Singleton
In order to avoid creating new connections continuously to the database, Singleton Pattern Design was chosen.

Factory
Orders, customers and books were created with the Factory design pattern because an interface is created, and if in the future new types are requested it is only created the new classes following the interface.
In order to illustrate the idea, here is shown Customers:
When a customer is needed:

```
@PostMapping("/customers")
public Customer addCustomer(@RequestBody Customer customerRequest) throws Error {
    try {
        Customer newCustomer = objectMapper.readValue(objectMapper.writeValueAsString(customerRequest),
        valueType:Customer.class);
        return customerRepository.save(newCustomer);
     } catch (JsonProcessingException e) {
        throw new Error();
    }
}
```

Customer.java

```
@Entity
@Table(name = "customer")
@JsonDeserialize(using = CustomerDeserializer.class)
public interface Customer {

    @Id
    public Long getId();
    public void setId(Long id);
    public String getName();
    public void setName(String name);
    public String getAddress();
    public void setAddress(String address);
    public String getType();
    public void setType(String type);

}
```

And the Deserializer switches between the types of customers:

```java
@Override
public Customer deserialize(JsonParser jp, DeserializationContext ctxt) throws IOException {
    ObjectMapper mapper = (ObjectMapper) jp.getCodec();
    JsonNode node = mapper.readTree(jp);

    // Extract necessary information from the JSON node
    String type = node.get(fieldName:"type").asText();
    String name = node.get(fieldName:"name").asText();
    String address = node.get(fieldName:"address").asText();

    // Create a Customer instance based on the type
    switch (type.toLowerCase()) {
        case "minor":
            return new MinorCustomer(name, address);
        case "major":
            return new MajorCustomer(name, address);
        default:
            throw new IllegalArgumentException("Invalid customer type: " + type);
    }
}
```

And then the different types of customers implement the Customer interface:

```java
@Entity
@JsonDeserialize(as = Customer.class)
public class MinorCustomer implements Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String customerName;
    private String customerAddress;
    private String type;

    public MinorCustomer(String name, String address) {
        this.customerName = name;
        this.customerAddress = address;
    }
```

```java
@Entity
@JsonDeserialize(as = Customer.class)
public class MajorCustomer implements Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String companyName;
    private String companyAddress;
    private String type;

    public MajorCustomer(String name, String address) {
        this.companyName = name;
        this.companyAddress = address;
    }
```

Strategy

The use of this pattern brings the possibility of creating new classes for different types of reports without having to modify the existing code. Every new type of report will have to implement an interface containing the method that generates the report.

```java
interface ReportsGenerator {
    String generateReport(String someData);
}
```

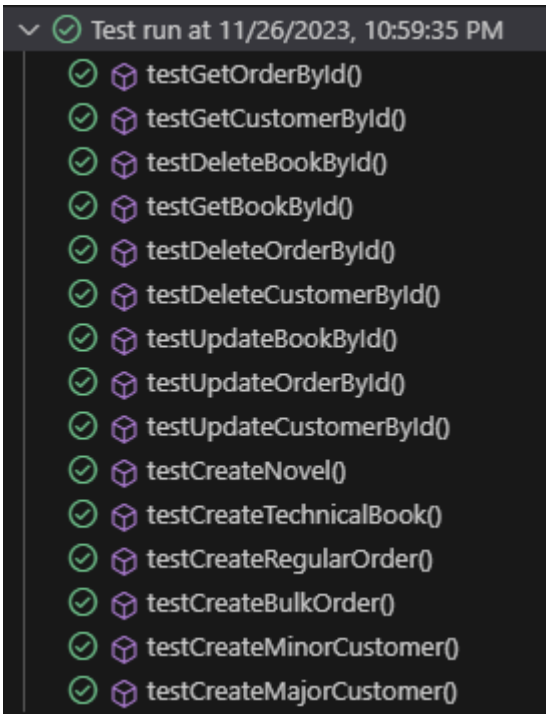
We have an interface with the method that will generate the report.

```java
public class SalesReport implements ReportsGenerator {
    @Override
    public String generateReport(String someData){
        return "Some mocked sales report based on: " + someData;
    }
}
```

And each class that will generate the desired report will implement the ReportsGenerator class.

Tests

In order to test the correctness of the functionality of the code, SpringBootTest was used. The output was as followed, being all the endpoints tested:



Also, a Postman collection is delivered in order to play with the different endpoints: