# Assignment 2 Technical Report

**Course**: COSC2440 - Software Architecture: Design & Implementation

**Semester:** 2021A

**Members:**

1. Truong Duc Khai - s3818074
2. Chung Quan Tin - s3818487

**Lecturer**: Minh Vu Thanh

## Table of contents

## I. Introduction

This report will describe the design and implementation of the backend which is a REST API of a Trade Management System. The content will include a brief description of how business will use this application, the technology and architecture used during the project development followed with the diagrams and implementation outcomes.

This report will not include the documentation and instructions on how to use the API programmatically.

## II. Business requirement

### 1. Description

This system is a REST API built for trading companies to manage their trading data and automate their business processes. The system will cover all create-read-update-delete (CRUD) operations on individuals which involve in trading such as staff and customers; and trading transactions such as order, delivery notes, receiving notes and sales invoices. The API will also provide additional functionalities such as searching and statistics which allow the company to easily view and filter the data based on their needs.

### 2. Functionalities and how company can use them

#### a) Manage customers

The company will be able to CRUD on their customers, who they are going to sell the products to. The company can store information about each customer such as name, phone number, email address, etc. The API also allows the company to view the customer revenue, which can be filtered by time period. To make managing a large number of customers easier, the customer can be searched by name, address or phone number.

#### b) Manage staffs

Staff is the person who will handle the trading processes, which involves buying the products from providers and selling the products to customers. Similar to customers, the company can CRUD on their staff and view their revenue from their sales invoices they make to customers.

#### c) Manage orders

The staff will buy the products from providers by making orders. Therefore, the API allows the company to CRUD on orders. An order will have order details which specify the quantity of each item and its price copied from the database. After an order has been created, the providers can view that order, prepare the products and sell them to the company with the calculated price.

### d) Manage inventory receiving notes

Products shipped to the company will need to be recorded in inventory receiving notes. The API allows the company to CRUD on receiving notes and filter the receiving notes by time period, making it easier for them to keep track of how many of each product are shipped to the inventory. Similar to an order, a receiving note will include receiving details where each includes a product with its quantity. The staff who makes the note and the date will also be recorded. To make it easier for warehouse keepers, the data of a receiving note can be transferred from an order, which copies the staff and the date; and all order details will become note details.

### e) Manage inventory delivery notes

When the company sells products to a customer, it will prepare inventory delivery notes. Similar to a receiving note, the company can also CRUD on delivery notes and filter them by time period. The staff who makes the note and the date will also be recorded.

### f) Manage sales invoices

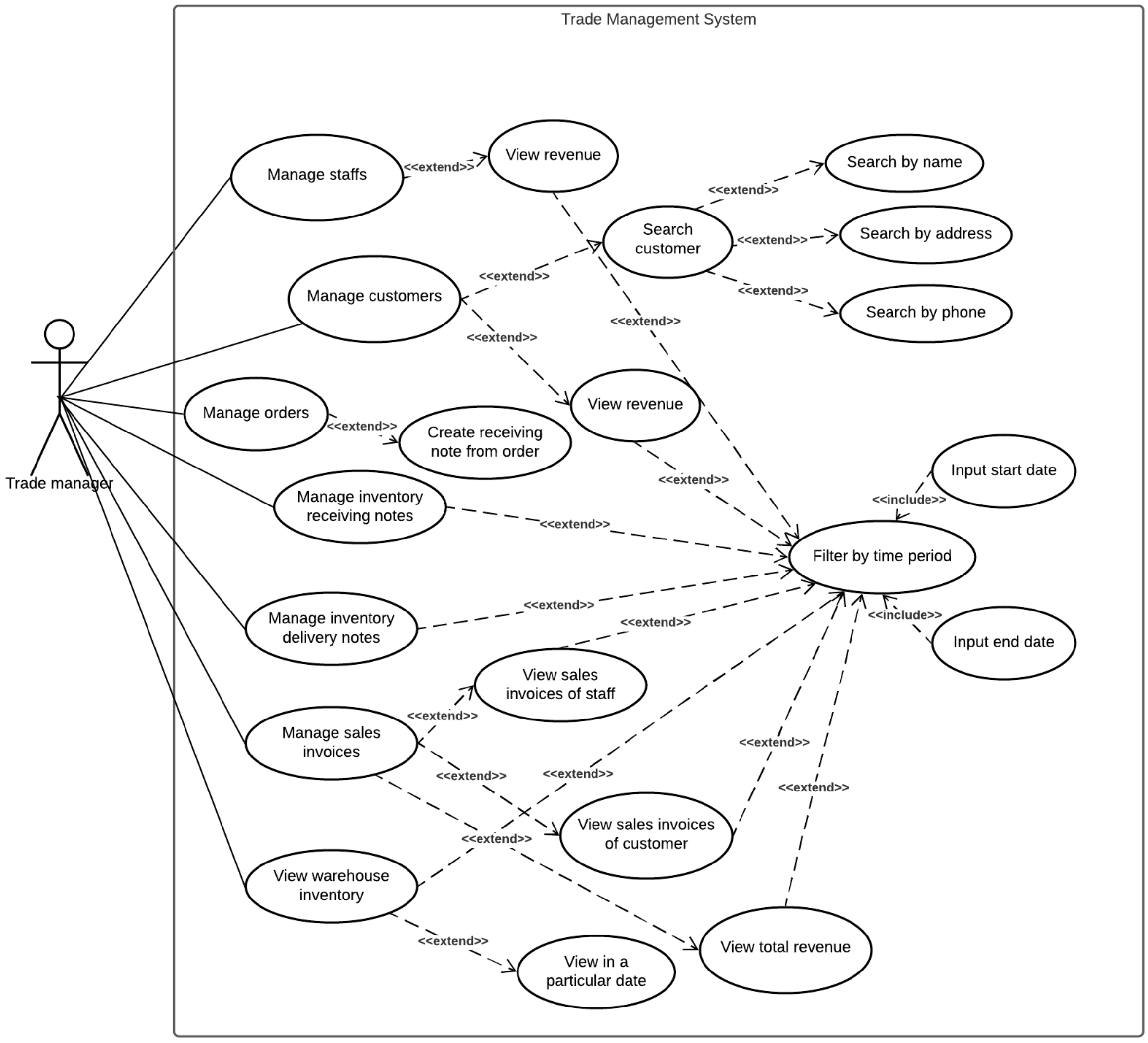
When shipping products to the customer, the company will prepare a sales invoice with sales details which include quantity and price of the products bought by that customer. The company can CRUD on sales invoices. The total price of a sales invoice will be calculated from the sales details, and the company can modify them through the API if needed. The staff who prepare the sales invoice and the date will also be recorded.

To make managing a large number of sales invoices easier, the API provides the functionality to search sales invoices and view total revenue by time period, by staff or by customer. This allows the company to have a clear overview on their sales data to make better trading decisions.

### g) View warehouse inventory

The API also allows the company to view the shipping information of each product in a particular date or a period, which helps them to manage the quantity of their products better, avoiding issues such as out-of-stock or having too many products kept in stock.

## III. Use case diagram



*(view the UseCaseDiagram.pdf if this is not clear enough)*

## IV. Technology use

### 1. Spring Boot

An open-source framework which is considered as the close-knit companion of Java due to its utility of the architectural concepts such as IoC (Inversion of Control) and DI (Dependency Injection). We take Spring Boot over Spring because Spring Boot only needs a minimal Spring configuration to create a stand-alone Spring application.

### 2. PostgreSQL

A free and open-source relational database management system (RDBMS). As Spring Data JPA has a large support for relational databases, we choose

PostgreSQL as our primary database to integrate it with Spring Data JPA in our Spring project.

3. **Spring Hibernate**

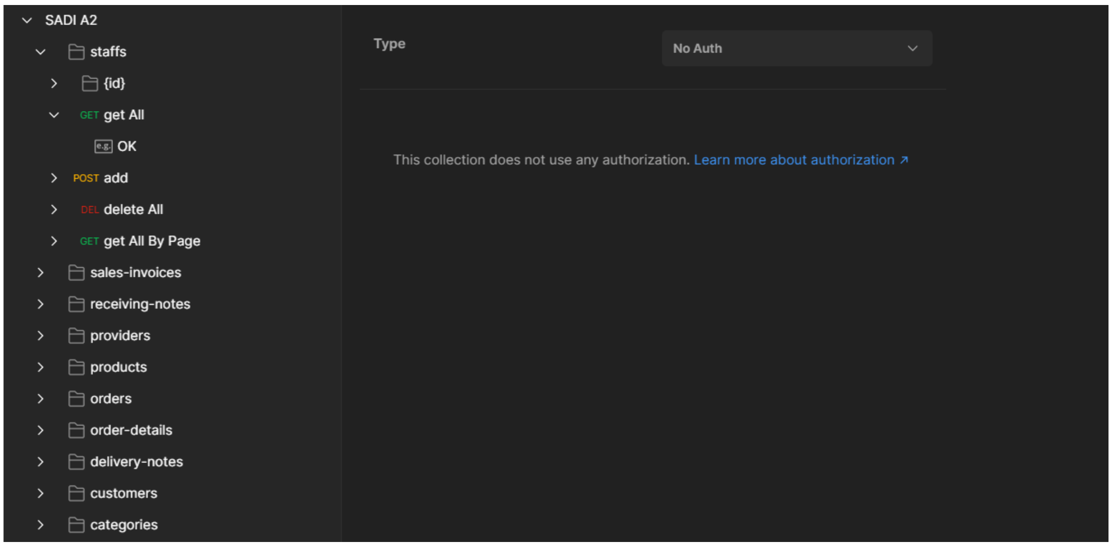
Instead of choosing to work on pure SQL queries, we pick Hibernate as our ORM (Object-Relational Mapping) framework. Hibernate uses Java Database Connectivity (JDBC), therefore, the configuration of PostgreSQL access would be easier. Hibernate also provides a strategy for constructing entities with complex schemas through the use of DI and IoC.

4. **Spring Web MVC**

As our objective in this project is to architect a backend application with the team of two, we must consider using a technology that can follow the concept of Separation of Concerns (SoC). Therefore, Spring MVC can satisfy our requirements by its Model-View-Controller (MVC) architecture. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

5. **Postman**

A collaboration platform for API development. We use Postman to set up a workspace for team members to test and run the API on the platform. Furthermore, we also import the OpenAPI specification of our API design into Postman for fast generation of API documentation.



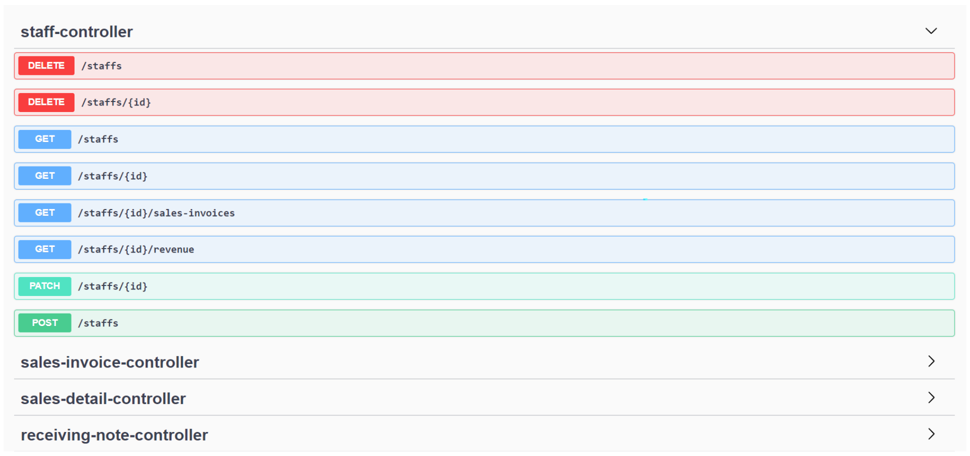
SADI A2
- staffs
  - {id}
  - GET get All
    - OK
  - POST add
  - DEL delete All
  - GET get All By Page
- sales-invoices
- receiving-notes
- providers
- products
- orders
- order-details
- delivery-notes
- customers
- categories

Type: No Auth

This collection does not use any authorization. Learn more about authorization ↗

6. **Swagger & OpenAPI**

When developing a REST API, it is quite hard to keep both team members up to date with every update the other makes. We came up with using Swagger as our API documentation generator to make things easier. Spring also has a package to support the integration of Swagger. To test an instant API request, we can also use Swagger playground.


staff-controller

DELETE /staffs

DELETE /staffs/{id}

GET /staffs

GET /staffs/{id}

GET /staffs/{id}/sales-invoices

GET /staffs/{id}/revenue

PATCH /staffs/{id}

POST /staffs

sales-invoice-controller

sales-detail-controller

receiving-note-controller

We also use OpenAPI Specification as a machine-readable interface file to demonstrate and visualize our REST API. Hence, we import this file into Postman to generate an API schema.

### 7. MockMVC, Mockito and JUnit 5

We use MockMVC to test the web layer of our Spring application. It provides an API to test HTTP requests directly on the testing environment. Mockito is a dependency used for wiring and mocking the data access layer to avoid mutating the data inside a production database.

## V. Architecture

### 1. Model-View-Controller

As we have researched, MVC or Model-View-Controller is a very popular design pattern. It follows the Separation of Concerns principle. MVC pattern separates the presentation and data processing from logic and interface. Each component in the MVC pattern takes a distinctive responsibility. While the model takes an account for data management and logic handling, the controller, as an input handler, will operate the commands for model and view. Regarding our application's requirements, we do not have a user interface, or we can say, a view.

### 2. Repository pattern

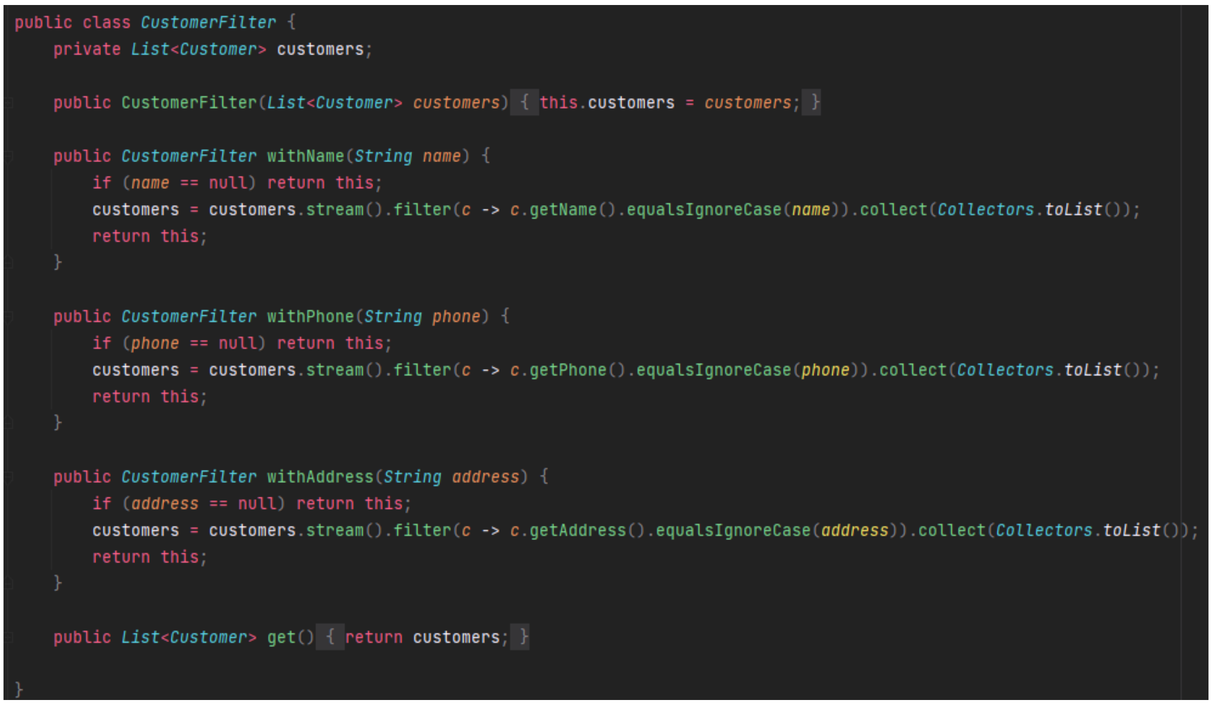
Implementing the repository pattern will create an abstraction layer between the data access layer and the business logic layer. Produced outcomes from this implementation include prompting a more loosely coupled approach to data access. Within our application, we use JPA Repository to construct the repository abstraction layer and implement a "One-per business model". To elaborate more about the applied model, each entity is mapped with a specific repository. Hence, instead of interacting with Entity Framework, controllers interact with the repository.

## 3. Service layer

A service layer is an additional layer in a MVC architecture which complements the business logic layer. The service is a communication method for controllers and repositories. It contains business logic and validation logic. The service layer bridges the presentation layer with the data access layer.

## 4. Builder pattern

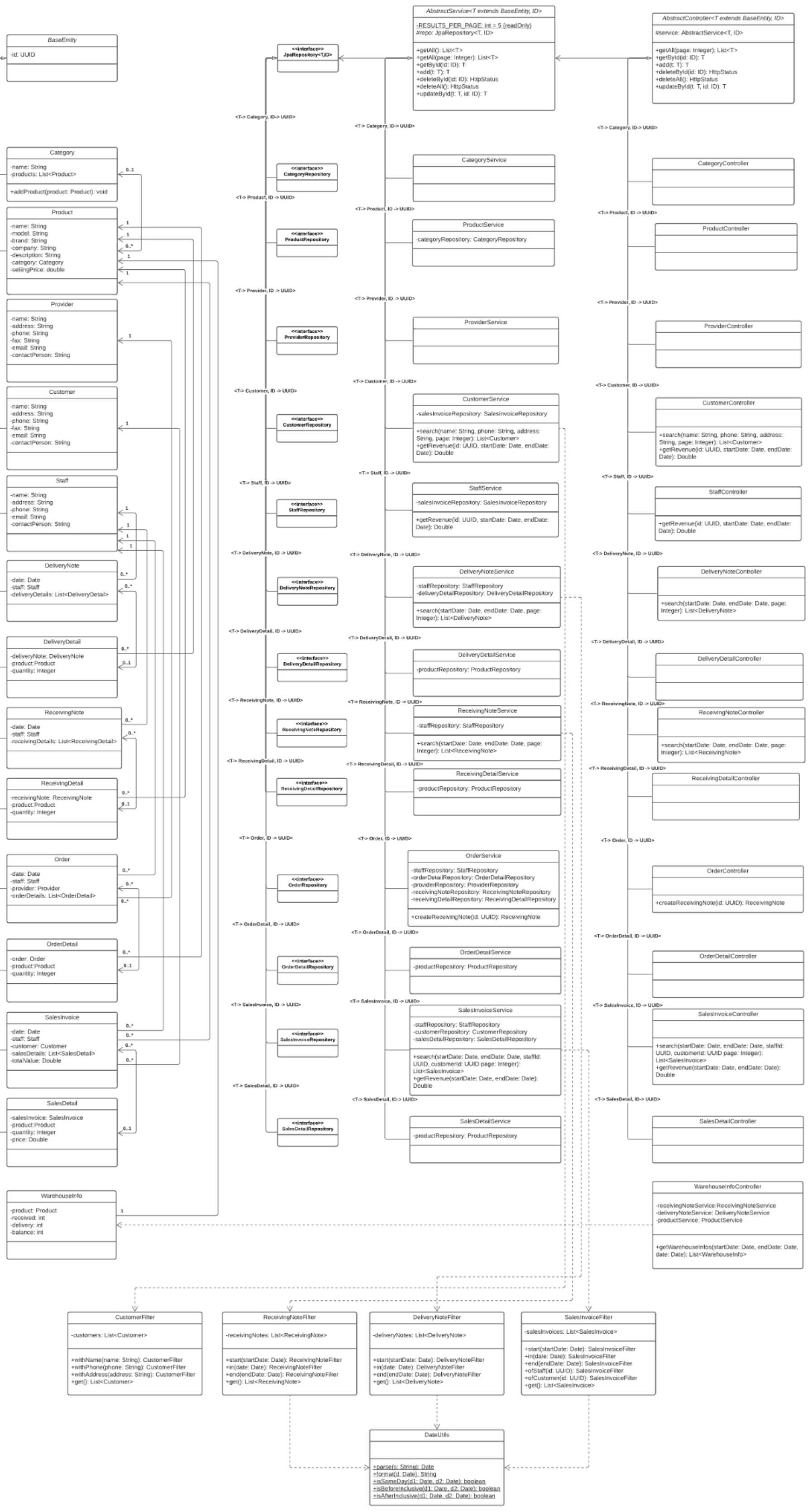
About the definition of the pattern: “Builder Pattern is a creational design pattern that separates the construction of a complex object from its representation so that the same construction process can create different representations”. Understanding the definition of it, we applied to our application to construct a filter machine for the search API. As search API receives many optional parameters, there would be code duplication if we try to build the functions for all combinations of parameters. To avoid the problem, we use a filter machine made with a builder pattern to find a separate data then filtering them as a whole.



```java
public class CustomerFilter {
    private List<Customer> customers;

    public CustomerFilter(List<Customer> customers) { this.customers = customers; }

    public CustomerFilter withName(String name) {
        if (name == null) return this;
        customers = customers.stream().filter(c -> c.getName().equalsIgnoreCase(name)).collect(Collectors.toList());
        return this;
    }

    public CustomerFilter withPhone(String phone) {
        if (phone == null) return this;
        customers = customers.stream().filter(c -> c.getPhone().equalsIgnoreCase(phone)).collect(Collectors.toList());
        return this;
    }

    public CustomerFilter withAddress(String address) {
        if (address == null) return this;
        customers = customers.stream().filter(c -> c.getAddress().equalsIgnoreCase(address)).collect(Collectors.toList());
        return this;
    }

    public List<Customer> get() { return customers; }

}
```

An example of the Customer Filter class contains several methods to filter with customer name, phone number or address. If we want to filter it with all requirements, we can call all methods to handle the operation.

## VI. Class diagram



BaseEntity
-id: UUID
Category
-name: String
-products: List<Product>
+addProduct(product: Product): void
Product
-name: String
-model: String
-brand: String
-company: String
-description: String
-category: Category
-sellingPrice: double
Provider
-name: String
-address: String
-phone: String
-fax: String
-email: String
-contactPerson: String
Customer
-name: String
-address: String
-phone: String
-fax: String
-email: String
-contactPerson: String
Staff
-name: String
-address: String
-phone: String
-email: String
-contactPerson: String
DeliveryNote
-date: Date
-staff: Staff
-deliveryDetails: List<DeliveryDetail>
DeliveryDetail
-deliveryNote: DeliveryNote
-product:Product
-quantity: Integer
ReceivingNote
-date: Date
-staff: Staff
-receivingDetails: List<ReceivingDetail>
ReceivingDetail
-receivingNote: ReceivingNote
-product:Product
-quantity: Integer
Order
-date: Date
-staff: Staff
-provider: Provider
-orderDetails: List<OrderDetail>
OrderDetail
-order: Order
-product:Product
-quantity: Integer
SalesInvoice
-date: Date
-staff: Staff
-customer: Customer
-salesDetails: List<SalesDetail>
-totalValue: Double
SalesDetail
-salesInvoice: SalesInvoice
-product:Product
-quantity: Integer
-price: Double
WarehouseInfo
-product: Product
-received: int
-delivery: int
-balance: int
0..1
0..*
1
<<interface>> JpaRepository<T,ID>
<<interface>> CategoryRepository
<<interface>> ProductRepository
<<interface>> ProviderRepository
<<interface>> CustomerRepository
<<interface>> StaffRepository
<<interface>> DeliveryNoteRepository
<<interface>> DeliveryDetailRepository
<<interface>> ReceivingNoteRepository
<<interface>> ReceivingDetailRepository
<<interface>> OrderRepository
<<interface>> OrderDetailRepository
<<interface>> SalesInvoiceRepository
<<interface>> SalesDetailRepository
<T-> Category, ID-> UUID>
<T-> Product, ID -> UUID>
<T-> Provider, ID -> UUID>
<T-> Customer, ID -> UUID>
<T-> Staff, ID -> UUID>
<T-> DeliveryNote, ID -> UUID>
<T-> DeliveryDetail, ID -> UUID>
<T-> ReceivingNote, ID -> UUID>
<T-> ReceivingDetail, ID -> UUID>
<T-> Order, ID -> UUID>
<T-> OrderDetail, ID -> UUID>
<T-> SalesInvoice, ID -> UUID>
<T-> SalesDetail, ID -> UUID>
AbstractService<T extends BaseEntity, ID>
-RESULTS_PER_PAGE: int = 5 {readOnly}
#repo: JpaRepository<T, ID>
+getAll(): List<T>
+getAll(page: Integer): List<T>
+getById(id: ID): T
+add(t: T): T
+deleteById(id: ID): HttpStatus
+deleteAll(): HttpStatus
+updateById(t: T, id: ID): T
CategoryService
ProductService
-categoryRepository: CategoryRepository
ProviderService
CustomerService
-salesInvoiceRepository: SalesInvoiceRepository
+search(name: String, phone: String, address: String, page: Integer): List<Customer>
+getRevenue(id: UUID, startDate: Date, endDate: Date): Double
StaffService
-salesInvoiceRepository: SalesInvoiceRepository
+getRevenue(id: UUID, startDate: Date, endDate: Date): Double
DeliveryNoteService
-staffRepository: StaffRepository
-deliveryDetailRepository: DeliveryDetailRepository
+search(startDate: Date, endDate: Date, page: Integer): List<DeliveryNote>
DeliveryDetailService
-productRepository: ProductRepository
ReceivingNoteService
-staffRepository: StaffRepository
+search(startDate: Date, endDate: Date, page: Integer): List<ReceivingNote>
ReceivingDetailService
-productRepository: ProductRepository
OrderService
-staffRepository: StaffRepository
-orderDetailRepository: OrderDetailRepository
-providerRepository: ProviderRepository
-receivingNoteRepository: ReceivingNoteRepository
-receivingDetailRepository: ReceivingDetailRepository
+createReceivingNote(id: UUID): ReceivingNote
OrderDetailService
-productRepository: ProductRepository
SalesInvoiceService
-staffRepository: StaffRepository
-customerRepository: CustomerRepository
-salesDetailRepository: SalesDetailRepository
+search(startDate: Date, endDate: Date, staffId: UUID, customerId: UUID page: Integer): List<SalesInvoice>
+getRevenue(startDate: Date, endDate: Date): Double
SalesDetailService
-productRepository: ProductRepository
AbstractController<T extends BaseEntity, ID>
#service: AbstractService<T, ID>
+getAll(page: Integer): List<T>
+getById(id: ID): T
+add(t: T): T
+deleteById(id: ID): HttpStatus
+deleteAll(): HttpStatus
+updateById(t: T, id: ID): T
CategoryController
ProductController
ProviderController
CustomerController
+search(name: String, phone: String, address: String, page: Integer): List<Customer>
+getRevenue(id: UUID, startDate: Date, endDate: Date): Double
StaffController
+getRevenue(id: UUID, startDate: Date, endDate: Date): Double
DeliveryNoteController
+search(startDate: Date, endDate: Date, page: Integer): List<DeliveryNote>
DeliveryDetailController
ReceivingNoteController
+search(startDate: Date, endDate: Date, page: Integer): List<ReceivingNote>
ReceivingDetailController
OrderController
+createReceivingNote(id: UUID): ReceivingNote
OrderDetailController
SalesInvoiceController
+search(startDate: Date, endDate: Date, staffId: UUID, customerId: UUID page: Integer): List<SalesInvoice>
+getRevenue(startDate: Date, endDate: Date): Double
SalesDetailController
WarehouseInfoController
-receivingNoteService:ReceivingNoteService
-deliveryNoteService: DeliveryNoteService
-productService: ProductService
+getWarehouseInfos(startDate: Date, endDate: Date, date: Date): List<WarehouseInfo>
CustomerFilter
-customers: List<Customer>
+withName(name: String): CustomerFilter
+withPhone(phone: String): CustomerFilter
+withAddress(address: String): CustomerFilter
+get(): List<Customer>
ReceivingNoteFilter
-receivingNotes: List<ReceivingNote>
+start(startDate: Date): ReceivingNoteFilter
+in(date: Date): ReceivingNoteFilter
+end(endDate: Date): ReceivingNoteFilter
+get(): List<ReceivingNote>
DeliveryNoteFilter
-deliveryNotes: List<DeliveryNote>
+start(startDate: Date): DeliveryNoteFilter
+in(date: Date): DeliveryNoteFilter
+end(endDate: Date): DeliveryNoteFilter
+get(): List<DeliveryNote>
SalesInvoiceFilter
-salesInvoices: List<SalesInvoice>
+start(startDate: Date): SalesInvoiceFilter
+in(date: Date): SalesInvoiceFilter
+end(endDate: Date): SalesInvoiceFilter
+ofStaff(id: UUID): SalesInvoiceFilter
+ofCustomer(id: UUID): SalesInvoiceFilter
+get(): List<SalesInvoice>
DateUtils
+parse(s: String): Date
+format(d: Date): String
+isSameDay(d1: Date, d2: Date): boolean
+isBeforeInclusive(d1: Date, d2: Date): boolean
+isAfterInclusive(d1: Date, d2: Date): boolean

*(view the ClassDiagram.pdf file if this is not clear enough)*

## VII. Implementation result

### 1. What we have done

Based on the given business requirements of the assignment, we have covered all the operations including the CRUD (Create-Read-Update-Delete) operations for required entities. Extending from that, we also implemented some additional API for non-required entities. The motivation to cover all CRUD operations is for testing and populating data. All implementations include:

- Feature 1: Create Read Update Delete (CRUD API)
- Feature 2: Search API & Pagination
- Feature 3: Additional REST APIs
    - List all inventory note, sale invoice by a period: start date and end date
    - Get all sales invoices by a customer and by a sales staff in a period.
    - Calculate revenue, revenue by a customer, revenue by a sales staff
    - Get inventory of all products in the warehouse at a particular date.
- Unit testing & Integration testing

Which can be demonstrated through the table below

| **Controller** | **Application Programming Interface (API)** | | | | **Testing** | |
|---|---|---|---|---|---|---|
| | **CRUD** | **Search** | **Other** | **Paging** | **Unit** | **Integration** |
| Category | Yes | No | No | Yes | No | No |
| Customer | Yes | Yes | getRevenue | Yes | Yes | Yes |
| DeliveryDetail | Yes | No | No | Yes | No | No |
| DeliveryNote | Yes | Yes | No | Yes | Yes | Yes |
| Order | Yes | Yes | createReceivingNote | Yes | Yes | Yes |
| OrderDetail | Yes | No | No | Yes | No | No |

| | | | | | | |
|---|---|---|---|---|---|---|
| Product | Yes | No | No | Yes | No | No |
| Provider | Yes | No | No | Yes | No | No |
| ReceivingDetail | Yes | No | No | Yes | No | No |
| ReceivingNote | Yes | Yes | No | Yes | Yes | Yes |
| SalesDetail | Yes | No | No | Yes | No | No |
| SalesInvoice | Yes | Yes | getRevenue | Yes | Yes | Yes |
| Staff | Yes | Yes | getRevenue | Yes | Yes | Yes |
| WarehouseInfo | No | No | getWarehouseInfo | No | Yes | Yes |

## 2. How we implement

To implement CRUD operations, we use REST API to call HTTP requests to the Tomcat server. The HTTP request that we use for our operations ruled as:

| **HTTP method** | **Usage** |
|---|---|
| POST | Create a new entity |
| GET | Get an entity by id, get all entities, search API, getRevenue, getWarehouseInfo |
| DELETE | Delete all entities, delete an entity by id |
| PATCH | Update an entity by id with declared fields |

We try to take advantage of abstract classes, especially AbstractService and AbstractController. The benefit we take from this approach is that we can reduce much time on writing duplicated code. To avoid a high coupling between inherited classes and its predecessor, we only cover basic operations. In case the class needs to expand its functionality, it can override or just add another method inside an inherited child. The same context is also applied for testing. We use AbstractUnitTest to cover all bases, if there are extended test cases, we can implement it inside a child class.

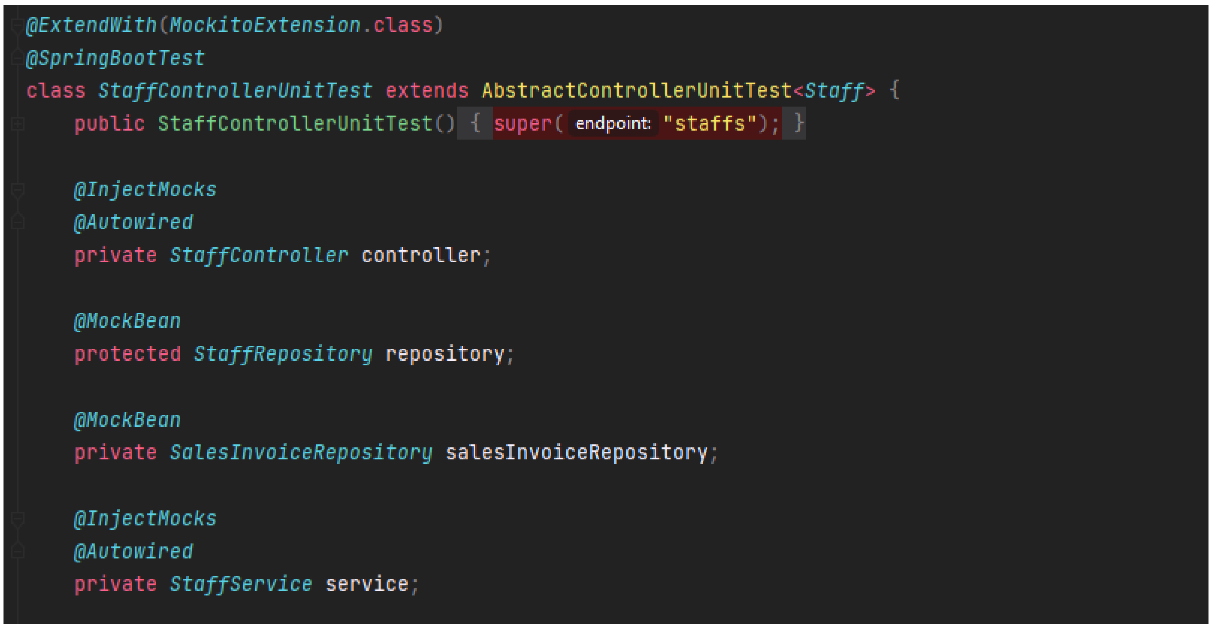
For the search API, to list all inventory notes and get inventory of all products are also categorized as search API. The search API takes input(s) as HTTP request parameters. Then, we filter data based on the given parameters with the filter machine constructed with the builder pattern.

To implement update methods, we use a PATCH method to handle the update operation. Unlike PUT, PATCH applies a partial update to the resource. Only a given data is updated, not a whole entity. Take the update operation of the Staff entity as an example, if we need to update a name of the staff, we only need to give a staff name inside a request body. This looks like a structure of the POST methods to create a new entity, however, for the POST methods, some fields are required and for the PATCH methods, all fields are optional.

## VIII. Testing

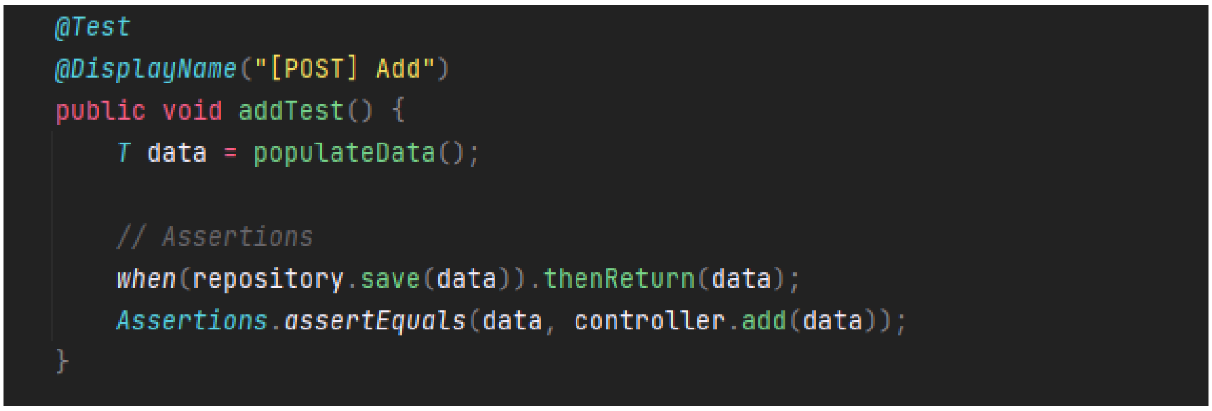
### 1. What we have done

As we have mentioned before, JUnit 5, Mockito and MockMVC are two dependencies that we used for unit testing and integration testing. Our testing process consists of two stages, the first one is unit testing and the other one is integration testing. For unit testing, we test the business logic layer, or we can say, the Service. We divide the test cases into classes, each test class depends on the setup controller which is constructed with standaloneSetup function from Mockito. By directing to the tested Controller, its related Services and Repositories are wired through the annotations called @MockBean and @Autowired. The purpose of all of this configuration is to test the logic without affecting the production database.



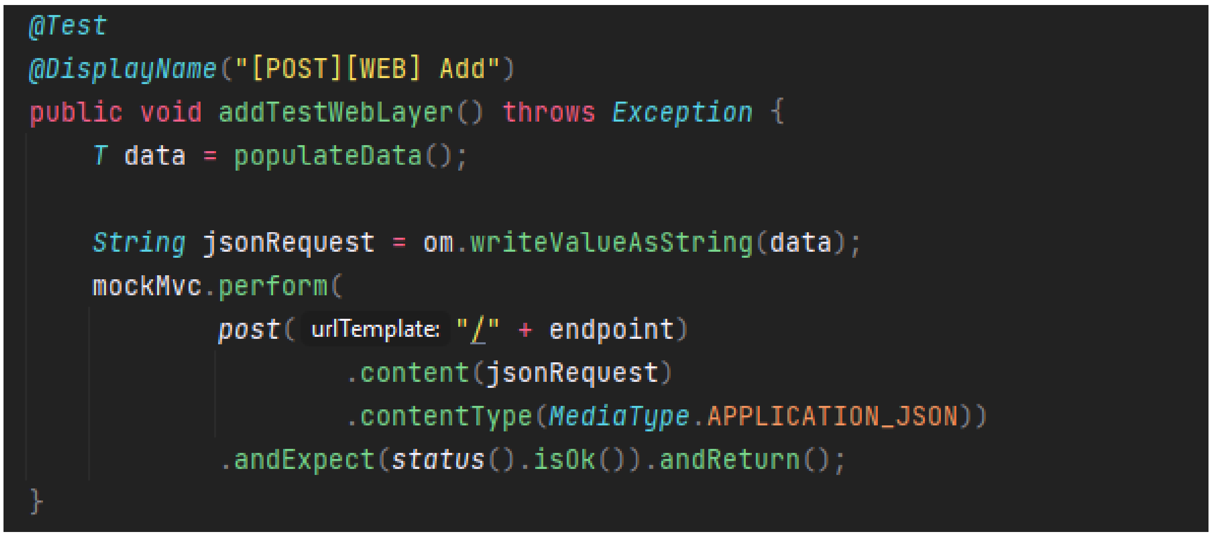
```
@ExtendWith(MockitoExtension.class)
@SpringBootTest
class StaffControllerUnitTest extends AbstractControllerUnitTest<Staff> {
    public StaffControllerUnitTest() { super( endpoint: "staffs"); }

    @InjectMocks
    @Autowired
    private StaffController controller;

    @MockBean
    protected StaffRepository repository;

    @MockBean
    private SalesInvoiceRepository salesInvoiceRepository;

    @InjectMocks
    @Autowired
    private StaffService service;
```

After finishing the configuration, we implemented the test cases by mocking the Repositories and asserting the Controller methods. Here is an example of the POST method unit test:



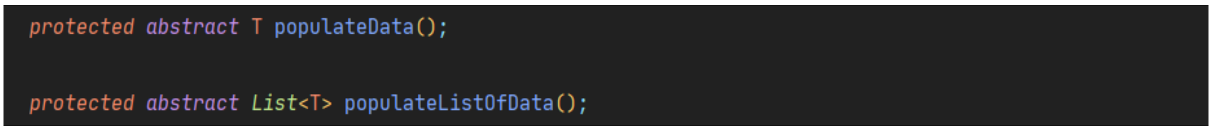
```java
@Test
@DisplayName("[POST] Add")
public void addTest() {
    T data = populateData();

    // Assertions
    when(repository.save(data)).thenReturn(data);
    Assertions.assertEquals(data, controller.add(data));
}
```

That is a structure for business logic layer test cases, for the presentation layer, we use MockMVC to send a HTTP request to a server and test the response of it. Here is an example of the POST method integration test:



```java
@Test
@DisplayName("[POST][WEB] Add")
public void addTestWebLayer() throws Exception {
    T data = populateData();

    String jsonRequest = om.writeValueAsString(data);
    mockMvc.perform(
            post( urlTemplate: "/" + endpoint)
                    .content(jsonRequest)
                    .contentType(MediaType.APPLICATION_JSON))
            .andExpect(status().isOk()).andReturn();
}
```

The primary pattern that we used to construct test cases is the Template Method pattern. Most of the test cases for CRUD operation are duplicated, therefore, we have a class called AbstractControllerUnitTest to cover all test cases for CRUD operation. This abstract class will have two abstract methods called populateData() and populateListOfData().

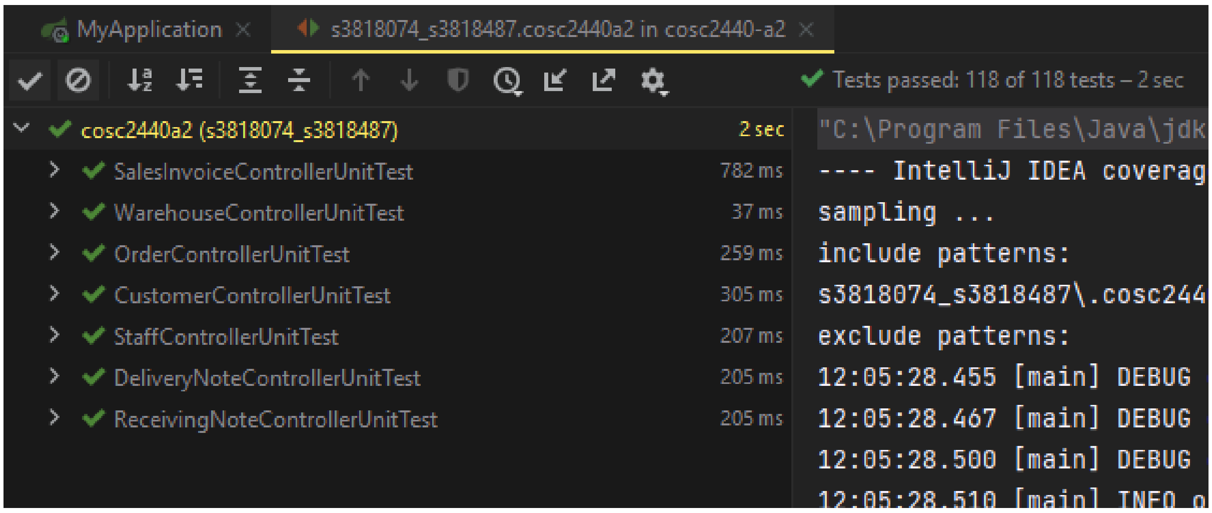


```java
protected abstract T populateData();

protected abstract List<T> populateListOfData();
```

The test class which inherits from the AbstractControllerUnitTest class will populate the specific data through implementing the two abstract methods. This implementation sounds complicated but it reduces a lot of time from writing duplicated test cases.

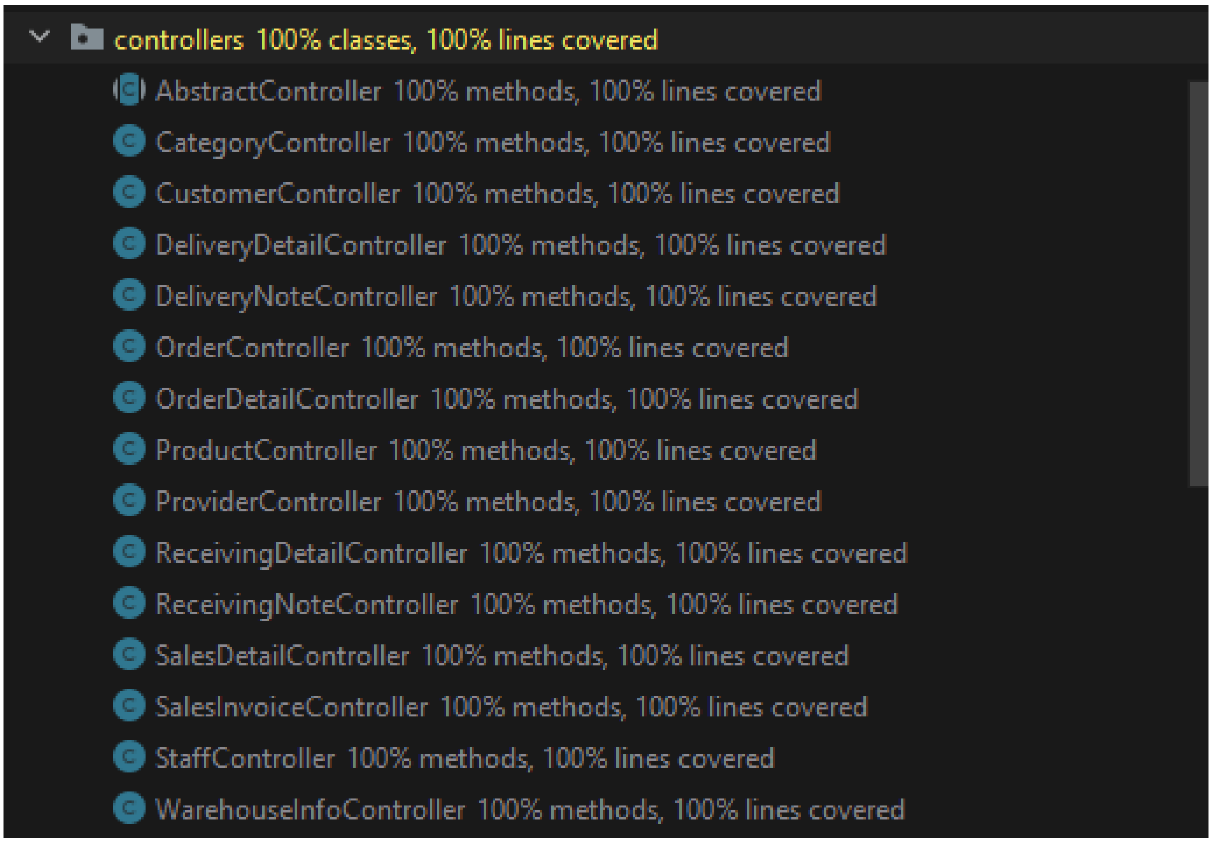
## 2. Results

In the end, to summarize the test cases we covered, there are 118 successful test cases, including basic CRUD operations from AbstractControllerUnitTest, search and additional API from concrete test classes. We also cover cases such as trying to insert a non-exist foreign key or missing required fields in the request body. We assure the certainty of all the test cases with 100% coverage.



```
Tests passed: 118 of 118 tests – 2 sec
cosc2440a2 (s3818074_s3818487)      2 sec
  SalesInvoiceControllerUnitTest    782 ms
  WarehouseControllerUnitTest        37 ms
  OrderControllerUnitTest           259 ms
  CustomerControllerUnitTest        305 ms
  StaffControllerUnitTest           207 ms
  DeliveryNoteControllerUnitTest    205 ms
  ReceivingNoteControllerUnitTest   205 ms
```

```
"C:\Program Files\Java\jdk
---- IntelliJ IDEA coverag
sampling ...
include patterns:
s3818074_s3818487\.cosc244
exclude patterns:
12:05:28.455 [main] DEBUG
12:05:28.467 [main] DEBUG
12:05:28.500 [main] DEBUG
12:05:28.510 [main] INFO o
```



```
controllers  100% classes, 100% lines covered
    AbstractController  100% methods, 100% lines covered
    CategoryController  100% methods, 100% lines covered
    CustomerController  100% methods, 100% lines covered
    DeliveryDetailController  100% methods, 100% lines covered
    DeliveryNoteController  100% methods, 100% lines covered
    OrderController  100% methods, 100% lines covered
    OrderDetailController  100% methods, 100% lines covered
    ProductController  100% methods, 100% lines covered
    ProviderController  100% methods, 100% lines covered
    ReceivingDetailController  100% methods, 100% lines covered
    ReceivingNoteController  100% methods, 100% lines covered
    SalesDetailController  100% methods, 100% lines covered
    SalesInvoiceController  100% methods, 100% lines covered
    StaffController  100% methods, 100% lines covered
    WarehouseInfoController  100% methods, 100% lines covered
```

## IX. Limitation, known bugs

### 1. No validation of insignificant fields

Since we only focus on building the API with the relationship of the entities involved in a trade company, we only validate the integrity of those relationships to make the data consistent and reliable. For example, a sales invoice must have a customer and a staff; and their IDs provided in the request must exist in the database; the sales details must not belong to any other sales invoices before being added to the requested sales invoice.
However, we do not validate insignificant fields such as name, email, phone number, etc. The word ‘insignificant’ here states that the fields have nothing to do with the relationship between these entities, so they do not break data integrity in any cases. For example, in a real world case, the email of a customer must be unique in the database and follow a specific format. However, we leave it open and let the company decide later. Having a null email or invalid email format will not break our system so we do not need to put constraints on the email or any other insignificant fields right away.

### 2. No update methods on details

The API does not provide update methods on details such as sales details. This prevents a sales invoice from being modified unintentionally by modifying the sales details. However, the trade off is that it takes more effort to update a sales invoice. If we want to modify the quantity of a sales detail, we need to delete the old one, create a new one with the same product and modified quantity, then update the sales invoice’ details with the new one and current details, which may be troublesome for the company. The same happens to receiving notes, delivery notes and orders.

## X. Notes

Read the README.md file for notes on how to use our API.