Errores críticos:

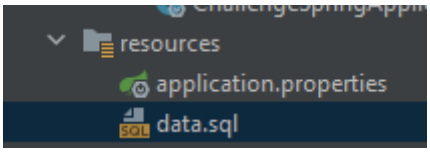
Error critico n°1:

Creo que un error critico es no saber como funciona la API, ni que requisitos debe cumplir. Debería haber alguna documentación para saber que es lo que debería hacer y cuales cosas debería cumplir.

Error critico n°2:

El primer error que tenemos al momento de compilar, es uno donde por consola se nos dice que la tabla a la que queremos solicitar no existe.

Esto sucede a causa de un archivo llamada “data.sql” que se encuentra en la carpeta “resources” en conjunto del “application.properties”, donde se está haciendo una llamada a la DB mediante un query. Este archivo que solo le veo como utilidad el hacer alguna prueba (porque no se que otra utilidad tendría) ya que de estar debería encontrarse en otro lugar (la carpeta test por ejemplo).



Archivo que no debería estar allí.

Por ende, mi solución en este caso es eliminar ese archivo que no hace falta y por otro lado, darle permisos al JPA para que sea el encargado de administrar las tablas de la DB, modificando la siguiente línea de código (que se encuentra en el “application.properties”) que inicialmente se encuentra en “none”.

```
spring.jpa.hibernate.ddl-auto=none
```

Y poniéndolo en “Update” (JPA actualice o modifique la DB si es necesario):

```
spring.jpa.hibernate.ddl-auto=update
```

Error critico n°3:

Luego nos encontramos con otro error que no permite compilar, el cual es:

```
mappedBy reference an unknown target entity property:
```

Error por consola.

Esto se debe a una mala configuración en la relación ManyToOne que hay entre el modelo “Category” y “Book”.

```java
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotEmpty
    private String title;

    private String author;

    private int edition;

    private double price;

    @ManyToOne(fetch = FetchType.LAZY, optional = false)
    @JoinColumn(name = "category_id", nullable = false, referencedColumnName = "id")
    private Category category;
}
```

Vista del modelo "Book" donde no está definido el targetEntity.

Para solucionarlo se le debe agregar al Annotation @ManyToOne el "targetEntity" para que el JPA sepa a quien pertenece esta unión.

```java
@ManyToOne(targetEntity = Category.class, fetch = FetchType.LAZY, optional = false)
@JoinColumn(name = "category_id", nullable = false, referencedColumnName = "id")
private Category category;
```

Se agrega el targetEntity en la Annotation @ManyToOne.

Así mismo también al modelo de Category, debemos agregar en el @OneToMany el "targetEntity" (que sería el Book.class) y también modificar el "mappedBy" ya que aquí debe ir el nombre del método que utiliza la clase Owner de la relación (en este caso sería Category).

```java
@OneToMany(targetEntity = Book.class,mappedBy = "category", fetch = FetchType.LAZY,
        cascade = CascadeType.ALL)
private Set<Book> books;
```

Se acomoda la relación One to many.

Terminado esto el programa lograría compilar e iniciar el API en el server local.

Malas prácticas:

Mala práctica n°1:

Se le debería agregar los nombres a cada entidad (tabla) que va a ser almacenada en la DB, de esa manera se tiene el control sobre las tablas que se están creando y es más fácil la lectura en el Workbench. (Ya que, si no se coloca, Java te crea el nombre por default, y para mí es mejor ponerlo uno mismo.)

```java
@Table(name = "users")
public class User implements Serializable {
```

Se le agregó el "name" a la tabla con el @Table.

Mala práctica n°2:

En el modelo User, se está repitiendo la validación del @NonNull y nullable = false. No es necesario ya que con solo la Annotation @NonNull se verifica que el dato no sea nulo.

```
@NonNull
@Column(nullable = false)
private String lastName;
```

Redundancia entre @NonNul y nullable = false.

Mala práctica n°3:

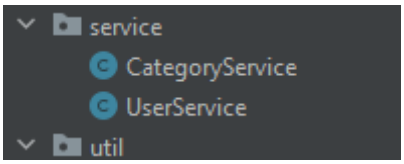
En el modelo User, el @Column(nullable = true) no hace falta, ya que este atributo viene por defecto en true.

```
@Column(nullable = true)
```

Línea innecesaria en el modelo User.

Mala práctica n°4:

Falta el BookService, para poder seguir el orden y estructura de una API REST. Se están haciendo operaciones en el BookController cuando esto debería hacerse en el Service.



Vista de los servicios donde se encuentra faltante el BookService.

Mala práctica n°5:

En la clase CategoryController no se esta respetando las capas (ya que las operaciones con el Repository deberían hacerse en el Service y no en el Controller).

```
@Autowired
private CategoryRepository categoryRepository;

@Autowired
private CategoryService categoryService;
```

Uso del Repository y el Service del modelo Category sin respetar las capas de la estructura Rest.

Errores de lógica:

Error n°1:

Redundancia en alguna de las Annotation del Lombok, ya que con poner el @Data ya te ofrece las opciones del @Getter, @Setter y el método toString.

```
@Getter
@Setter
@NoArgsConstructor
@Entity
@Data
@AllArgsConstructor
@Builder
@SQLDelete(sql = "UPDATE categories SET deleted = true WHERE id_categories=?")
@Where(clause = "deleted=false")
@Table(name = "Category")
public class Category implements Serializable {
```

Redundancia de algunas Annotations.

Así que no sería necesario escribir esas Annotations, ni tampoco definir el método toString ya que se encuentra contenido en el @Data.

```
@Override
public String toString() {
    return "Category{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", description='" + description + '\'' +
            ", images='" + images + '\'' +
            ", regdate=" + regdate +
            ", updatedate=" + updatedate +
            ", deleted=" + deleted +
            ", books=" + books +
            '}';
}
```

Método toString que podría eliminarse con el @Data.

Error n°2:

En la clase User faltan los getters y setters para poder acceder a sus atributos y así cumplir con el encapsulamiento. Para solucionarlo, podríamos poner la Annotation @Data. También esto hace que el código resulte más corto ya que no haría falta el método toString. Además, se podría agregar las Annotation @AllArgsConstructor y @NoArgsConstructor para no tener que definir los constructores.

Error n°3:

El ID de los modelos deberían ser definido como único (ya que debe existir un ID único para cada modelo en su respectiva tabla) y que no pueda ser null.

```java
public class Category implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

Método toString que podría eliminarse con el @Data.

Esto se arregla poniendo la siguiente línea de código encima de el atributo id del modelo.

```java
@Column(unique = true, nullable = false)
```

Error n°4:

En el UserService, está mal hecha la inyección de dependencias, ya que la Annotation @AutoWired se encarga de hacerlo de manera automática y en este proyecto se está tratando de hacer manualmente y al mismo tiempo se usa la Annotation.

```java
private final UserRepository userRepository;

@Autowired
public UserService(UserRepository userRepository) { this.userRepository = userRepository; }
```

Inyección de dependencia mal hecha en el UserService

Esto debería quedar de esta manera:

```java
@Autowired
UserRepository userRepository;
```

Corrección al error antes expuesto.

Error n°5:

En el UserService, está mal definido el método “loadUserByUsername”, ya que nunca retorna lo que se mandó a buscar (ya que se está retornando null) y aparte se especifica que lanza una excepción que no está siendo declarada en la lógica del algoritmo.

```java
@Override
public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
    userRepository.findByEmail(email);
    return null;
}
```

Error lógico en el método loadUserByUsername, que retorna null.

Error n°6:

En la clase BookController están invertidas las Annotations de Get y Post. También falta delegar la responsabilidad al Service y no al Repository.

```java
@PostMapping
public ResponseEntity<?> getBooks(){
    return new ResponseEntity<>(bookRepository.findAll(), HttpStatus.BAD_REQUEST);
}


@GetMapping
public ResponseEntity<?> createBook(@RequestBody Book book){
    return new ResponseEntity<>(bookRepository.save(book), HttpStatus.CREATED);
}
```

Error en la función de los métodos Post y Get de la clase BookController.

Debería verse así:

```java
@PostMapping
public ResponseEntity<?> createBook(@RequestBody Book book){
    return new ResponseEntity<>(bookService.createBook(book), HttpStatus.CREATED);
}

@GetMapping("/colletctorPrice/book/{id}")
public ResponseEntity<?> getCollectorPrice (@PathVariable long id){

    return new ResponseEntity<>(bookService.getCollectorPrice(id).getPrice(), HttpStatus.OK);
}
```

Posible solución al error antes mencionado.

Error n°7:

En el método getCategories de la clase CategoryController, el llenado de la lista es una responsabilidad que debería encargarse el Service y no el Controller.

```java
@GetMapping
public ResponseEntity<?> getCategories(){
    List<Category> listCategories = categoryRepository.findAll();
    List<String> listCategoriesByname = new ArrayList<>();
    for (Category c : listCategories) {
        listCategoriesByname.add(c.getName());
    }
    return new ResponseEntity(listCategoriesByname, HttpStatus.OK);
}
```

El Controller haciendo un procedimiento que debería hacer el Service.