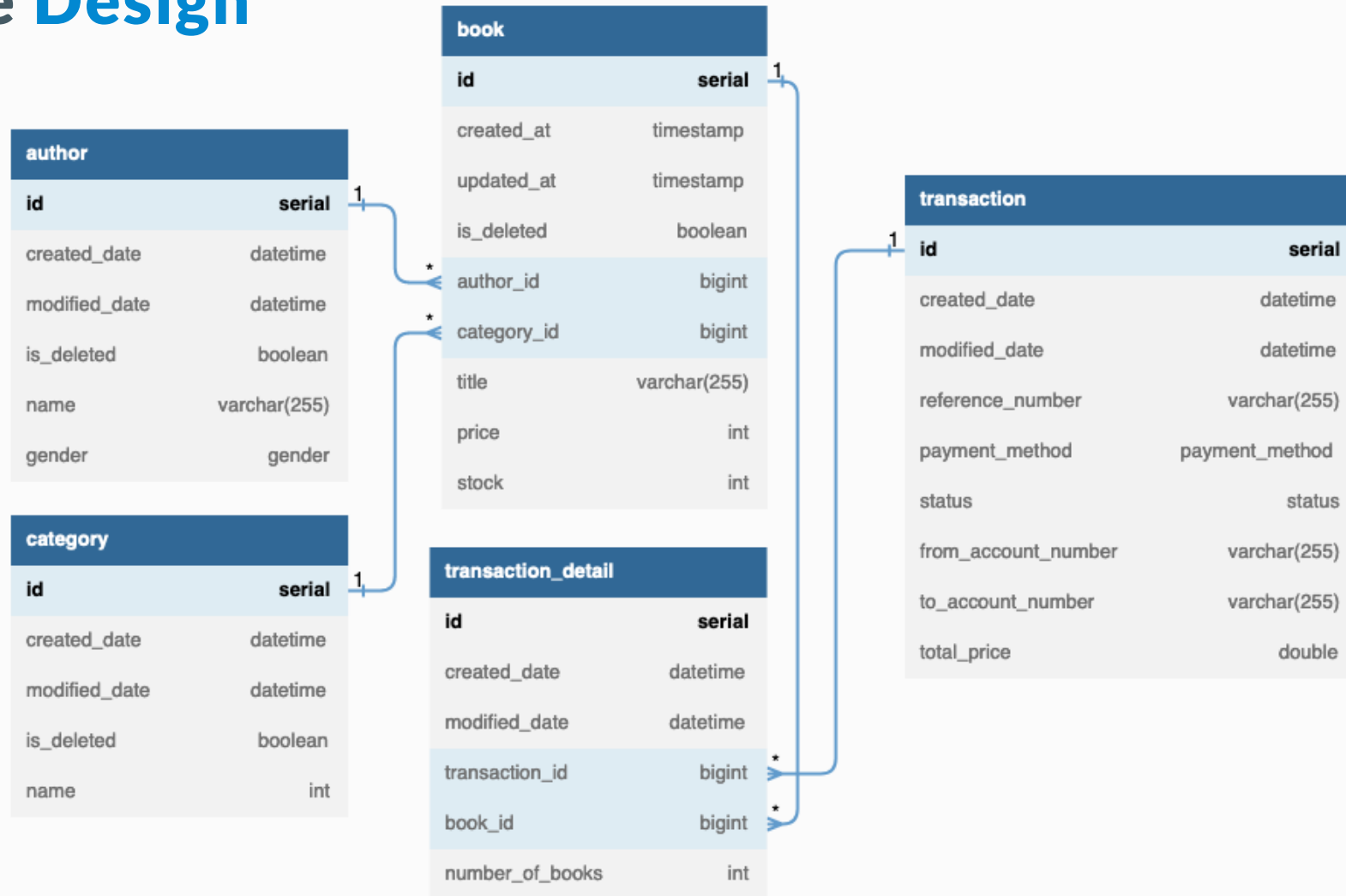




# Spring Boot JPA Relationship

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

```
@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 **immediately**

```
@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 createdAt;

    @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 **immediately**
- `mappedBy` to be used for mapping attribute on the **Book DAO**

```
@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 createdAt;

    @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)





```
@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;

}
```



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

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

}
```



```
@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;

}
```



```
@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)**



```
@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 `AuthService` will be used to create new author of the book.

```
@Service
public class AuthService {

    private final AuthorRepository authorRepository;

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

    public Author createAuthor(AuthorDto request) {
        Author author = Author.builder()
            .createdDate(LocalDate.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.

```
@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()
            .createdAt(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        .createdAt(LocalDate.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**

```
1 public List<Book> getBooksByCategory(String categoryName) {
2     Category category = categoryRepository.findCategoryByNameIgnoreCase(categoryName);
3     if (category != null) return category.getBooks();
4
5     return new ArrayList<>();
6 }
7
8 public List<Book> getBooksByAuthor(String authorName) {
9     Author author = authorRepository.findAuthorByNameIgnoreCase(authorName);
10    if (author != null) return author.getBooks();
11    return new ArrayList<>();
12 }
13
14 public List<Book> getBooksByCategoryAndAuthor(String category, String author) {
15     return bookRepository.findAllByCategoryAndAuthor(category, author);
16 }
```

# 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 **reduced** by request value

```
1 public Book updateStock(Long bookId, StockDto request) {
2     Optional<Book> bookOptional = bookRepository.findById(bookId);
3     if (bookOptional.isEmpty()) {
4         // Book is empty
5         throw new RuntimeException("Book not found");
6     }
7
8     Book book = bookOptional.get();
9     Integer stock = book.getStock();
10
11     if (StockType.ADDITIONS.equals(request.getType())) {
12         stock = stock + request.getValue();
13     } else {
14         stock = stock - request.getValue();
15     }
16
17     book.setStock(stock);
18     return bookRepository.save(book);
19 }
```

# Controller

# Author Controller

```
1 @RestController
2 @RequestMapping(value = "/author")
3 public class AuthorController {
4
5     private final AuthorService authorService;
6
7     @Autowired
8     public AuthorController(AuthorService authorService) {
9         this.authorService = authorService;
10    }
11
12    @PostMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
13    @ApiOperation(summary = "Save new author")
14    @ApiResponses(value = {
15        @ApiResponse(responseCode = "200", description = "Success")
16    })
17    public Author saveAuthor(@RequestBody AuthorDto request) {
18        return authorService.createAuthor(request);
19    }
20
21 }
```



# Category Controller

```
1 @RestController
2 @RequestMapping(value = "/category")
3 public class CategoryController {
4
5     private final CategoryService categoryService;
6
7     @Autowired
8     public CategoryController(CategoryService categoryService) {
9         this.categoryService = categoryService;
10    }
11
12    @PostMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
13    @Operation(summary = "Save new category")
14    @ApiResponses(value = {
15        @ApiResponse(responseCode = "200", description = "Success")
16    })
17    public Category saveCategory(@RequestBody CategoryDto request) {
18        return categoryService.saveCategory(request);
19    }
20
21 }
```

# 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

```
1 @GetMapping(value = "", produces = MediaType.APPLICATION_JSON_VALUE)
2 @Operation(summary = "Get all books")
3 @ApiResponses(value = {
4     @ApiResponse(responseCode = "200", description = "Success")
5 })
6 public List<Book> getAllBooks(@RequestParam(value = "category", required = false) String category,
7                               @RequestParam(value = "author", required = false) String author) {
8     if (StringUtils.isNotEmpty(category) && StringUtils.isNotEmpty(author)) {
9         return bookService.getBooksByCategoryAndAuthor(category, author);
10    }
11
12    if (StringUtils.isNotEmpty(category)) {
13        return bookService.getBooksByCategory(category);
14    }
15
16    if (StringUtils.isNotEmpty(author)) {
17        return bookService.getBooksByAuthor(author);
18    }
19
20    return bookService.getBooks();
21 }
```

# Update Book Stock



```
1 @PatchMapping(value =("/{id}/update-stock", produces = MediaType.APPLICATION_JSON_VALUE)
2 @Operation(summary = "Update book stock")
3 @ApiResponses(value = {
4     @ApiResponse(responseCode = "200", description = "Success")
5 })
6 public Book updateStock(@PathVariable(value = "id") Long id,
7     @RequestBody StockDto request) {
8     return bookService.updateStock(id, request);
9 }
```

# Soft Deletes

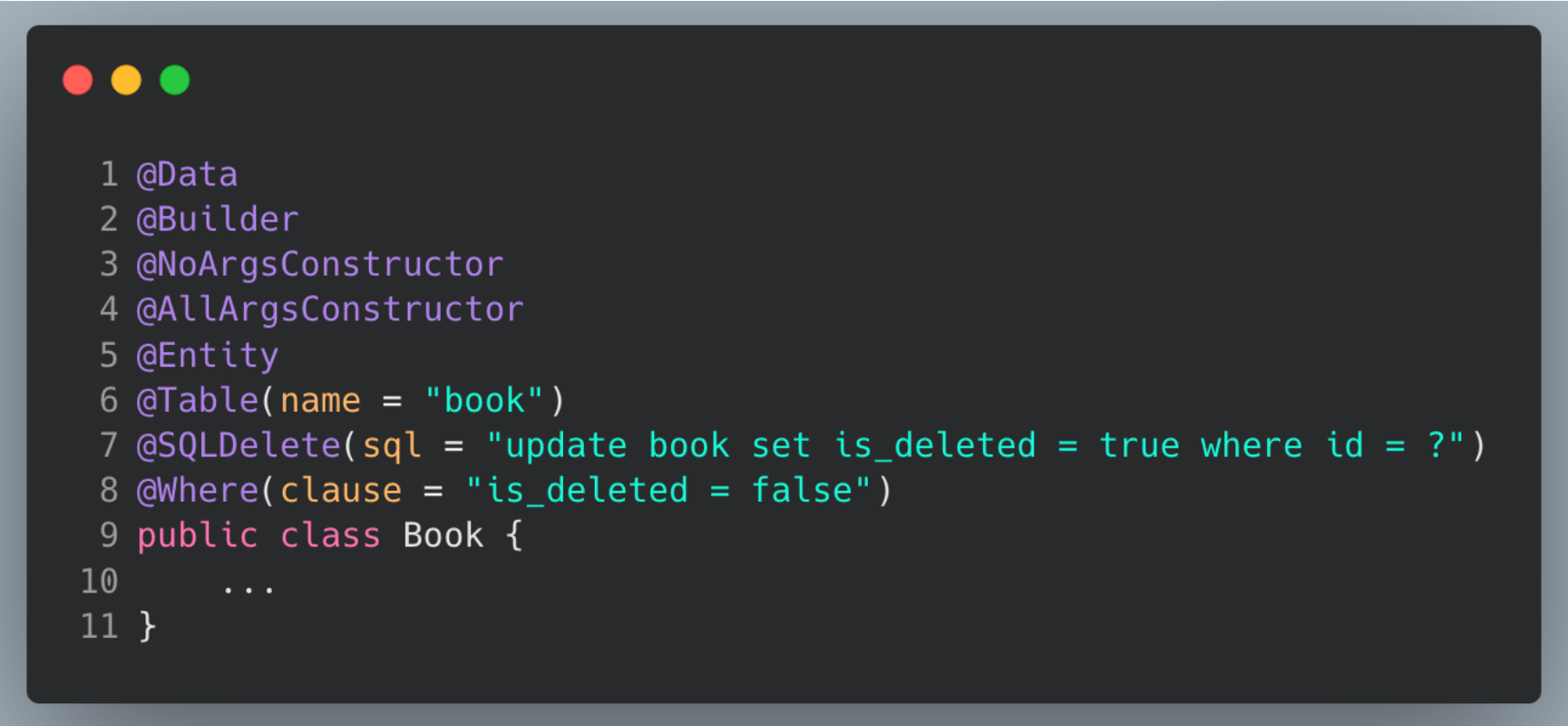
# What is **Soft Deletes**?

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.



```
1 @Data
2 @Builder
3 @NoArgsConstructor
4 @AllArgsConstructor
5 @Entity
6 @Table(name = "book")
7 @SQLDelete(sql = "update book set is_deleted = true where id = ?")
8 @Where(clause = "is_deleted = false")
9 public class Book {
10     ...
11 }
```

## 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



```
1 @Repository
2 public interface BookRepository extends JpaRepository<Book, Long> {
3
4     Integer countBooksByPriceBetween(Integer startPrice, Integer endPrice);
5
6 }
```



# JPA Query Min and Max Example



```
1 @Repository
2 public interface BookRepository extends JpaRepository<Book, Long> {
3
4     @Query(value = "select min(b.stock) from Book b")
5     Integer getMinimumStock();
6
7     @Query(value = "select max(b.stock) from Book b")
8     Integer getMaximumStock();
9
10 }
```

