DEA II – GROUP ASSESSMENT

GROUP 15

Student ID	Student Name
20841	IU Ratnayake
20760	NMN Jeewanthi
20794	WSWMI Wimukthi
20786	HLN Sewwandi
20836	VPYC Perera

APRIL 27, 2024

NSBM GREEN UNIVERSITY

Contents

Introduction

For this project we have decided to develop a prototype for a Supply Chain Management System for Lahiru Auto International. The system must have facilities to edit, delete, update and view users, customers, and products. It must also be able to add, view and delete two types of invoices, one for customers and one for suppliers. Each invoice can have many products.

Team Overview

Member contribution

Student ID	Student Name	Contribution
20841	IU Ratnayake	Administration service, Security for all services
20760	NMN Jeewanthi	Supplier Service
20794	WSWMI Wimukthi	Products Service
20786	HLN Sewwandi	Customer Service
20836	VPYC Perera	Invoice Service

Challenges Faced and how they were overcome

Teams had disparities in technical skills required for the project.

Through numerous group discussions and KT sessions we managed to update each other's knowledge to the level required for the project.

Asynchronous calls were not suitable for user authentication.

Through self-study and research, synchronous calls via RestTemplate were implemented.

Architecture

For the backend a Micro-service architecture was used. The backend is composed of five microservices: Administration, Suppliers, Customers, Products and Invoices, which are independently deployable and loosely coupled.

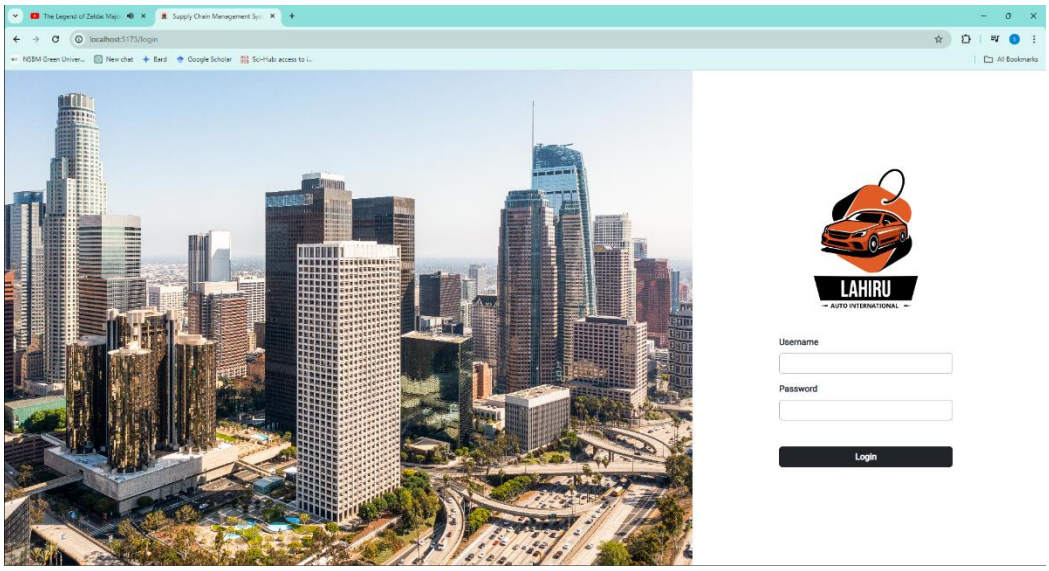
For the front end an MVP architecture was used. The front end was built using React and React Router. The entire system was designed using the Rest Architectural pattern.

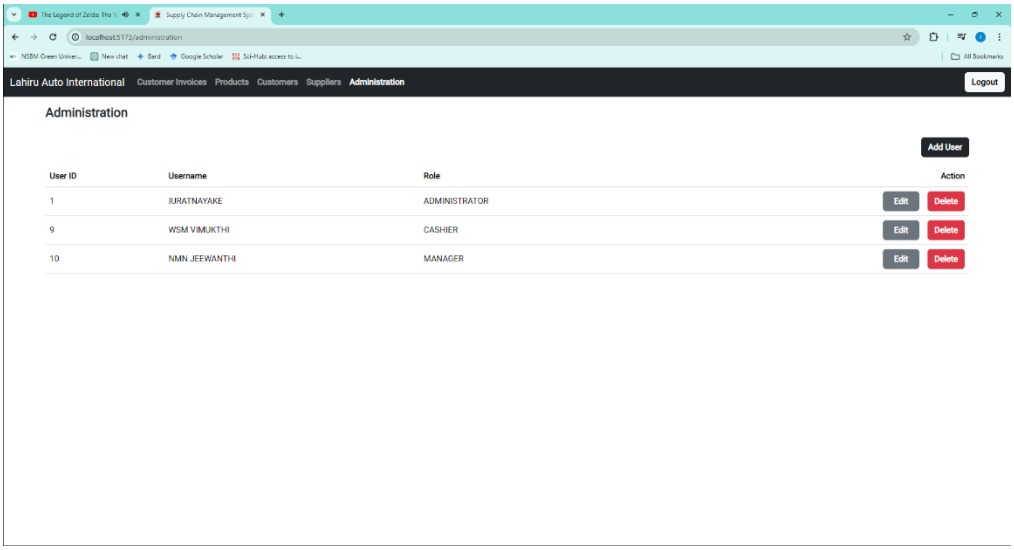
Microservices

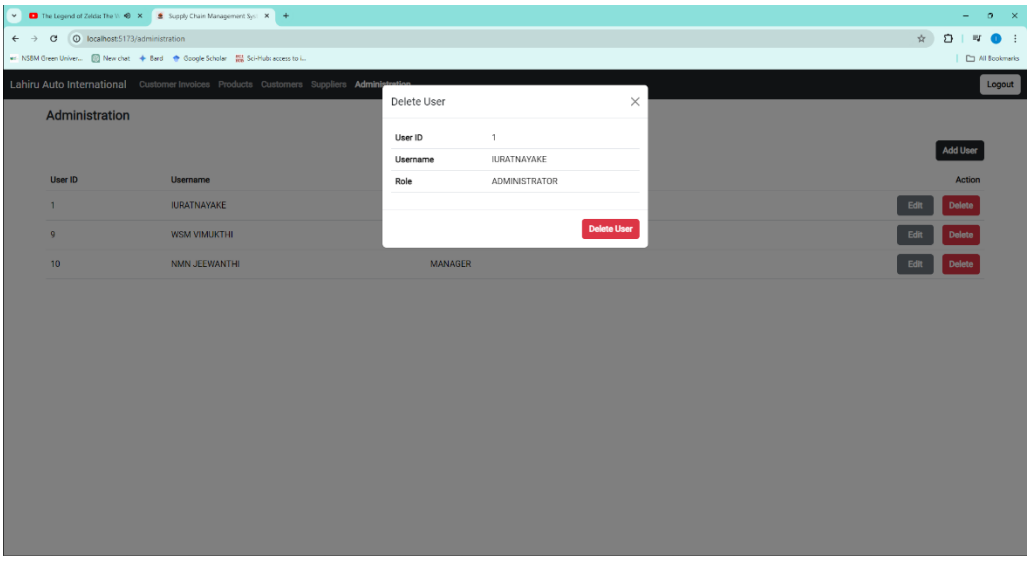
Administration

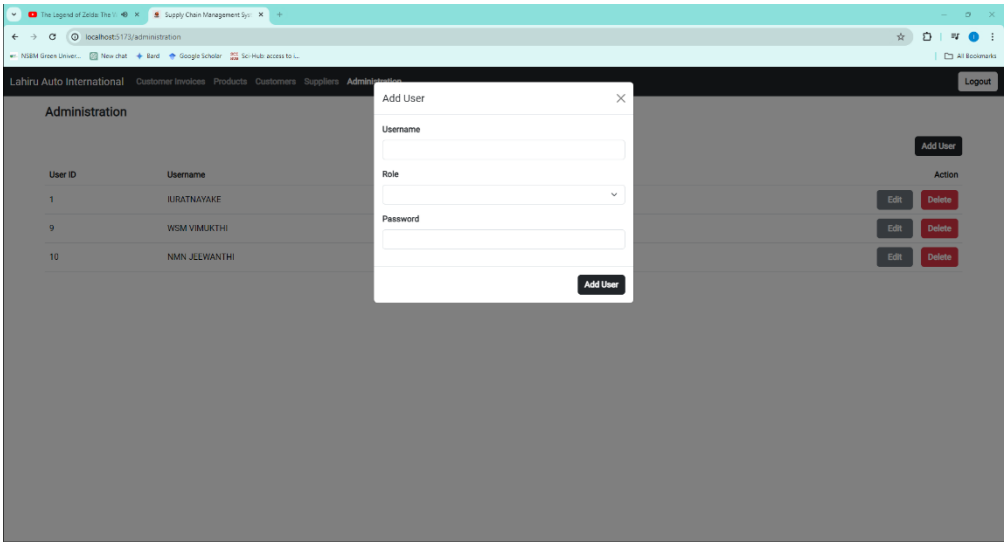
The Administration microservice is responsible for Authentication, Authorization and User Management. It provides functionality for login users into the system, checking if the user is currently logged in, authorization of actions, adding new users, deleting, and updating users and fetching user details.

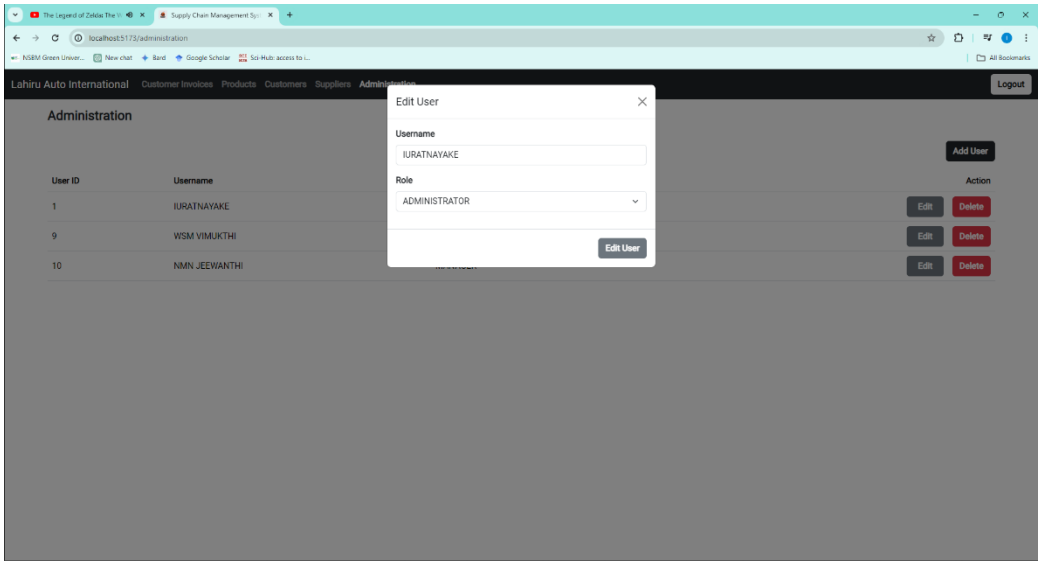
Spring Web has been used for the development of the microservice via implementation of the REST architectural pattern. Spring Data JPA has been used for database interfacing. Spring Security has been implemented for security functions of the application including authentication and authorization. IT also uses Lombok to minimize boilerplate code use.





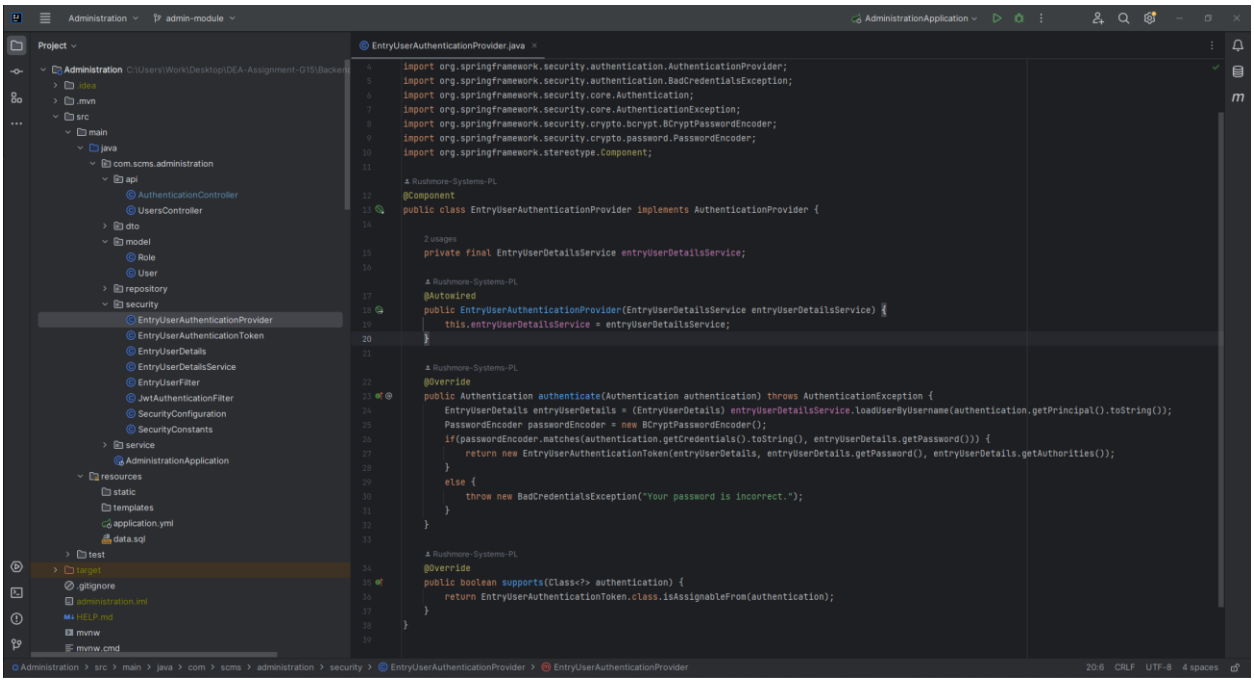

Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Administration
Logout
Administration
Add User
User ID
Username
Action
1
IURATNAYAKE
Edit
Delete
9
WSM VIMUKTHI
Edit
Delete
10
NMN JEEWANTHI
MANAGER
Edit
Delete
Delete User
User ID
1
Username
IURATNAYAKE
Role
ADMINISTRATOR
Delete User

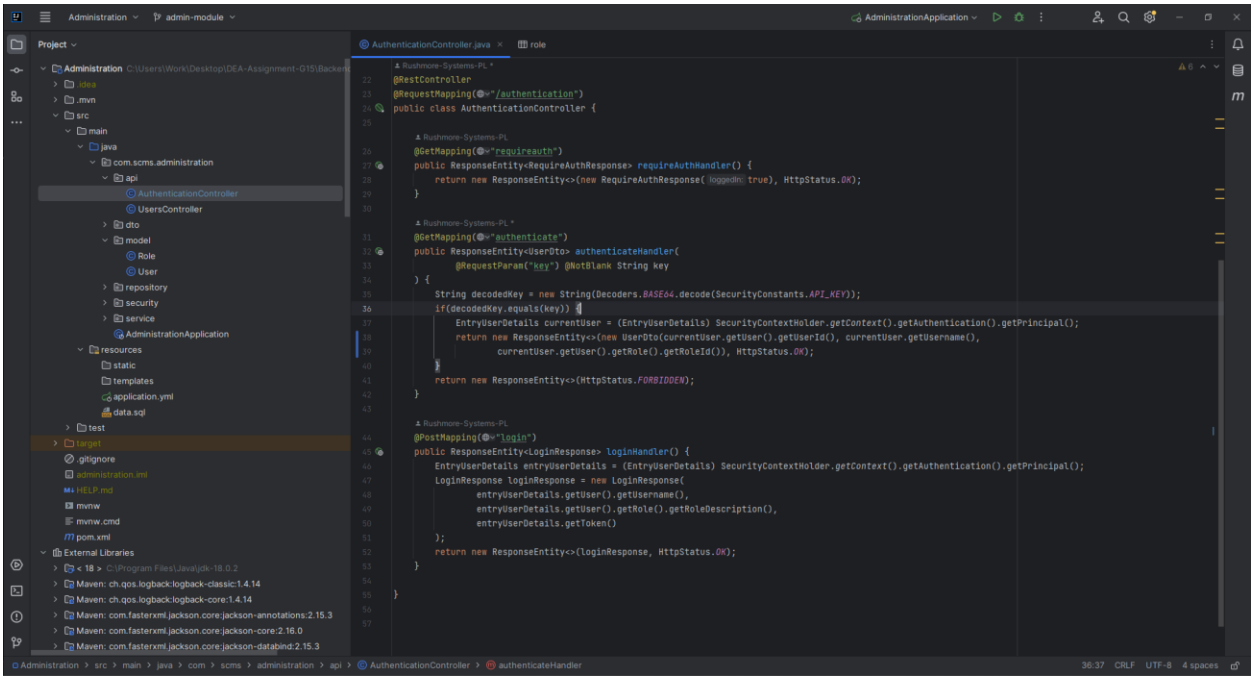

Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Administration
Logout
Administration
Add User
User ID
Username
Action
1
IURATNAYAKE
Edit
Delete
9
WSM VIMUKTHI
Edit
Delete
10
NMN JEEWANTHI
Edit
Delete
Add User
Username
Role
Password
Add User

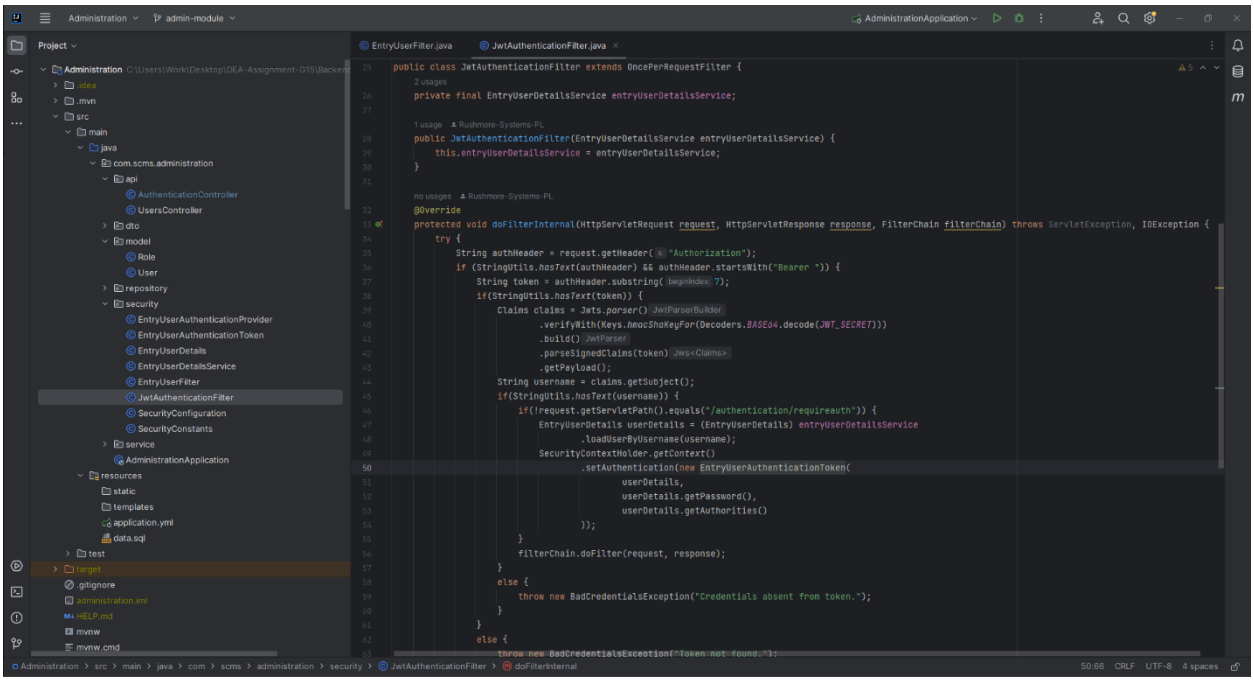

Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Administration
Logout
Administration
Add User
User ID
Username
Action
1
IURATNAYAKE
Edit
Delete
9
WSM VIMUKTHI
Edit
Delete
10
NMN JEEWANTHI
Edit
Delete
Edit User
Username
IURATNAYAKE
Role
ADMINISTRATOR
Edit User

Authentication

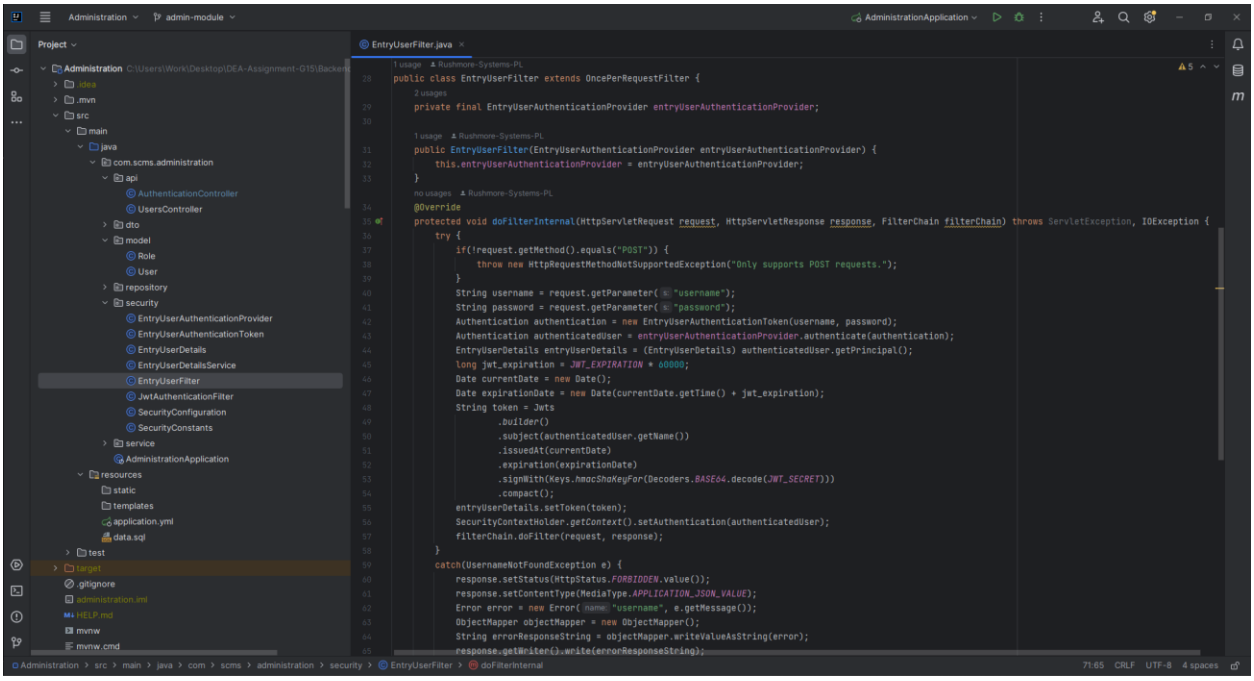
Authentication is the process of identifying a user. A user is authenticated by a combination of a username and password. This will be provided by the microservice by exposing a login endpoint in the AuthenticationController class.







An EntryUserFilter class, which is an extension of OncePerRequestFilter, intercepts the request in order to perform authentication on the request. An EntryUserAuthenticationProvider class is responsible for implementation of the authentication logic. It produces an EntryUserAuthenticationToken (which extends the Authentication class) which is then stored in the



Spring Security Context. During the above process if the authentication fails the request is rejected with an Http status code of FORBIDDEN.

Once authentication is successful, a unique Java JWT access token is generated and added to the Security Context. For all other requests made to the Administration microservice, the request will be check for a JJWT token that is sent using the Bearer Scheme. In addition to authentication, in a

stateless architectural pattern (REST), the user of JJWT tokens also provides CSFR protection. (Default protection is disabled)

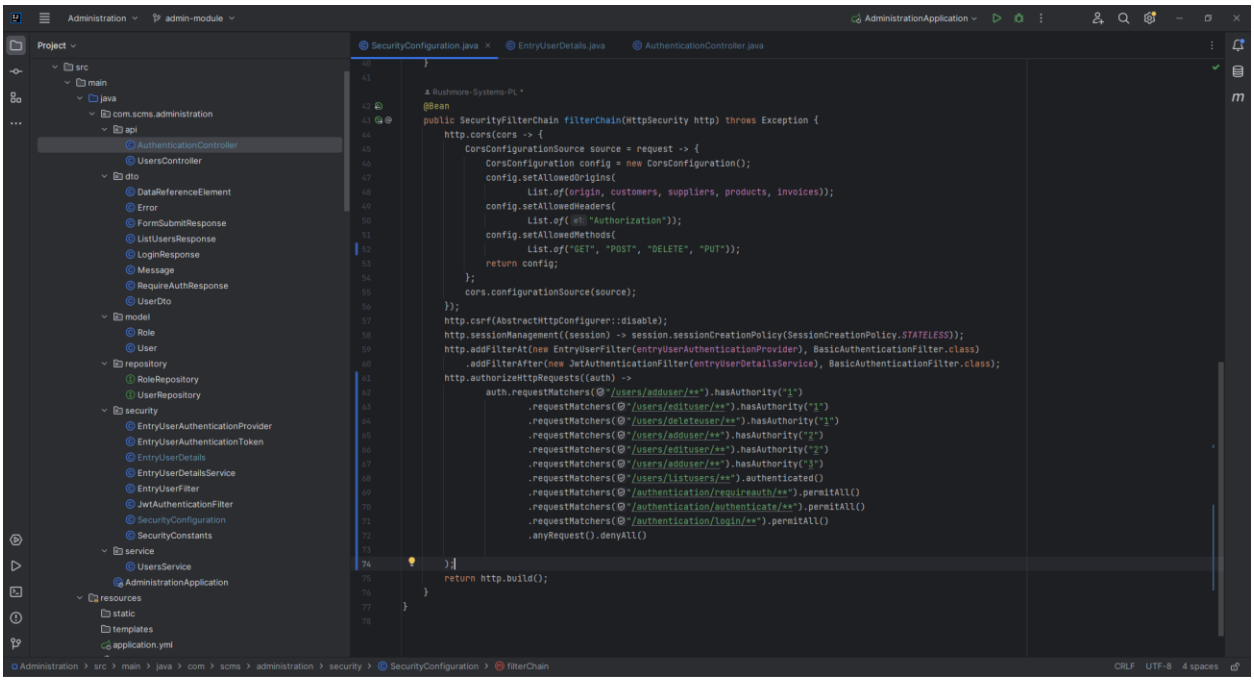
Authentication for other Microservices

The Administration module also facilitates authentication for other services of the application. This is done by a synchronous request sent to the application via a RestTemplate instance. The request will contain the Authorization header sent to the service. The request is sent to the authenticate endpoint of the AuthenticationController. After authentication, a UserDto object is sent to the calling service with information about the logging in user, to be stored in their SecurityContext.

Authorization

Sprin security is used to control end point access based on user role. The system provides three roles Administrator (1), Manager (2) and Cashier (3). This access is controlled via the SecuiryConfig class which is a @Configuration type component.

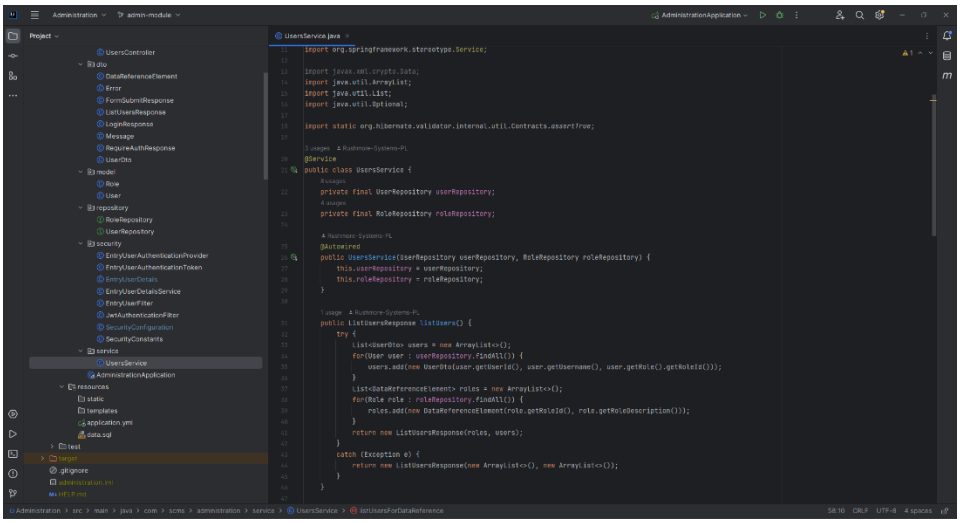
Based on the UserDto object received by other Services, authorization is handled for the other services as well.



User Management

User management facilitates CRUD operations on the User entity. It is composed of the UserController, UserService, UserRepository and User classes. In addition, Role, RoleRepository classes exist to supplement the functionality.

The User Controller exposes endpoints for User management. These include adding a user, editing a user, deleting a user and retrieving list of all users in the system. Add a user is done via a post

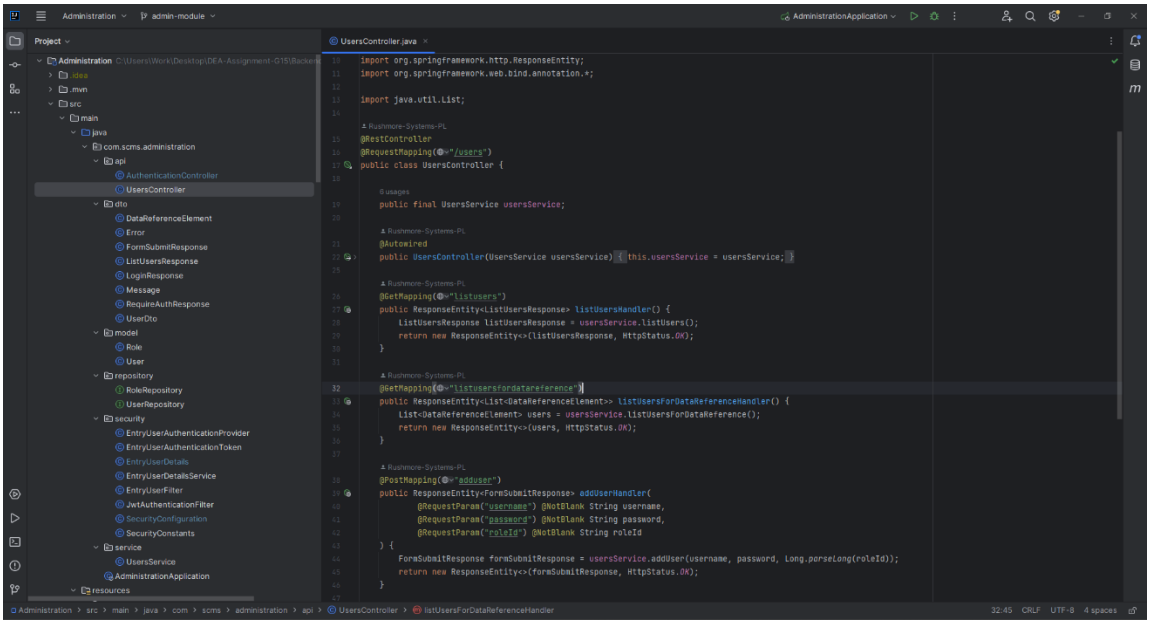


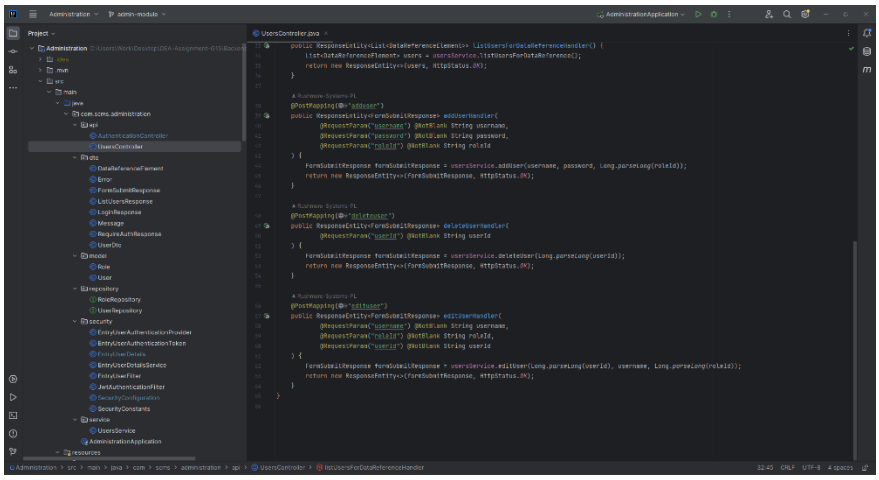
request, editing via a PUT request and deleting via a DELETE request. Data retrieval occurs via GET request.

The User Conroller then hands off the data extracted from the requests to their respective methods of the UserService to perform business logic. Data base operations are performed by methods in the UserRepository which are overridden from the CrudRepository interface.

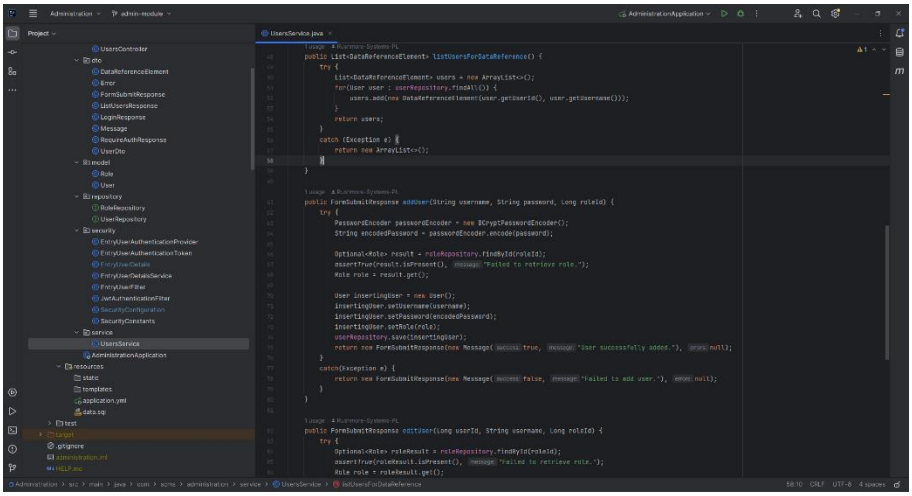
The User entity provides a domain model representation of the User table in the database. Similarly, the Role class represents the Role table, and they have a Many to One relationship as denoted by the @ManyToOne annotation.

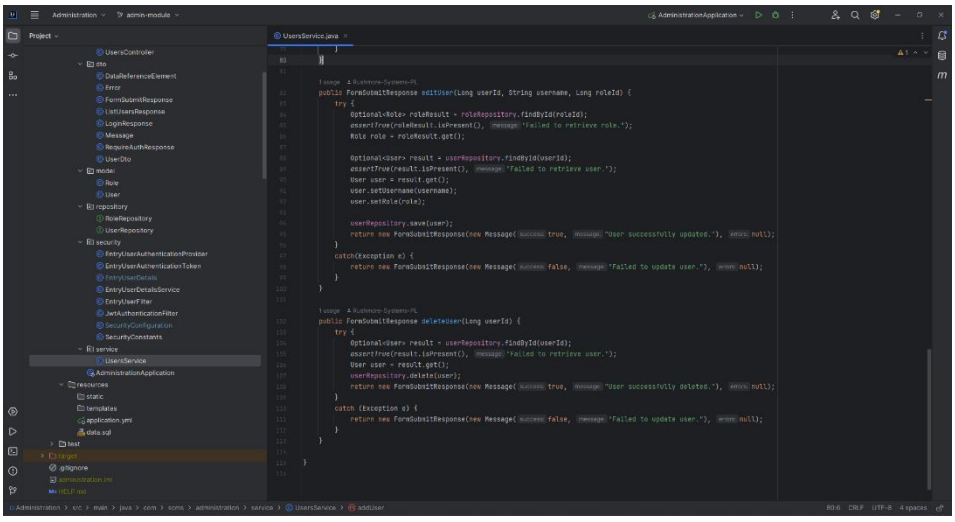
Controller



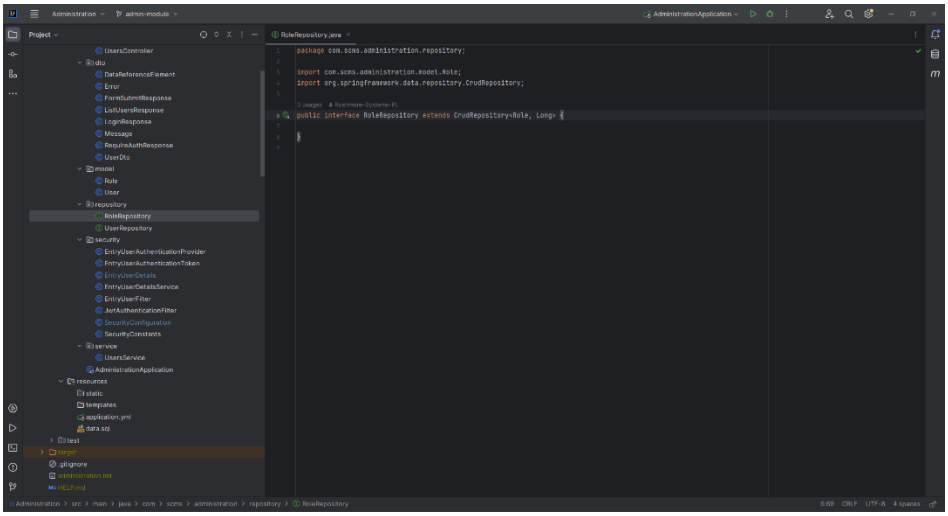


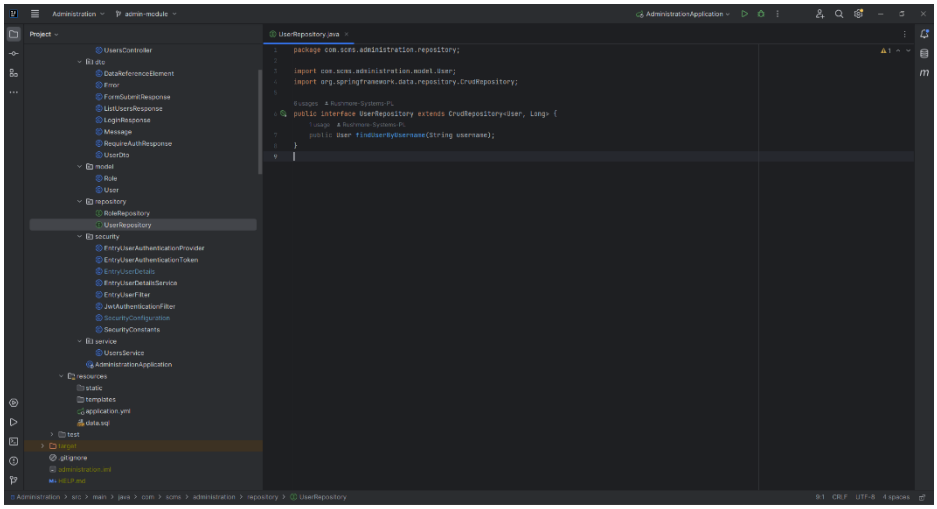
Service



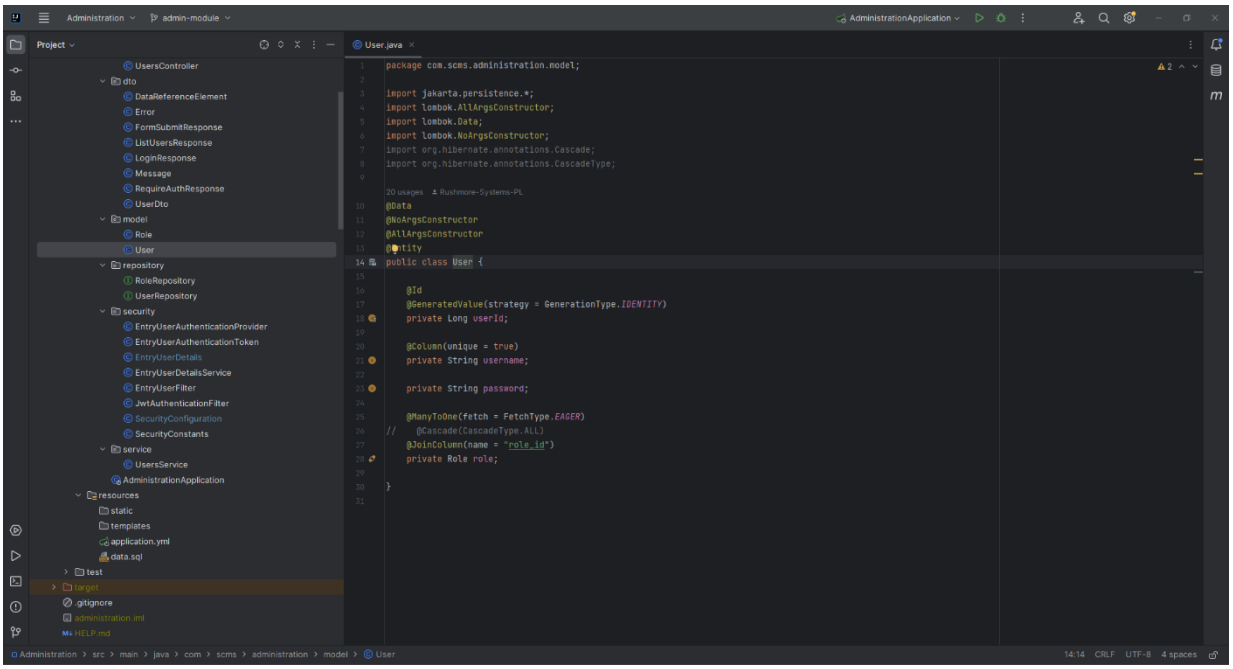


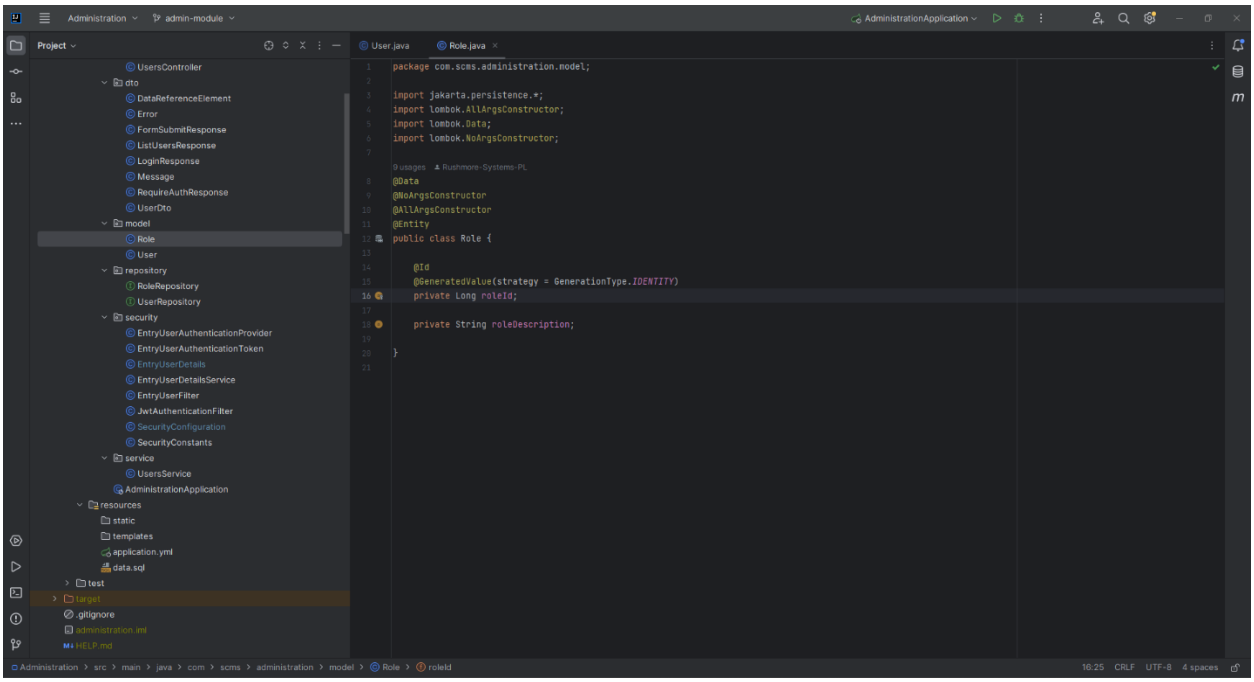
Repository


package com.scms.administration.repository;
import com.scms.administration.model.Role;
import org.springframework.data.repository.CrudRepository;
public interface RoleRepository extends CrudRepository<Role, Long> {
}


package com.scms.administration.repository;
import com.scms.administration.model.User;
import org.springframework.data.repository.CrudRepository;
public interface UserRepository extends CrudRepository<User, Long> {
public User findUserByUsername(String username);
}

Models

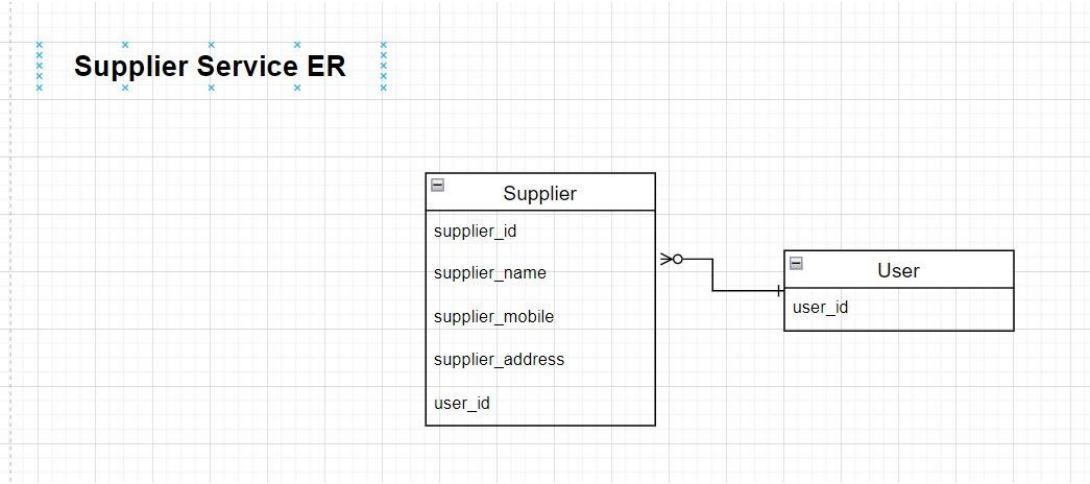

Administration
admin-module
AdministrationApplication
Project
User.java
UsersController
dto
DataReferenceElement
Error
FormSubmitResponse
ListUsersResponse
LoginResponse
Message
RequireAuthResponse
UserDto
model
Role
User
repository
RoleRepository
UserRepository
security
EntryUserAuthenticationProvider
EntryUserAuthenticationToken
EntryUserDetails
EntryUserDetailsService
EntryUserFilter
JwtAuthenticationFilter
SecurityConfiguration
SecurityConstants
service
UsersService
AdministrationApplication
resources
static
templates
application.yml
data.sql
test
target
.gitignore
administration.iml
HELP.md
package com.scms.administration.model;
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.Cascade;
import org.hibernate.annotations.CascadeType;
20 usages Rushmore-Systems-PL
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class User {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long userId;
@Column(unique = true)
private String username;
private String password;
@ManyToOne(fetch = FetchType.EAGER)
// @Cascade(CascadeType.ALL)
@JoinColumn(name = "role_id")
private Role role;
}
Administration › src › main › java › com › scms › administration › model › User
14:14 CRLF UTF-8 4 spaces


Administration
admin-module
AdministrationApplication
Project
User.java
Role.java
UsersController
dto
DataReferenceElement
Error
FormSubmitResponse
ListUsersResponse
LoginResponse
Message
RequireAuthResponse
UserDto
model
Role
User
repository
RoleRepository
UserRepository
security
EntryUserAuthenticationProvider
EntryUserAuthenticationToken
EntryUserDetails
EntryUserDetailsService
EntryUserFilter
JwtAuthenticationFilter
SecurityConfiguration
SecurityConstants
service
UsersService
AdministrationApplication
resources
static
templates
application.yml
data.sql
test
target
.gitignore
administration.iml
HELP.md
package com.scms.administration.model;
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
9 usages Rushmore-Systems-PL
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class Role {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long roleId;
private String roleDescription;
}
Administration › src › main › java › com › scms › administration › model › Role › roleId
16:25 CRLF UTF-8 4 spaces

Supplier Microservice

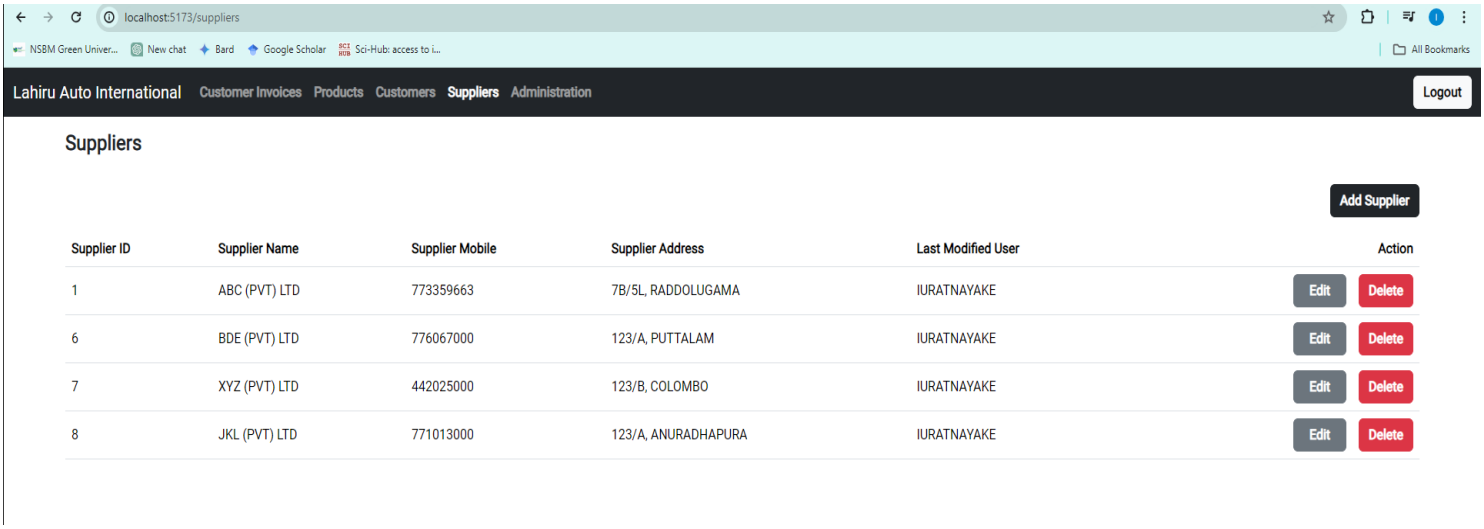
The Supplier Microservice helps us manage supplier information smoothly. Its main job is to handle basic tasks like adding new suppliers, checking supplier details, updating information, and removing suppliers if needed. Think of it as a central hub where all supplier-related actions take place.

This microservice is built in a way that makes it easy to connect with other parts of our system. We use something called RESTful architecture, which ensures that different systems can talk to each other easily. It's like having a common language for communication. This setup not only makes things flexible for us but also sets a strong foundation for future improvements. We can add more features or scale up the system without much hassle because of this design.

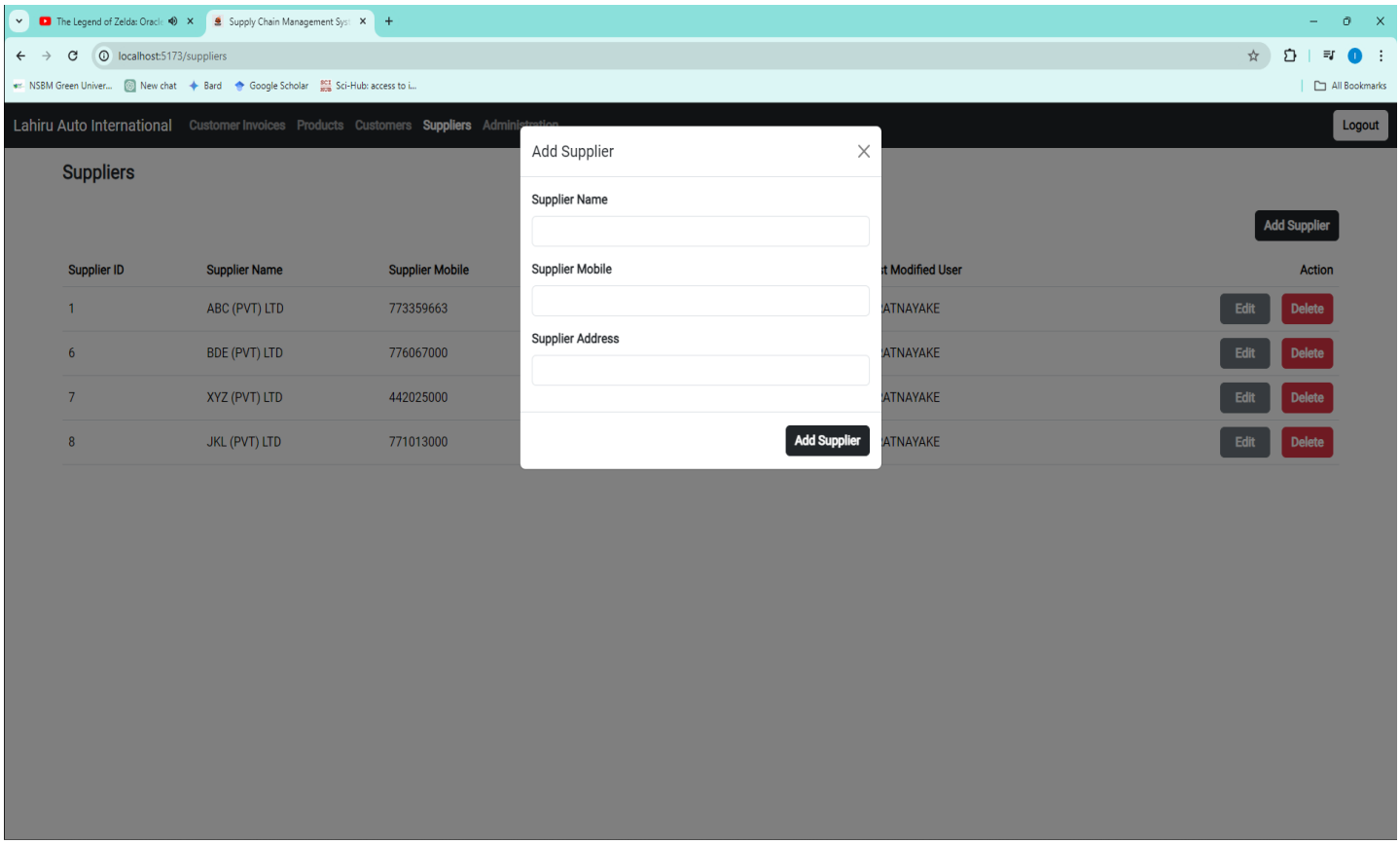


Functionality

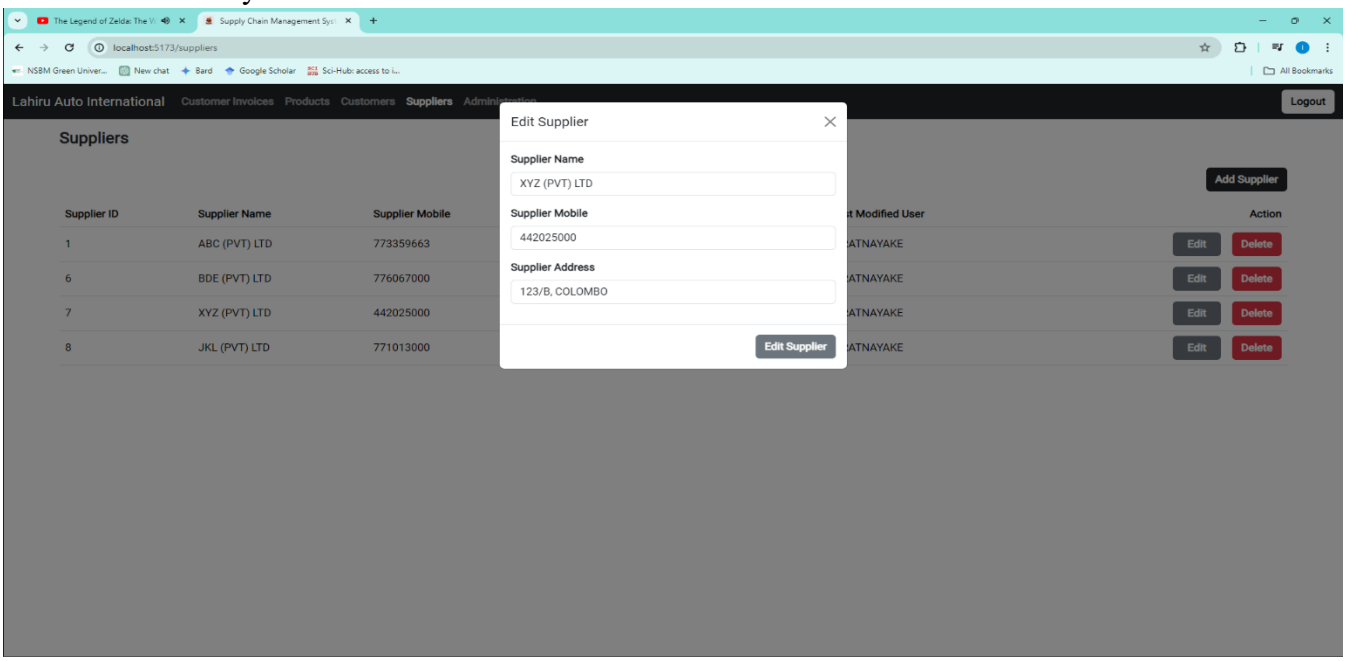
- **Search Supplier**



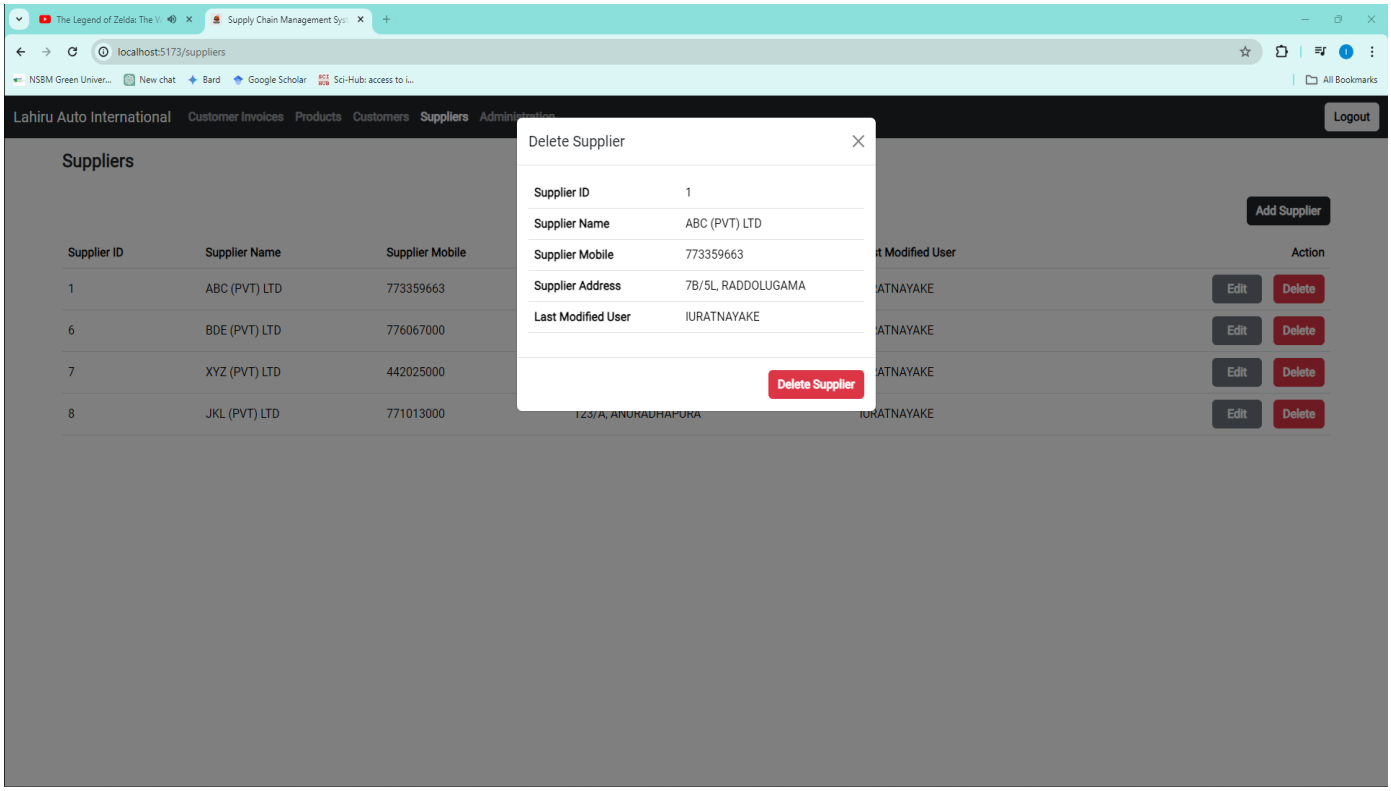
❖ **Add Supplier**: Enables the addition of new suppliers to the system with relevant details



❖ **Edit Supplier**: Allows users to update existing supplier information, ensuring data accuracy and relevance.



- **Delete Supplier**: Provides functionality to remove suppliers from the system, with appropriate validation and authorization checks.

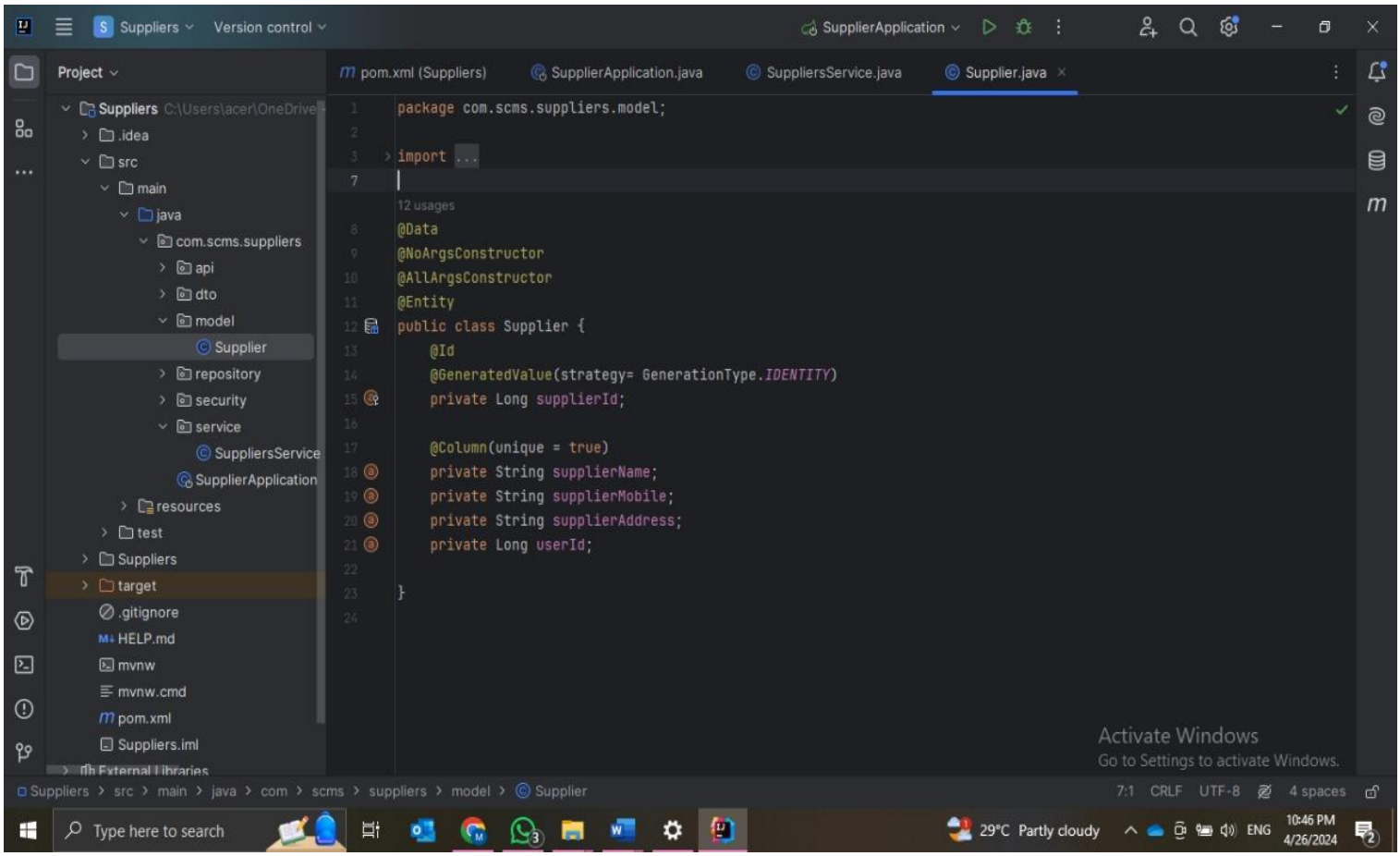


Architecture

- **Spring Boot**: The microservice is built using the Spring Boot framework, offering rapid development and deployment capabilities.
- **RESTful API**: Endpoints are exposed following REST principles, allowing easy interaction with the microservice over HTTP.
- **Maven**: Dependency management and project build are handled using Maven, ensuring project consistency and efficiency.

model

- Supplier

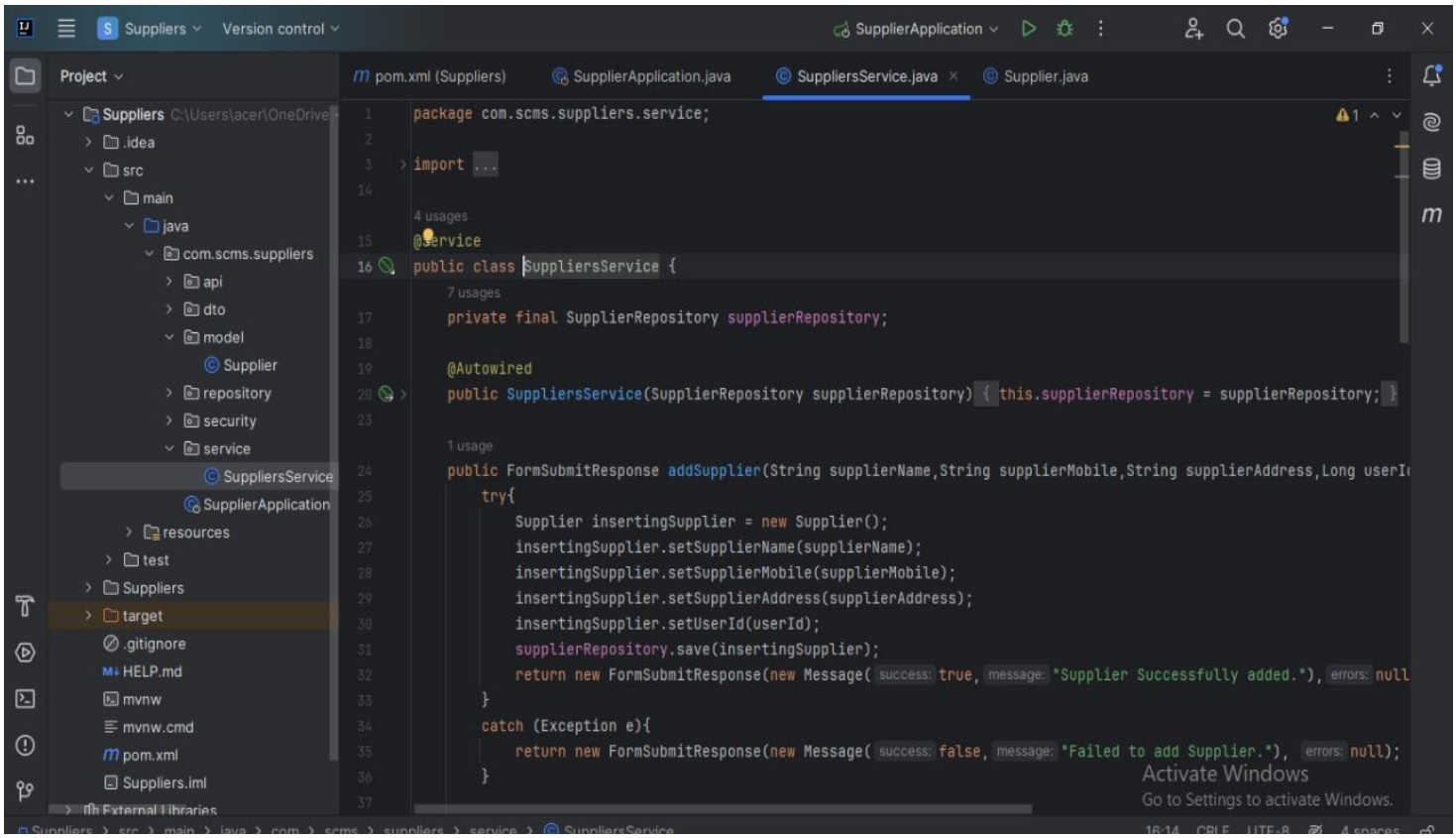


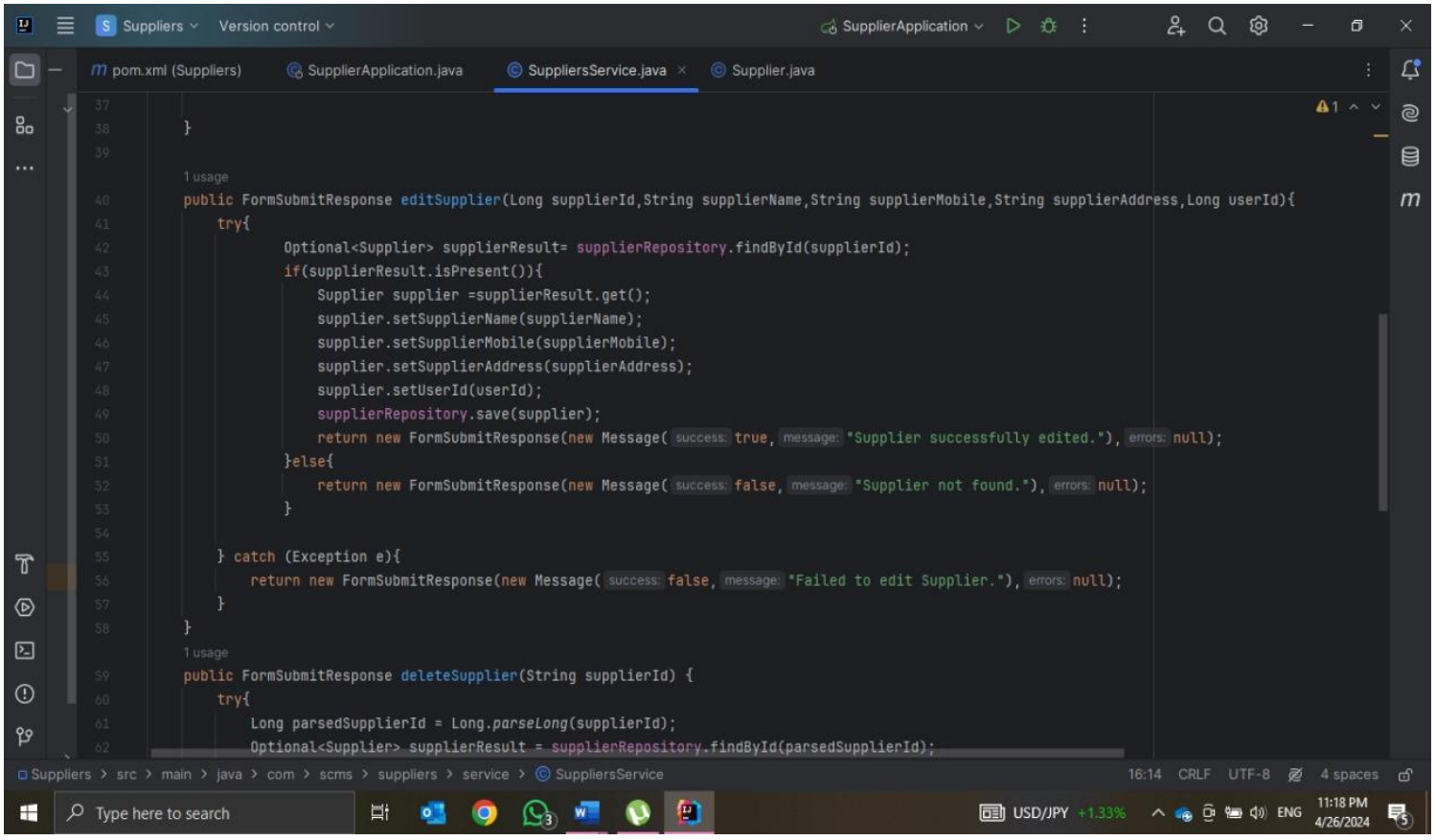
service

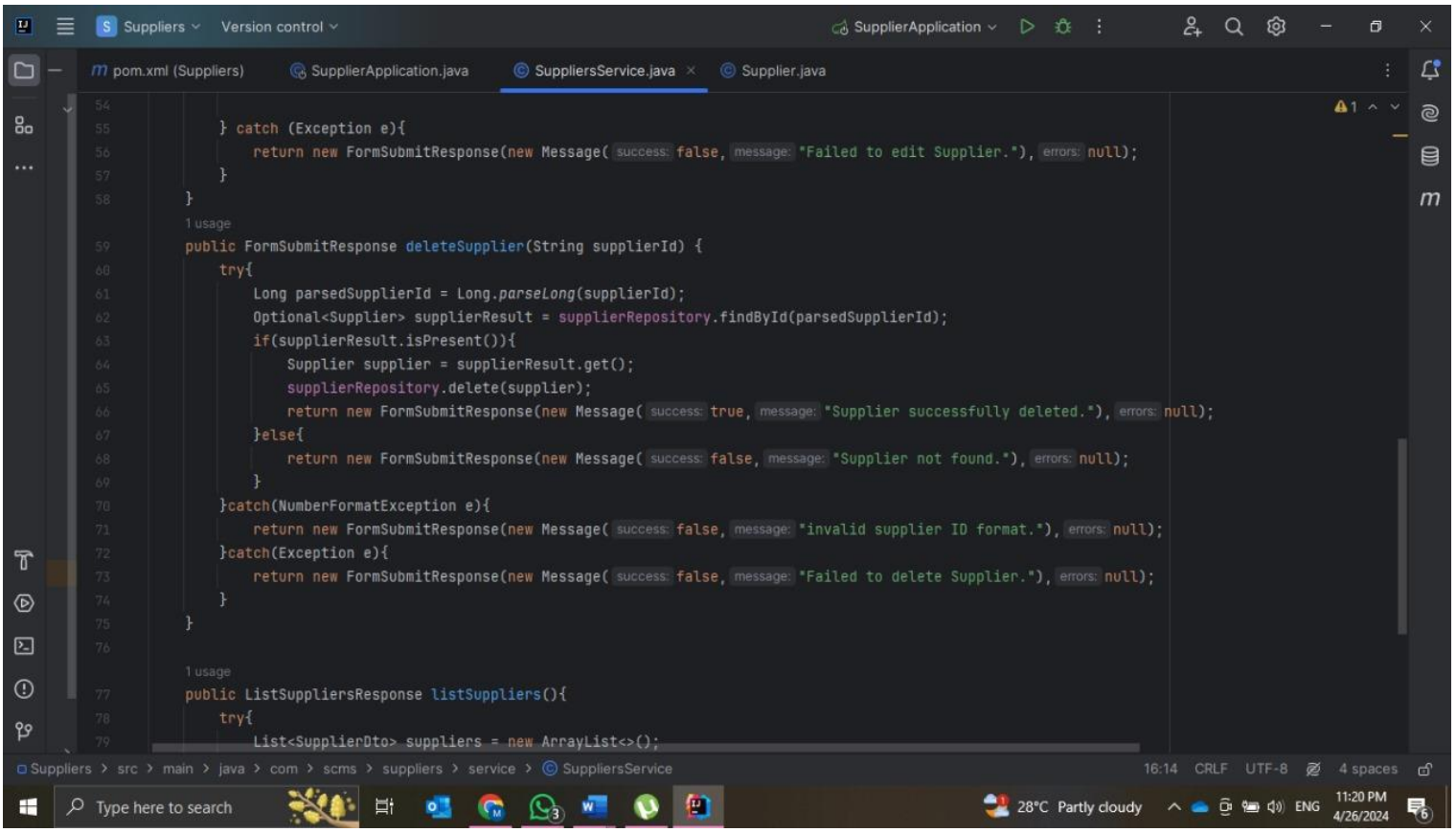
- SuppliersService
  - ✓ addSupplier
  - ✓ editSupplier
  - ✓ deleteSupplier
  - ✓ listSuppliers

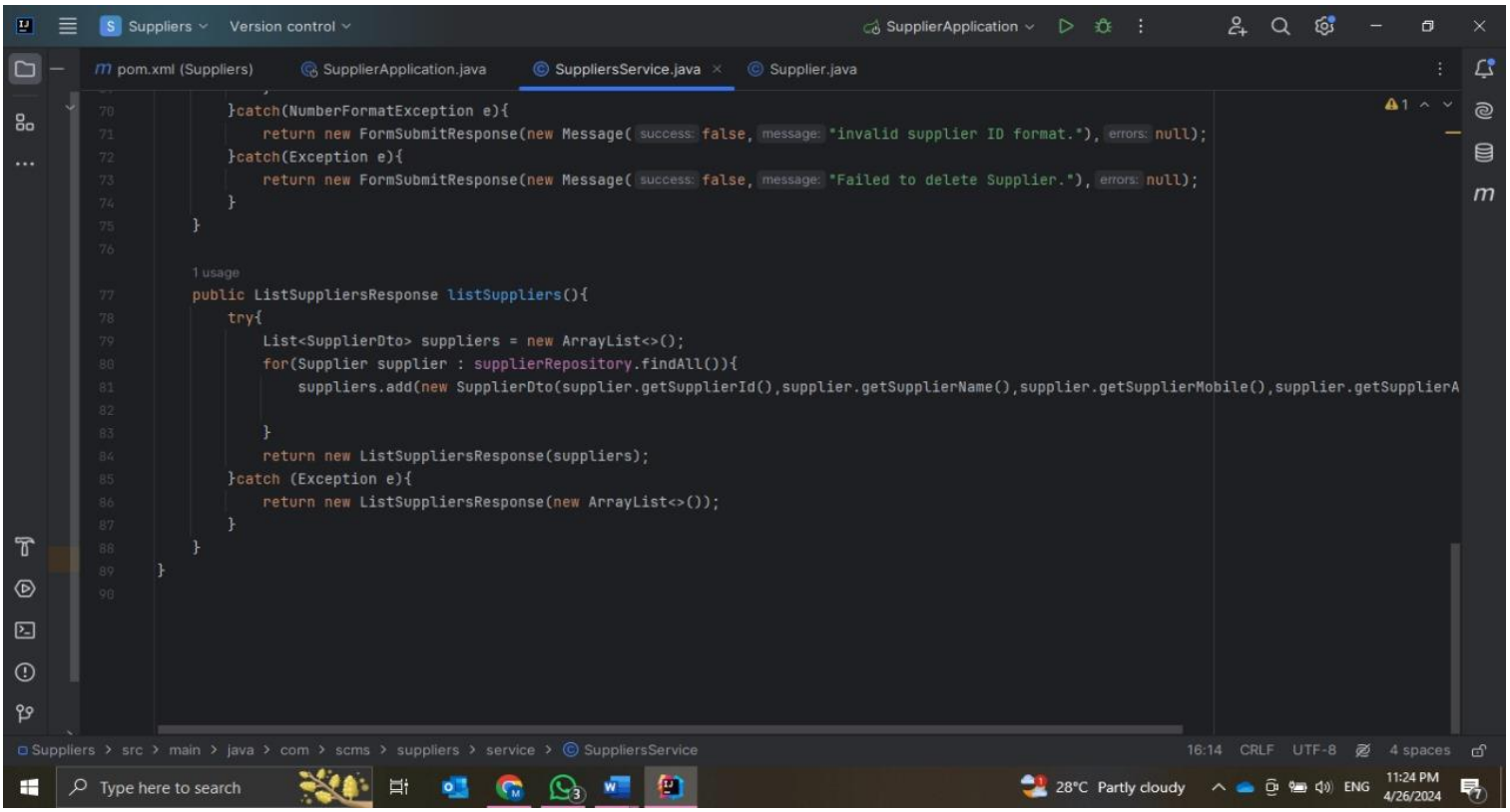
api

- SupplierController


Suppliers
Version control
SupplierApplication
Project
pom.xml (Suppliers)
SupplierApplication.java
SuppliersService.java
Supplier.java
Suppliers C:\Users\acer\OneDrive
.idea
src
main
java
com.scms.suppliers
api
dto
model
Supplier
repository
security
service
SuppliersService
SupplierApplication
resources
test
Suppliers
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
Suppliers.iml
External Libraries
package com.scms.suppliers.service;
import ...
4 usages
@Service
public class SuppliersService {
7 usages
private final SupplierRepository supplierRepository;
@Autowired
public SuppliersService(SupplierRepository supplierRepository) { this.supplierRepository = supplierRepository; }
1 usage
public FormSubmitResponse addSupplier(String supplierName,String supplierMobile,String supplierAddress,Long userI
try{
Supplier insertingSupplier = new Supplier();
insertingSupplier.setSupplierName(supplierName);
insertingSupplier.setSupplierMobile(supplierMobile);
insertingSupplier.setSupplierAddress(supplierAddress);
insertingSupplier.setUserId(userId);
supplierRepository.save(insertingSupplier);
return new FormSubmitResponse(new Message(success: true, message: "Supplier Successfully added."), errors: null
}
catch (Exception e){
return new FormSubmitResponse(new Message(success: false, message: "Failed to add Supplier."), errors: null);
}
Activate Windows
Go to Settings to activate Windows.


Suppliers
Version control
SupplierApplication
pom.xml (Suppliers)
SupplierApplication.java
SuppliersService.java
Supplier.java
}
1 usage
public FormSubmitResponse editSupplier(Long supplierId,String supplierName,String supplierMobile,String supplierAddress,Long userId){
try{
Optional<Supplier> supplierResult= supplierRepository.findById(supplierId);
if(supplierResult.isPresent()){
Supplier supplier =supplierResult.get();
supplier.setSupplierName(supplierName);
supplier.setSupplierMobile(supplierMobile);
supplier.setSupplierAddress(supplierAddress);
supplier.setUserId(userId);
supplierRepository.save(supplier);
return new FormSubmitResponse(new Message(success: true, message: "Supplier successfully edited."), errors: null);
}else{
return new FormSubmitResponse(new Message(success: false, message: "Supplier not found."), errors: null);
}
} catch (Exception e){
return new FormSubmitResponse(new Message(success: false, message: "Failed to edit Supplier."), errors: null);
}
}
1 usage
public FormSubmitResponse deleteSupplier(String supplierId) {
try{
Long parsedSupplierId = Long.parseLong(supplierId);
Optional<Supplier> supplierResult = supplierRepository.findById(parsedSupplierId);
Suppliers > src > main > java > com > scms > suppliers > service > SuppliersService
16:14 CRLF UTF-8 4 spaces
Type here to search
USD/JPY +1.33%
ENG
11:18 PM
4/26/2024


Suppliers
Version control
SupplierApplication
pom.xml (Suppliers)
SupplierApplication.java
SuppliersService.java
Supplier.java
} catch (Exception e){
return new FormSubmitResponse(new Message(success: false, message: "Failed to edit Supplier."), errors: null);
}
}
1 usage
public FormSubmitResponse deleteSupplier(String supplierId) {
try{
Long parsedSupplierId = Long.parseLong(supplierId);
Optional<Supplier> supplierResult = supplierRepository.findById(parsedSupplierId);
if(supplierResult.isPresent()){
Supplier supplier = supplierResult.get();
supplierRepository.delete(supplier);
return new FormSubmitResponse(new Message(success: true, message: "Supplier successfully deleted."), errors: null);
}else{
return new FormSubmitResponse(new Message(success: false, message: "Supplier not found."), errors: null);
}
}catch(NumberFormatException e){
return new FormSubmitResponse(new Message(success: false, message: "invalid supplier ID format."), errors: null);
}catch(Exception e){
return new FormSubmitResponse(new Message(success: false, message: "Failed to delete Supplier."), errors: null);
}
}
1 usage
public ListSuppliersResponse listSuppliers(){
try{
List<SupplierDto> suppliers = new ArrayList<>();
Suppliers > src > main > java > com > scms > suppliers > service > SuppliersService
16:14 CRLF UTF-8 4 spaces
Type here to search
28°C Partly cloudy
ENG
11:20 PM
4/26/2024


Suppliers
Version control
SupplierApplication
pom.xml (Suppliers)
SupplierApplication.java
SuppliersService.java
Supplier.java
}catch(NumberFormatException e){
return new FormSubmitResponse(new Message(success: false, message: "invalid supplier ID format."), errors: null);
}catch(Exception e){
return new FormSubmitResponse(new Message(success: false, message: "Failed to delete Supplier."), errors: null);
}
}
1 usage
public ListSuppliersResponse listSuppliers(){
try{
List<SupplierDto> suppliers = new ArrayList<>();
for(Supplier supplier : supplierRepository.findAll()){
suppliers.add(new SupplierDto(supplier.getSupplierId(),supplier.getSupplierName(),supplier.getSupplierMobile(),supplier.getSupplierA
}
return new ListSuppliersResponse(suppliers);
}catch (Exception e){
return new ListSuppliersResponse(new ArrayList<>());
}
}
}
Suppliers > src > main > java > com > scms > suppliers > service > SuppliersService
16:14 CRLF UTF-8 4 spaces
Type here to search
28°C Partly cloudy
ENG
11:24 PM
4/26/2024

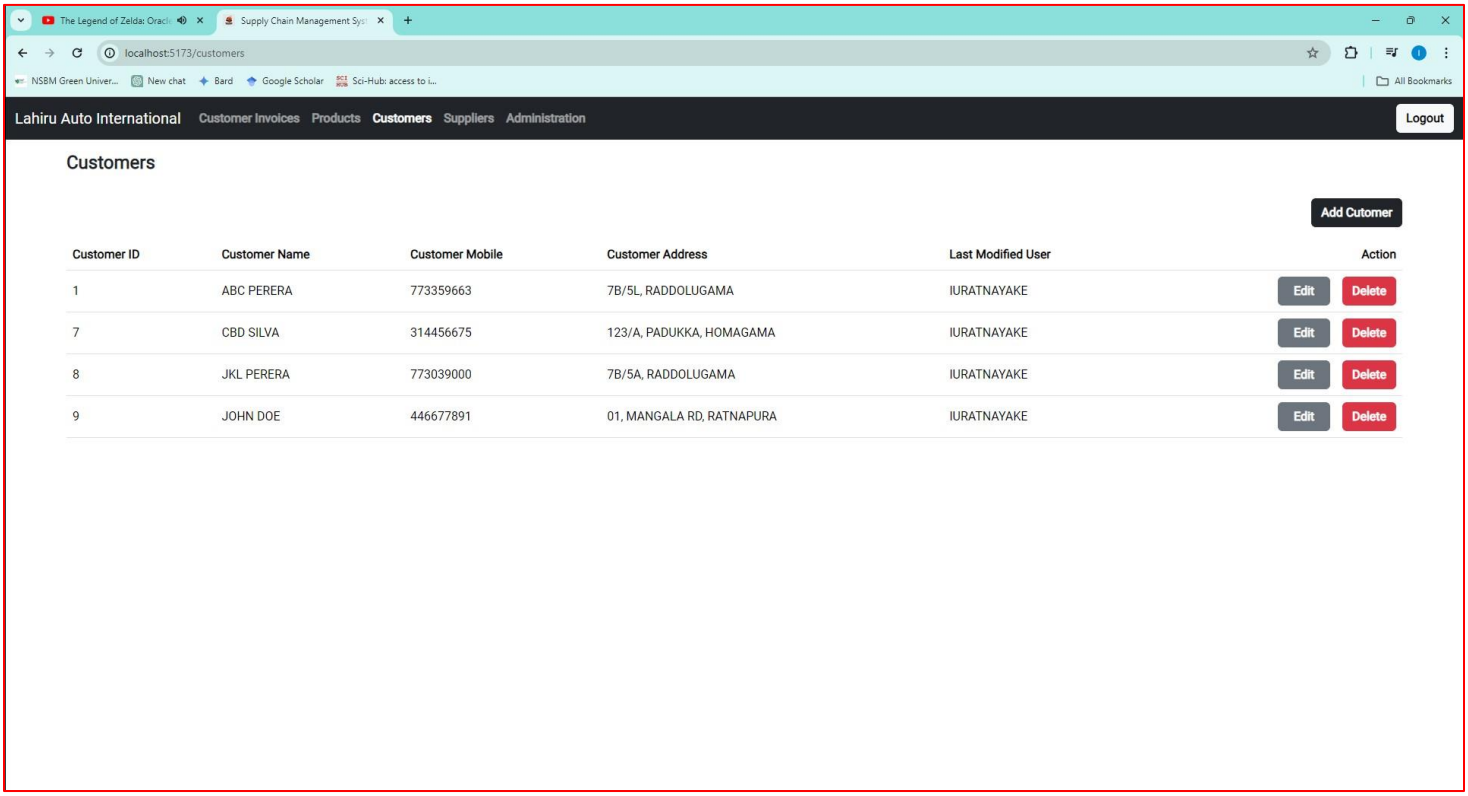
Customer Microservice

Front End

The Customer Management Microservice

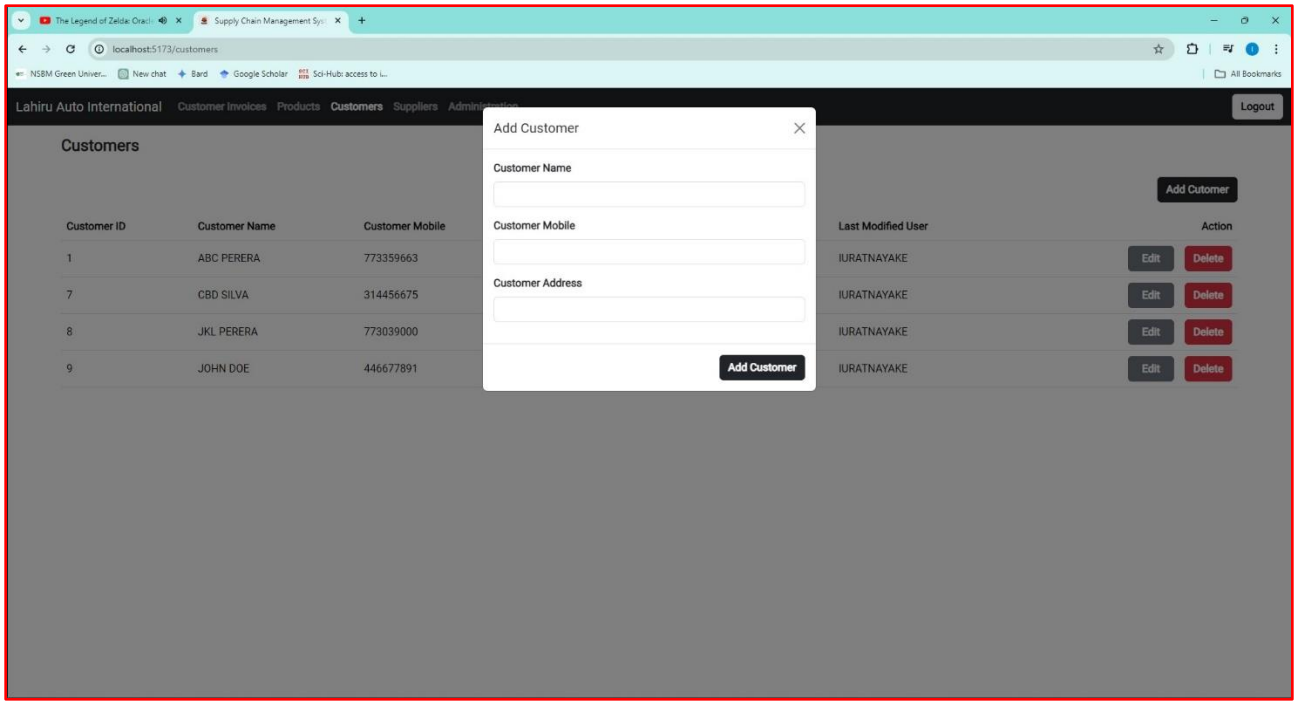
The Customer Management Microservice helps us manage Customer information smoothly. Its main job is to handle basic tasks like adding new Customer, checking Customer details, updating information, and deleting Customer's if needed.

Customer details page



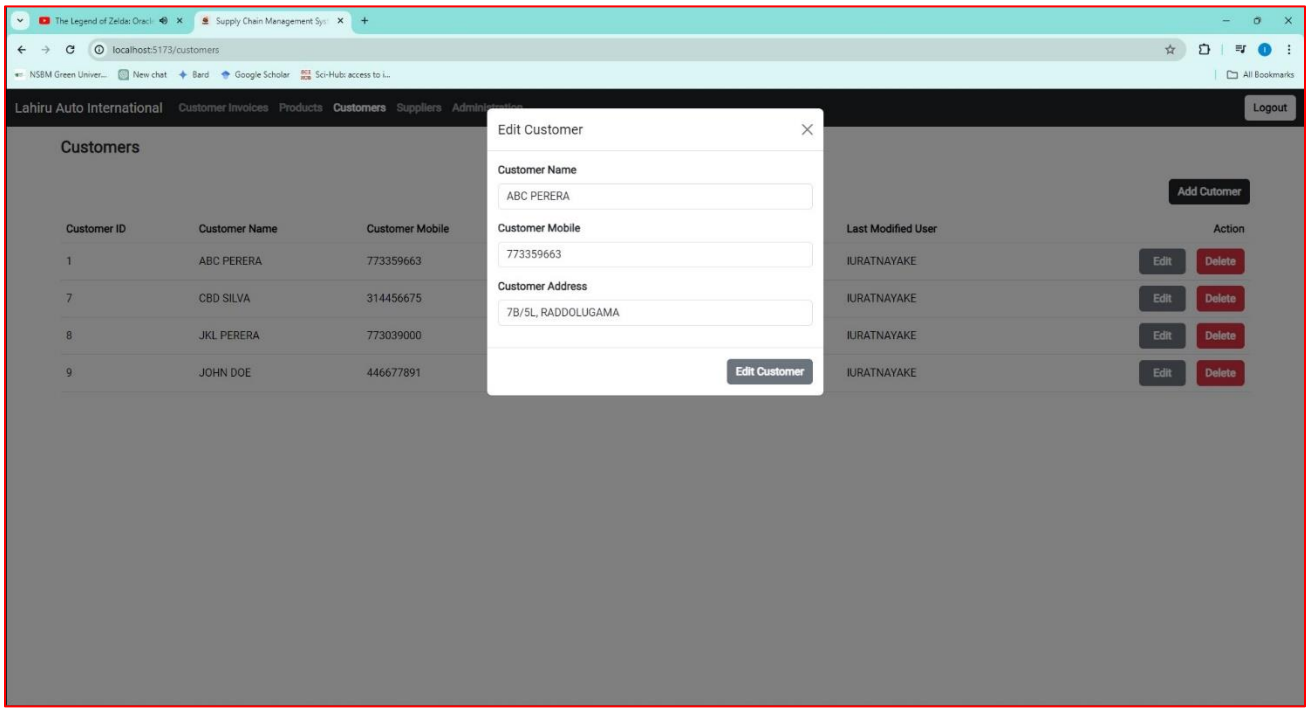
Customer Create Page

In here you can add new customer, Add Customer Name, Customer Mobile and Customer Address.



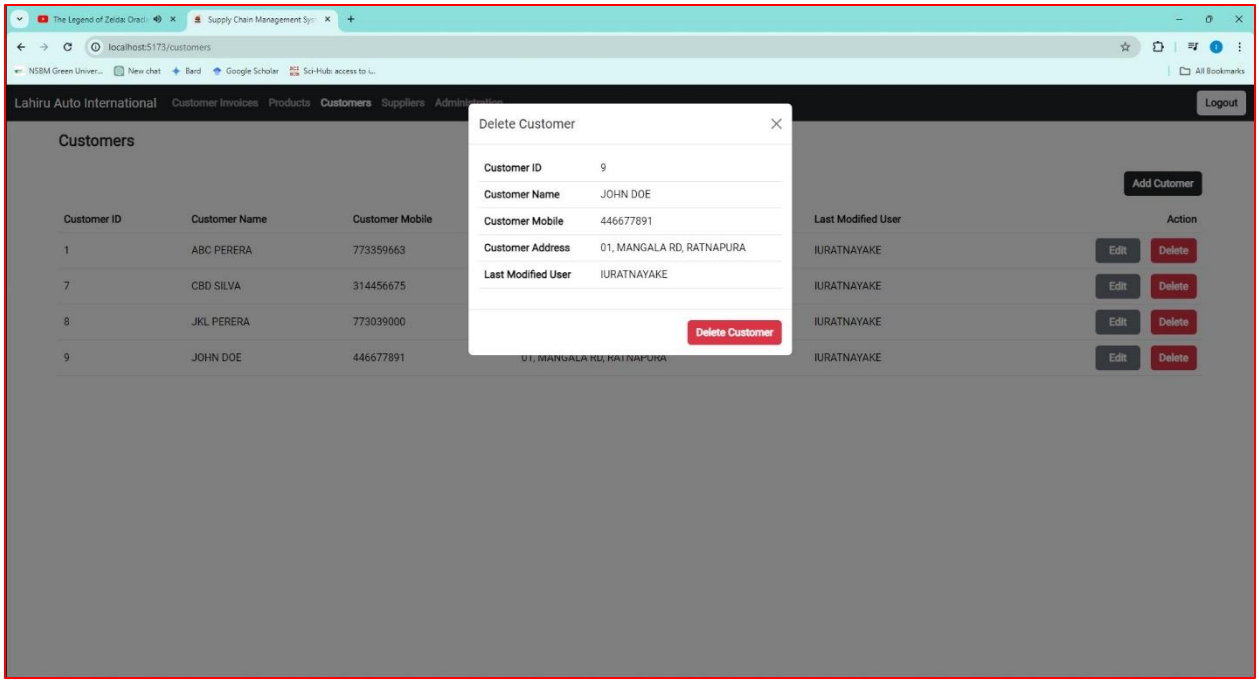
Customer Edit Page

In here you can Edit customer, Edit Customer Name, Customer Mobile and Customer Address

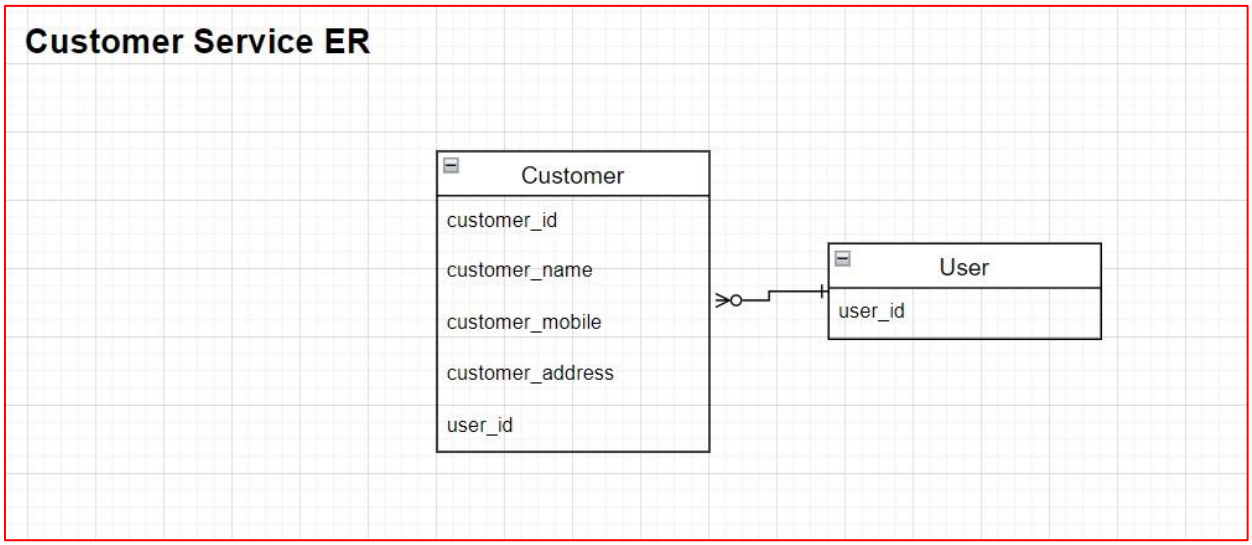


Customer delete Page

In here you can delete customer use by delete button.



ER Diagram

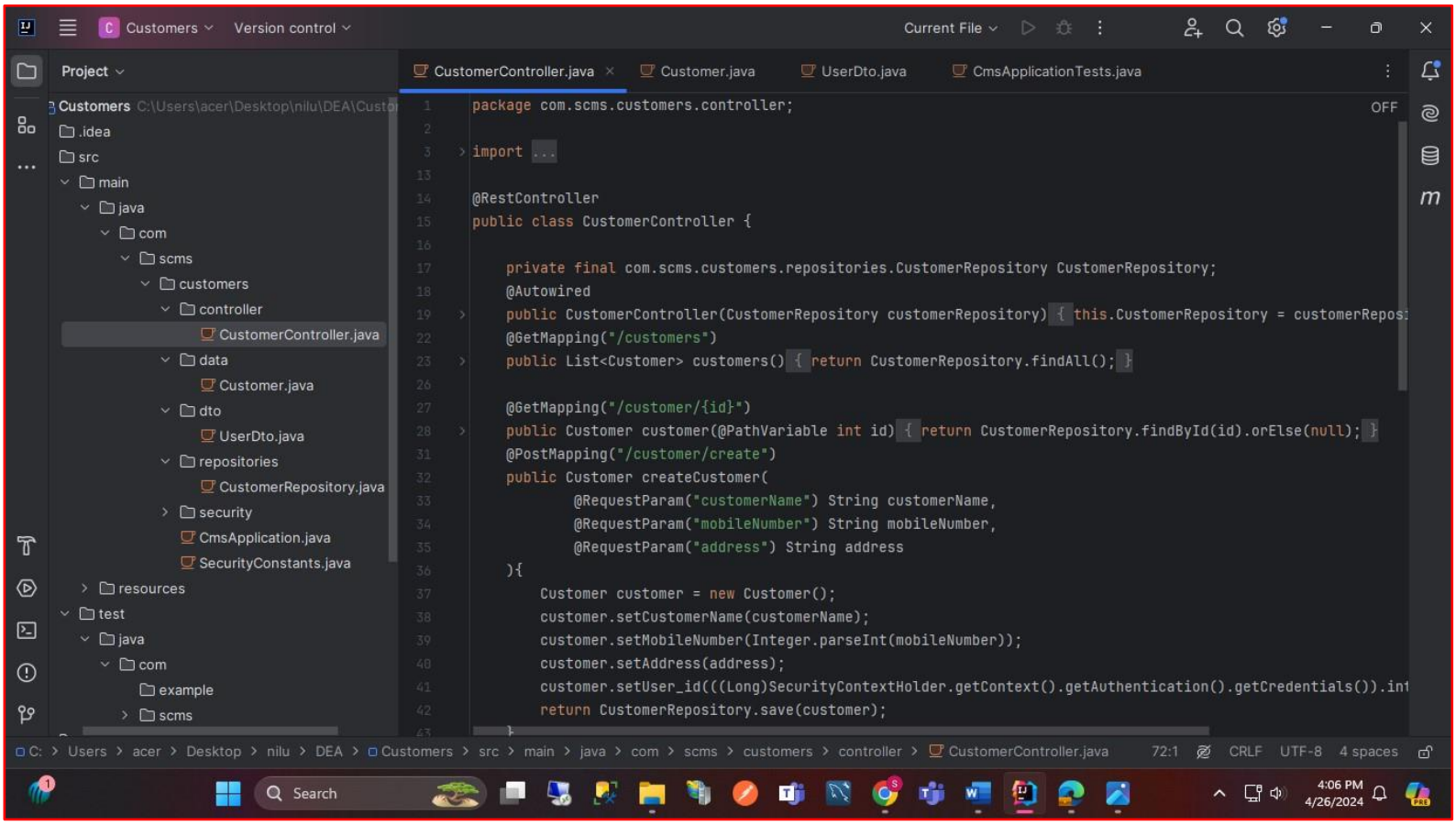

Customer Service ER
Customer
customer_id
customer_name
customer_mobile
customer_address
user_id
User
user_id

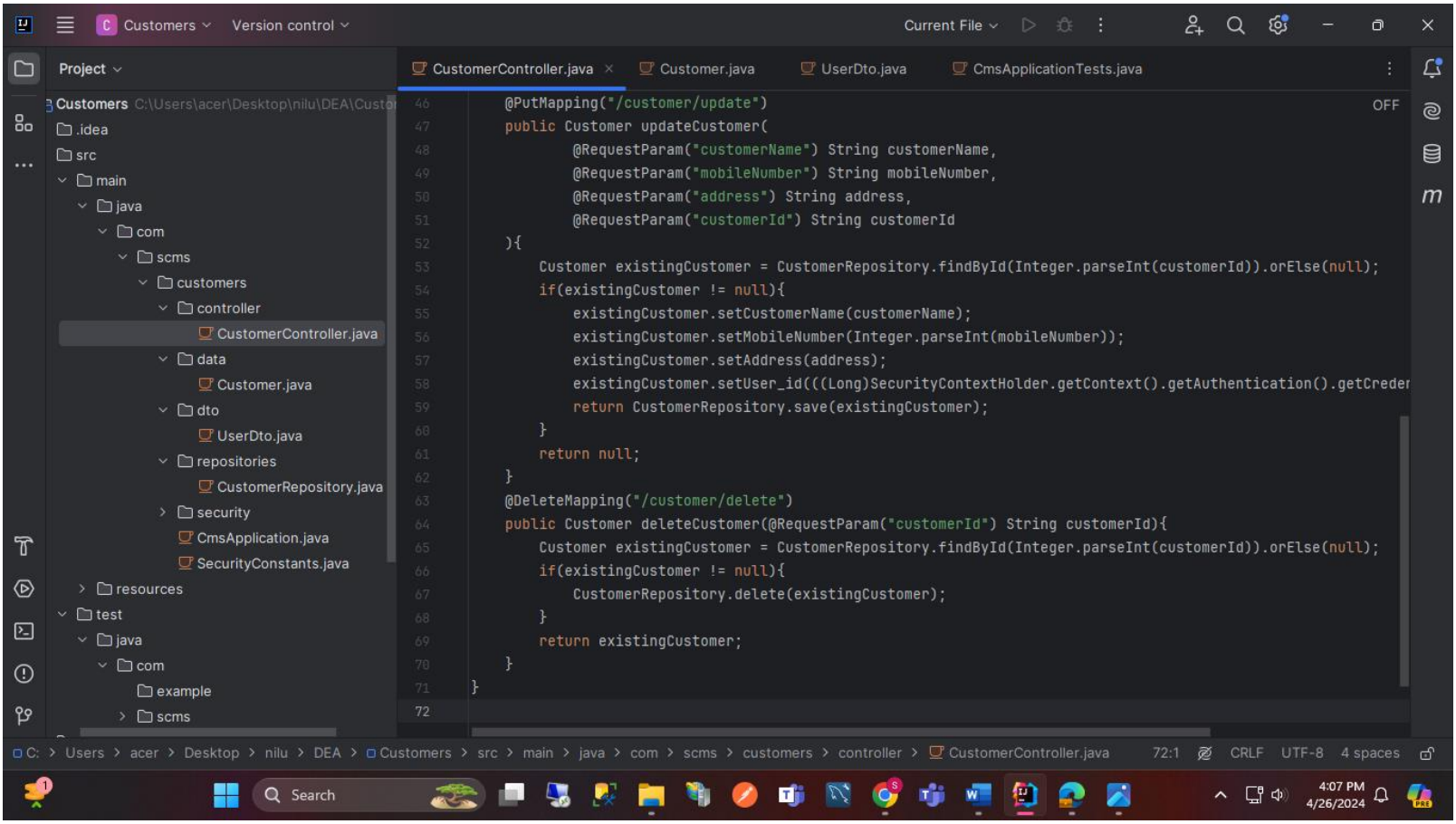
Backend Code

CustomerController.java

Purpose: The CustomerController class serves as a controller for handling HTTP requests related to customer data .

Dependencies: It depends on CustomerRepository for data access operations.





Methods:

01. GET /customers: Retrieves a list of all customers.

- Endpoint: /customers
- HTTP Method: GET
- Return Type: List<Customer>

02. GET /customer/{id}: Retrieves a specific customer by ID.

- Endpoint: /customer/{id}
- HTTP Method: GET
- Path Variable: id (int)
- Return Type: Customer

03. POST /customer/create: Creates a new customer.

- Endpoint: /customer/create
- HTTP Method: POST
- Request Parameters:
- customerName (String)
- mobileNumber (String)
- address (String)
- Return Type: Customer

04. PUT /customer/update: Updates an existing customer.

- Endpoint: /customer/update
- HTTP Method: PUT
- Request Parameters:
- customerId (String)
- customerName (String)
- mobileNumber (String)
- address (String)
- Return Type: Customer

05. DELETE /customer/delete: Deletes a customer.

- Endpoint: /customer/delete
- HTTP Method: DELETE
- Request Parameter:
- customerId (String)
- Return Type: Customer

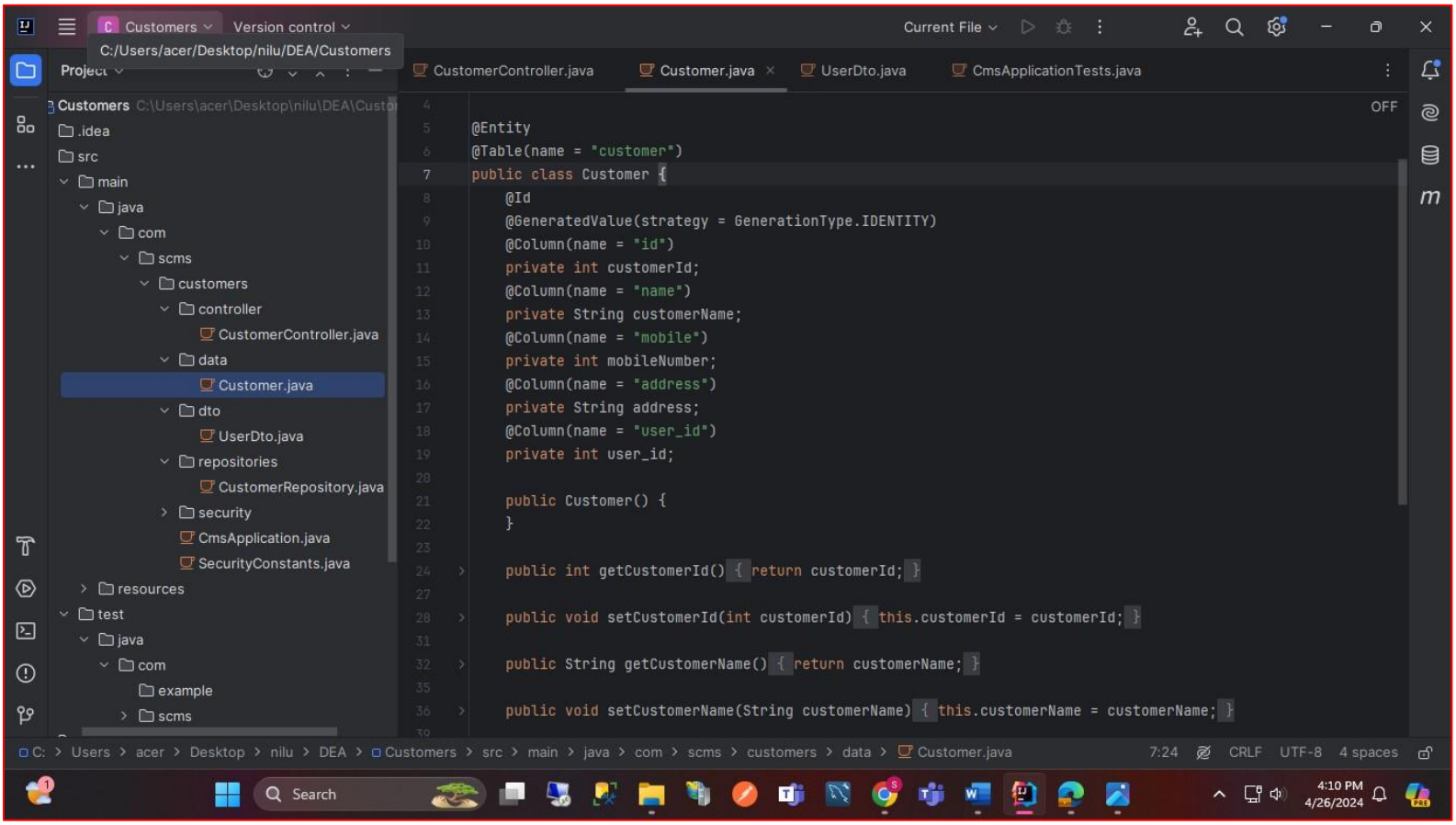
Key Features:

- Data Binding: Uses @RequestParam to bind request parameters to method parameters.

- Data Conversion: Converts mobileNumber from String to int for database storage.
- Security: Utilizes SecurityContextHolder for user authentication and authorization.
- CRUD Operations: Supports Create, Read, Update, and Delete operations for customer data.

Customer.java

Purpose: The Customer class represents the structure of customer data.



Fields: Includes fields such as customerId, customerName, mobileNumber, address, and user_id.

Attributes:

- customerId (int): Unique identifier for each customer.
- customerName (String): Name of the customer.
- mobileNumber (int): Mobile number of the customer.
- address (String): Address of the customer.

- user_id (int): ID of the user associated with the customer.

Constructors:

Includes a default constructor and getters/setters for all fields.

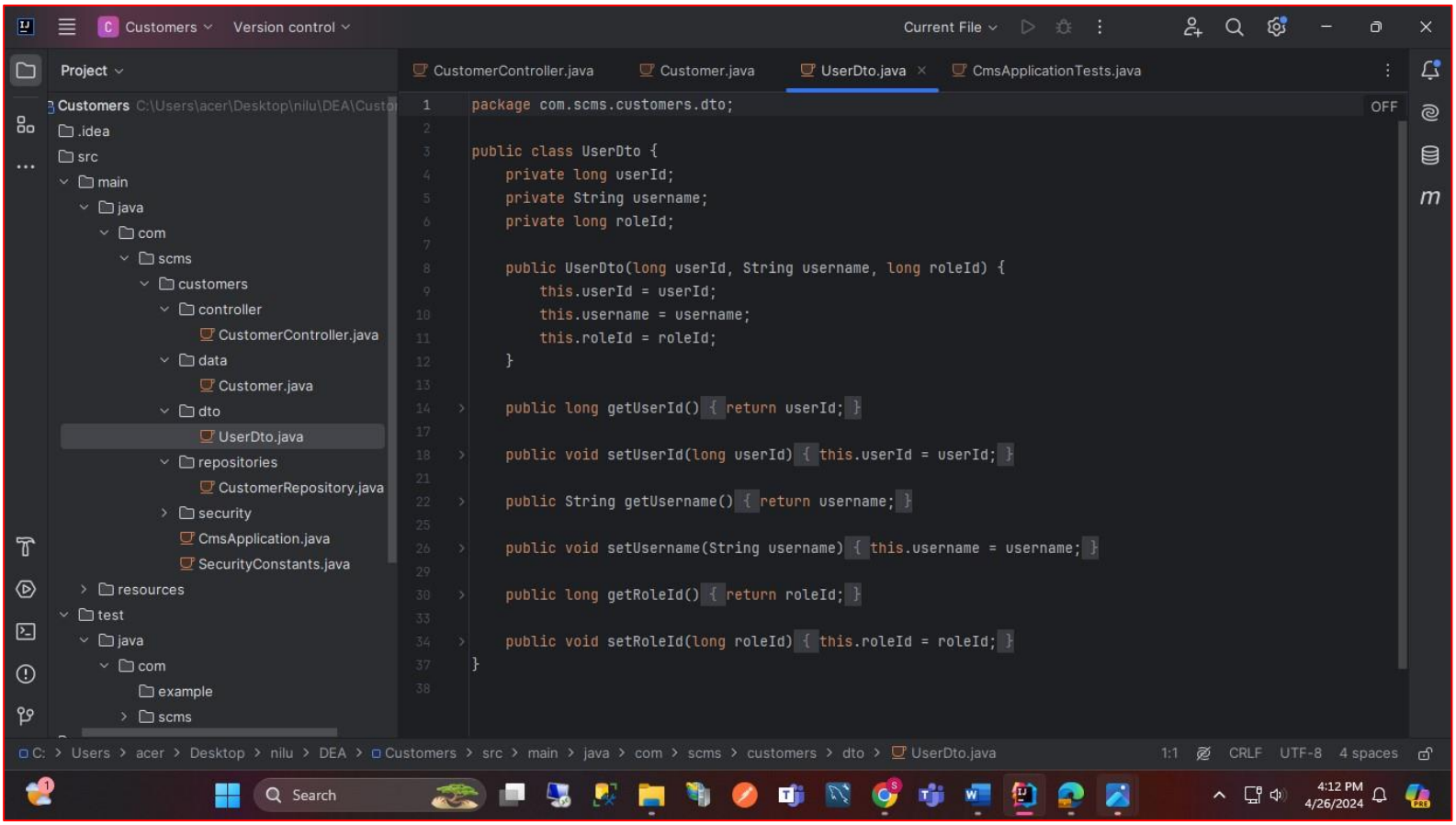
Key Features:

- Entity Mapping: Annotated with @Entity and mapped to the "customer" table in the database.
- Primary Key: Uses @Id and @GeneratedValue for auto-generated primary key.
- Table Mapping: Mapped to the "customer" table using @Table(name = "customer").
- Column Mapping: Maps fields to specific columns in the database using @Column.

UserDto.java

Overview:

Purpose: The UserDto class represents a Data Transfer Object (DTO) for user-related data in your application.



Fields: Includes fields such as userId, username, and roleId for user information.

Attributes:

- userId (long): Unique identifier for each user.
- username (String): User's username.
- roleId (long): ID of the role associated with the user.

Constructor:

Includes a constructor to initialize the UserDto object with user-related information.

Methods:

Getter and setter methods for all fields to access and modify user data.

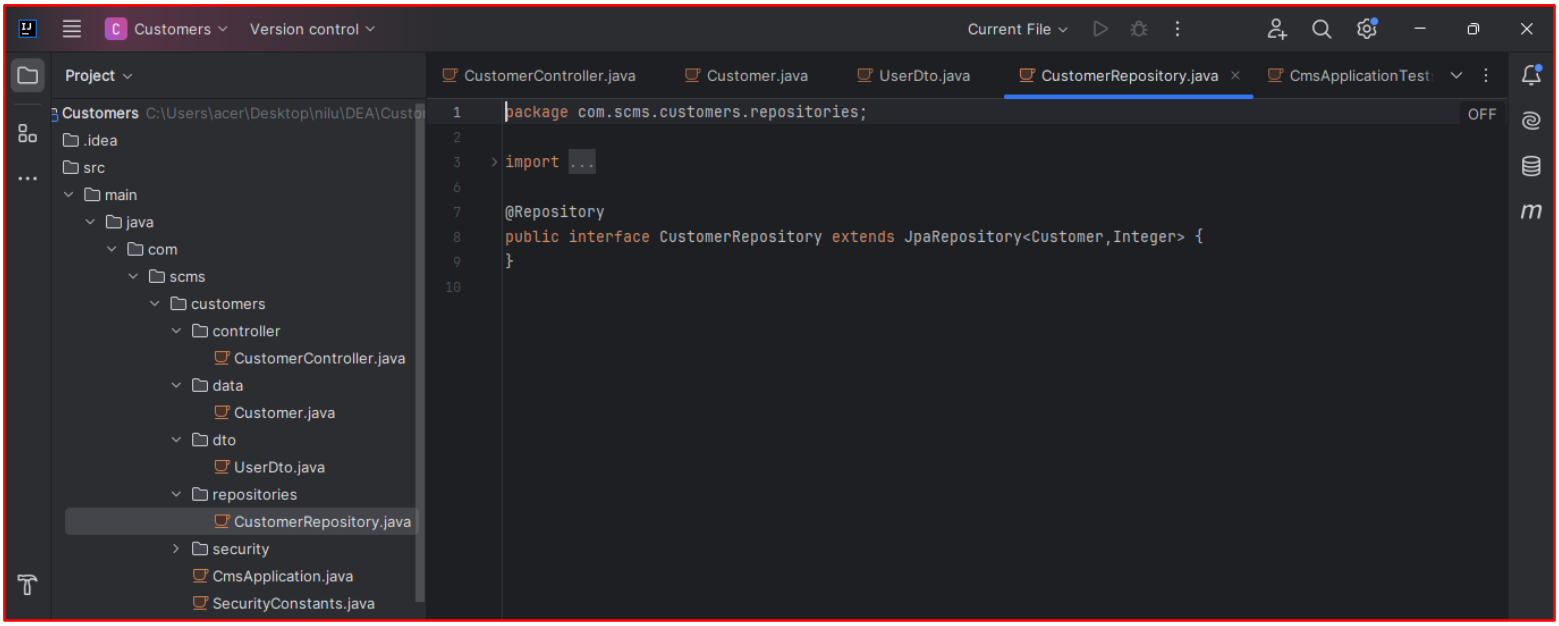
Usage:

Typically used to transfer user data between layers of application, such as between the controller and service layers.

CustomerRepository.java

Overview:

Purpose: The CustomerRepository interface extends JpaRepository<Customer, Integer> to handle database operations for the Customer entity.



Functionality: Provides CRUD (Create, Read, Update, Delete) operations for customer data using Spring Data JPA.

Methods:

- Inherits methods from JpaRepository, including save, findById, findAll, and delete.
- Additional custom methods can be added as needed for specific data access requirements.

Annotation:

Annotated with @Repository to indicate that it's a Spring repository bean.

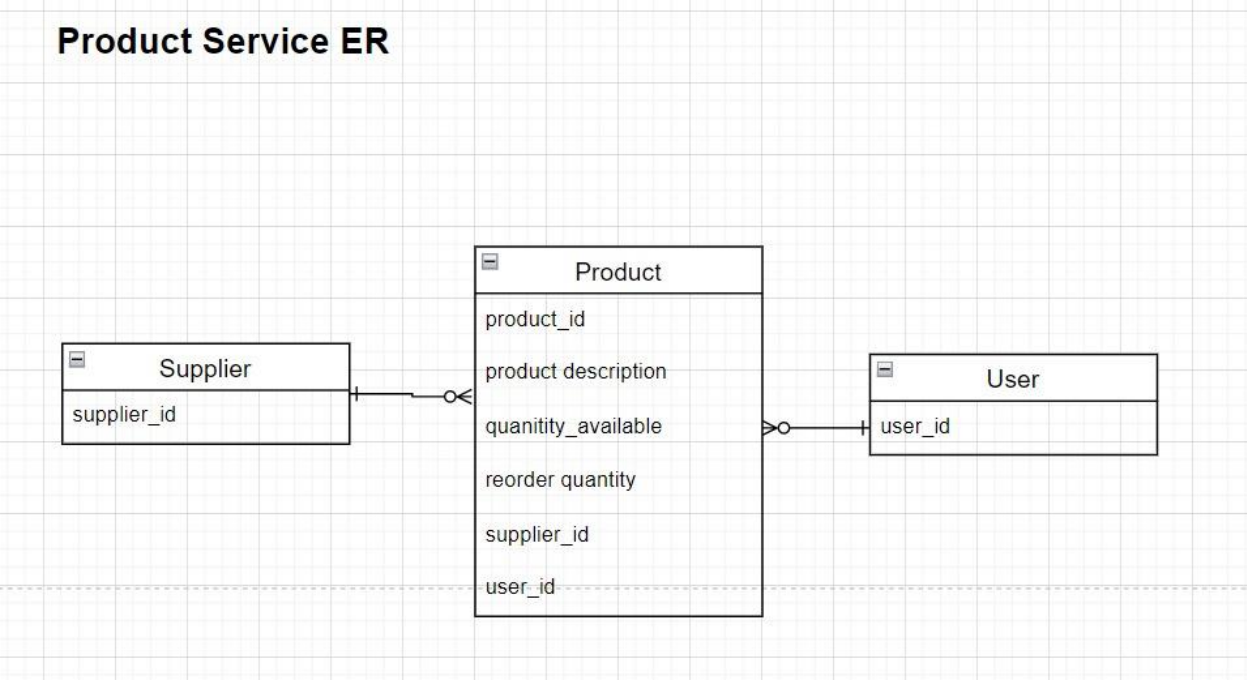
Usage:

Used by the CustomerController to perform database operations related to customers, such as saving, updating, and retrieving customer data.

ER Diagram

Entity-Relationship Diagram (ERD) here to visually represent the table structure and relationships.

Product Service



API Endpoints Description

Here's a detailed description of each endpoint in your Product API:

1. List All Products

- Endpoint: GET /products/listproducts
- Description: Retrieves a list of all products available in the inventory.

- Request Parameters: None
- Request Body: None
- Response Status Codes:
    - 200 OK: Successfully retrieved the list of products.


```
{
[
 {
  "productId": 1,
  "productDescription": "High-quality wireless mouse",
  "quantityAvailable": 150,
  "reorderQuantity": 30,
  "supplierId": 2,
  "userId": 1
 },
 {
  "productId": 2,
  "productDescription": "Ergonomic keyboard",
  "quantityAvailable": 80,
  "reorderQuantity": 20,
  "supplierId": 3,
  "userId": 1
 }
]
```

]

2. List Products for Reference

- Endpoint: GET /products/listproductsforreference
- Description: Retrieves a list of products formatted for reference use, possibly with simplified data.
- Request Parameters: None
- Request Body: None
- Response Status Codes:
    - 200 OK: Successfully retrieved the list.


```
[[
 {
  "productId": 1,
  "productDescription": "Wireless Mouse"
 },
 {
  "productId": 2,
  "productDescription": "Key
```


3. Add Product

- Endpoint: POST /products/addproduct

- Description: Adds a new product to the inventory.
- Request Parameters: None


```
{
 "productDescription": "Compact USB drive",
 "quantityAvailable": 100,
 "reorderQuantity": 25,
 "supplierId": 4,
 "userId": 1
}
```


Response Status Codes:

- 200 OK: Product successfully added.
- 400 Bad Request: Required fields are missing or invalid data was provided.


```
{
 "productId": 3,
 "productDescription": "Compact USB drive",
 "quantityAvailable": 100,
 "reorderQuantity": 25,
 "supplierId": 4,
 "userId": 1
}
```


4. Edit Product

- Endpoint: POST /products/editproduct

- Description: Updates an existing product's details in the inventory.
- Request Parameters: None


```
{
 "productId": 3,
 "productDescription": "Updated USB drive",
 "quantityAvailable": 150,
 "reorderQuantity": 30,
 "supplierId": 4,
 "userId": 1
}
```


- Response Status Codes:
    - 200 OK: Product successfully updated.
    - 400 Bad Request: Required fields are missing or invalid data was provided.


```
{
 "productId": 3,
 "productDescription": "Updated USB drive",
 "quantityAvailable": 150,
 "reorderQuantity": 30,
 "supplierId": 4,
 "userId": 1
}
```

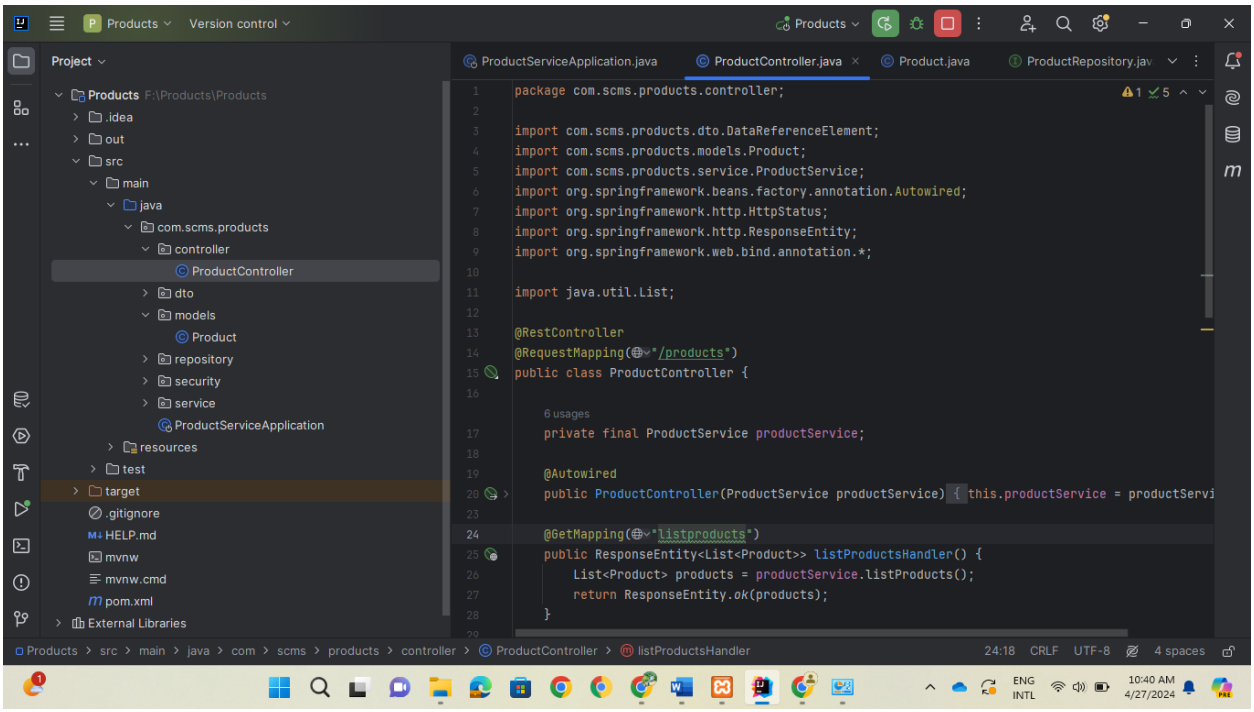
5. Delete Product

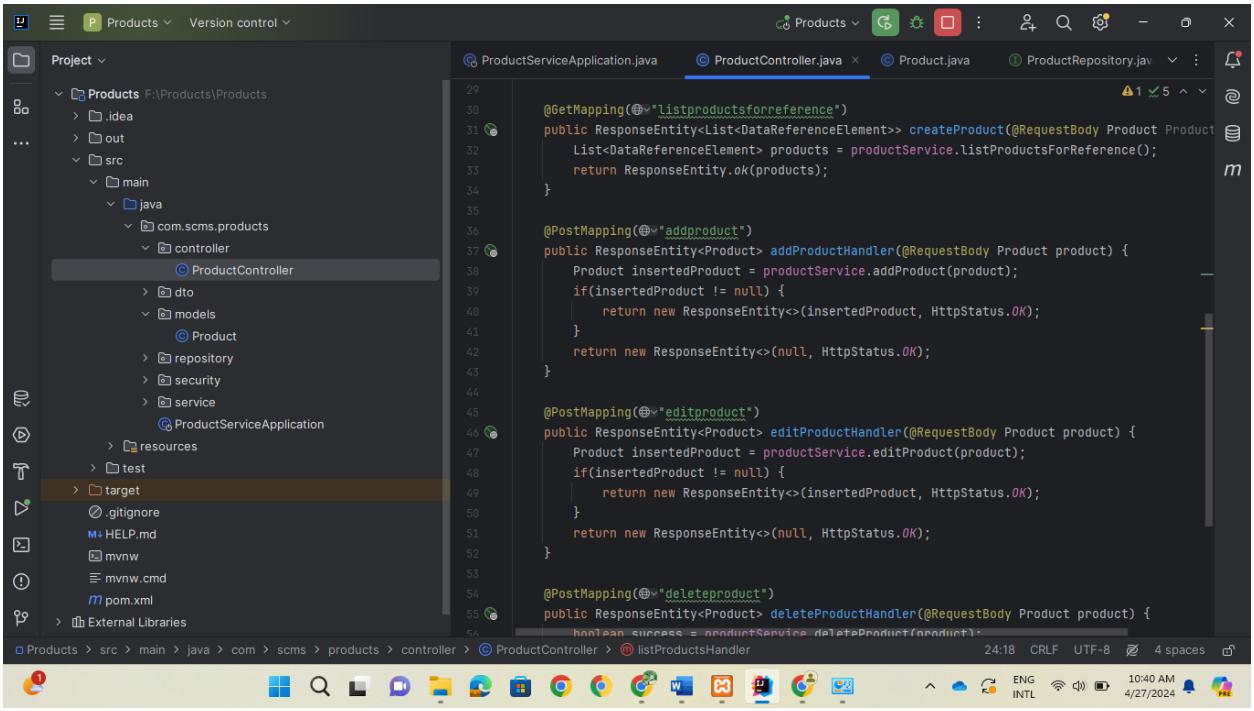
- Endpoint: POST /products/deleteproduct
- Description: Deletes a product from the inventory.
- Request Parameters: None
- Request Body:
- json
- Copy code

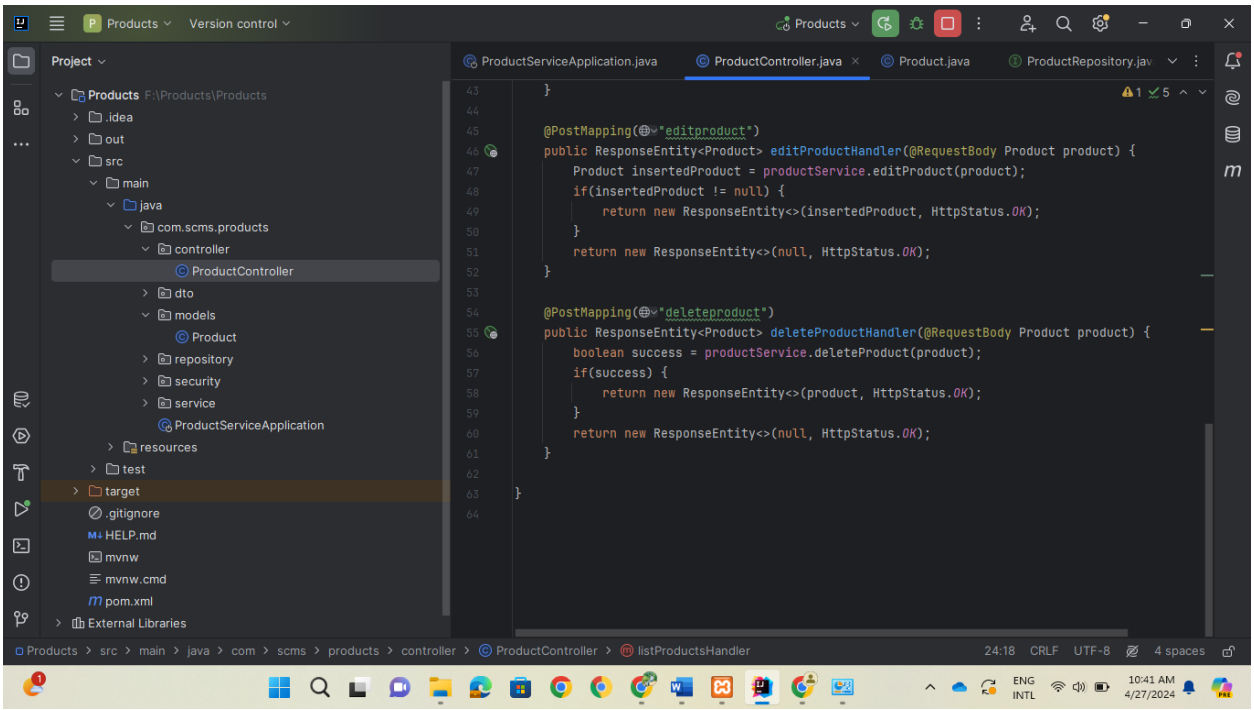
```
{
 "productId": 3
}
```

-
- Response Status Codes:
    - 200 OK: Product successfully deleted.
    - 400 Bad Request: Invalid product ID or product cannot be found.
- Response Body: None

This section provides comprehensive details for each API endpoint, including the expected inputs and outputs. The report should now clearly explain how each part of the API is used, helping developers and stakeholders understand the functionalities provided by your Product API.


Products
Version control
Project
ProductServiceApplication.java
ProductController.java
Product.java
ProductRepository.jav
Products F:\Products\Products
.idea
out
src
main
java
com.scms.products
controller
ProductController
dto
models
Product
repository
security
service
ProductServiceApplication
resources
test
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
package com.scms.products.controller;
import com.scms.products.dto.DataReferenceElement;
import com.scms.products.models.Product;
import com.scms.products.service.ProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import java.util.List;
@RestController
@RequestMapping("/products")
public class ProductController {
6 usages
private final ProductService productService;
@Autowired
public ProductController(ProductService productService) { this.productService = productServi
@GetMapping("listproducts")
public ResponseEntity<List<Product>> listProductsHandler() {
List<Product> products = productService.listProducts();
return ResponseEntity.ok(products);
}
Products › src › main › java › com › scms › products › controller › ProductController › listProductsHandler
24:18
CRLF
UTF-8
4 spaces
ENG
INTL
10:40 AM
4/27/2024


Products
Version control
Project
ProductServiceApplication.java
ProductController.java
Product.java
ProductRepository.jav
Products F:\Products\Products
.idea
out
src
main
java
com.scms.products
controller
ProductController
dto
models
Product
repository
security
service
ProductServiceApplication
resources
test
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
@GetMapping("listproductsforreference")
public ResponseEntity<List<DataReferenceElement>> createProduct(@RequestBody Product Product
List<DataReferenceElement> products = productService.listProductsForReference();
return ResponseEntity.ok(products);
@PostMapping("addproduct")
public ResponseEntity<Product> addProductHandler(@RequestBody Product product) {
Product insertedProduct = productService.addProduct(product);
if(insertedProduct != null) {
return new ResponseEntity<>(insertedProduct, HttpStatus.OK);
return new ResponseEntity<>(null, HttpStatus.OK);
@PostMapping("editproduct")
public ResponseEntity<Product> editProductHandler(@RequestBody Product product) {
Product insertedProduct = productService.editProduct(product);
if(insertedProduct != null) {
return new ResponseEntity<>(insertedProduct, HttpStatus.OK);
return new ResponseEntity<>(null, HttpStatus.OK);
@PostMapping("deleteproduct")
public ResponseEntity<Product> deleteProductHandler(@RequestBody Product product) {
Products › src › main › java › com › scms › products › controller › ProductController › listProductsHandler
24:18
CRLF
UTF-8
4 spaces
ENG
INTL
10:40 AM
4/27/2024


Products
Version control
Project
ProductServiceApplication.java
ProductController.java
Product.java
ProductRepository.jav
Products F:\Products\Products
.idea
out
src
main
java
com.scms.products
controller
ProductController
dto
models
Product
repository
security
service
ProductServiceApplication
resources
test
target
.gitignore
HELP.md
mvnw
mvnw.cmd
pom.xml
External Libraries
}
@PostMapping("editproduct")
public ResponseEntity<Product> editProductHandler(@RequestBody Product product) {
Product insertedProduct = productService.editProduct(product);
if(insertedProduct != null) {
return new ResponseEntity<>(insertedProduct, HttpStatus.OK);
return new ResponseEntity<>(null, HttpStatus.OK);
@PostMapping("deleteproduct")
public ResponseEntity<Product> deleteProductHandler(@RequestBody Product product) {
boolean success = productService.deleteProduct(product);
if(success) {
return new ResponseEntity<>(product, HttpStatus.OK);
return new ResponseEntity<>(null, HttpStatus.OK);
}
Products › src › main › java › com › scms › products › controller › ProductController › listProductsHandler
24:18
CRLF
UTF-8
4 spaces
ENG
INTL
10:41 AM
4/27/2024

Database Schema Description for Product Entity

Entity: Product

The Product class is mapped as an entity to a database table (presumably named Product by default) and includes several fields that represent columns in the database. Below are the details of each field along with their constraints and relationships:

Table Name: Product

Fields:

1. productId (Primary Key)
    - Type: long
    - Annotations:
        - @Id: Specifies that this field is the primary key.
        - @GeneratedValue(strategy = GenerationType.IDENTITY): Specifies that the primary key value is automatically generated by the database, incrementing by default.
    - Description: Unique identifier for each product.

2. productDescription
    - Type: String
    - Annotations:
        - @Column(nullable = false): Indicates that this column cannot have a null value.
    - Description: Descriptive text detailing the product features or specifications. This field is mandatory.
3. quantityAvailable
    - Type: int
    - Annotations:
        - @Column(name = "quantity_available"): Specifies the exact column name in the database.
    - Description: Represents the number of units of the product currently available in inventory.
4. reorderQuantity
    - Type: int
    - Annotations: None
    - Description: The threshold quantity at which an order should be placed to replenish the stock.
5. supplierId
    - Type: int
    - Annotations: None
    - Description: Identifier for the supplier of the product. This could be used to join with a Supplier table in a relational database context.

6. userId
    - Type: int
    - Annotations: None
    - Description: Identifier for the user associated with the product record, potentially for tracking who added or last updated the product details.

Constraints and Indexes:

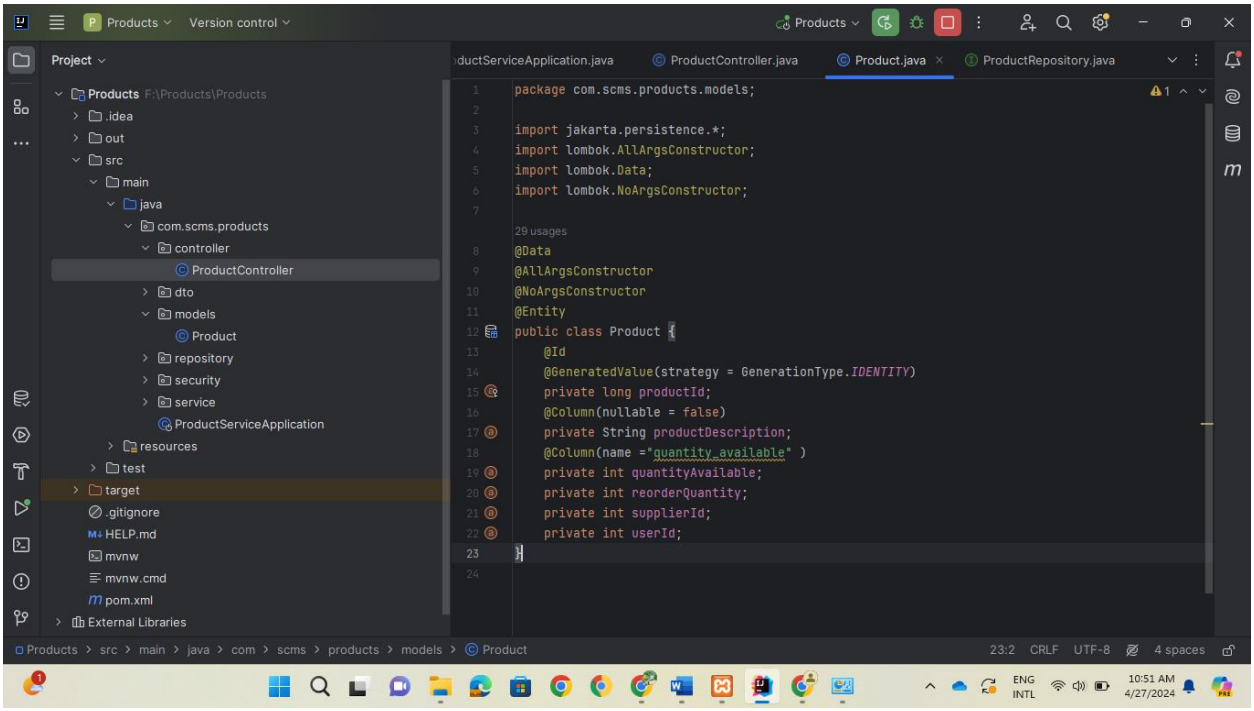
- The primary key (productId) ensures uniqueness across the product records.
- The productDescription field is marked as non-nullable, ensuring that every product record must have a description.

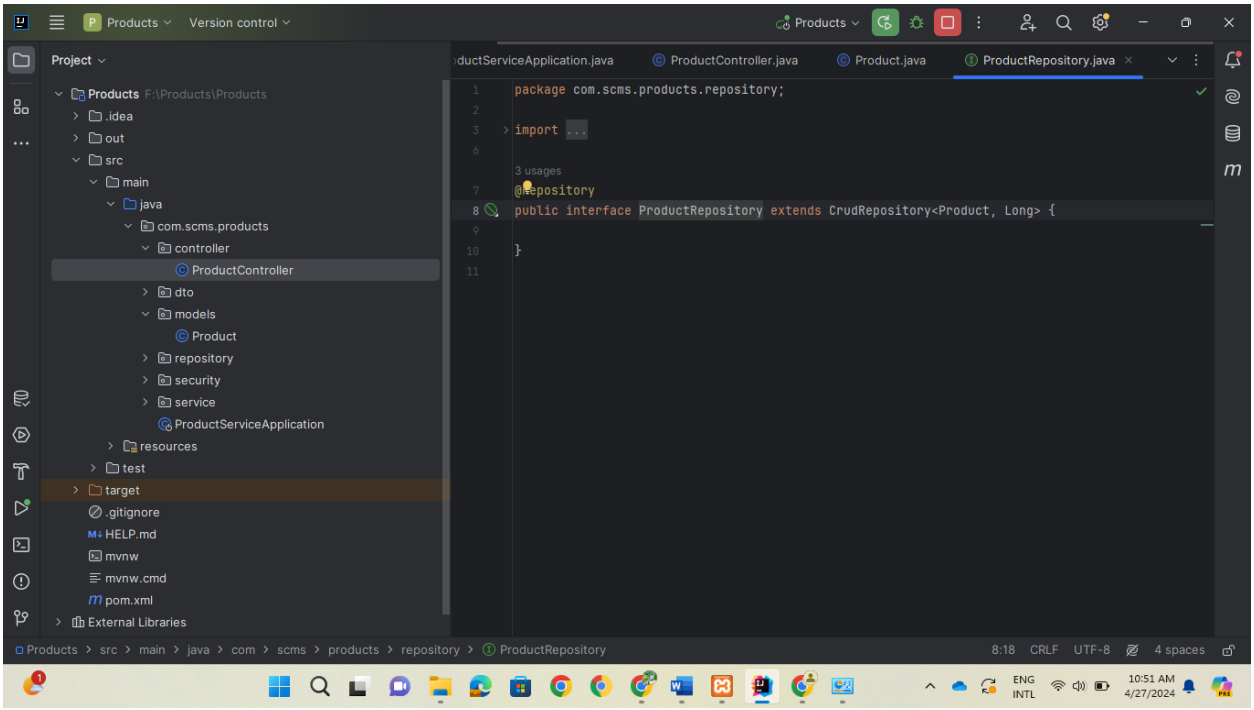
Relationships:

- Supplier Relationships: If there exists a Supplier entity, supplierId would typically be used as a foreign key linking Product to Supplier.
- User Relationships: Similar to Supplier, if user data is stored in another table, userId could link Product to a User entity, indicating ownership or responsibility.

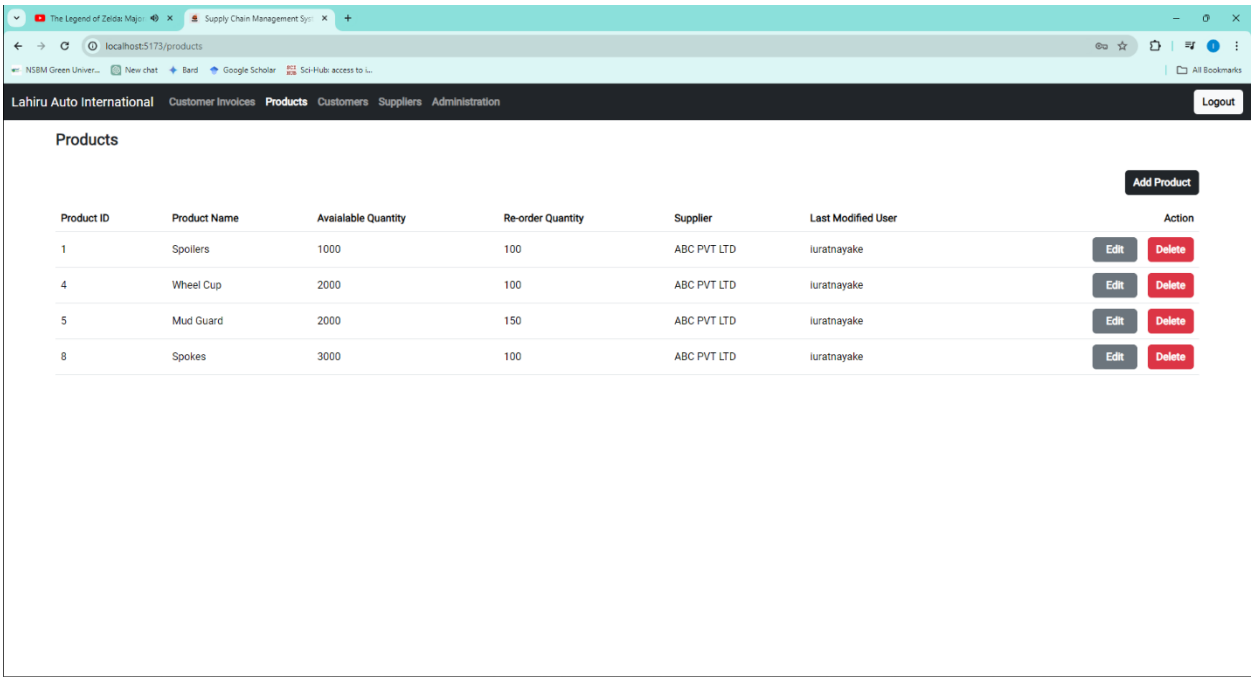
Notes:

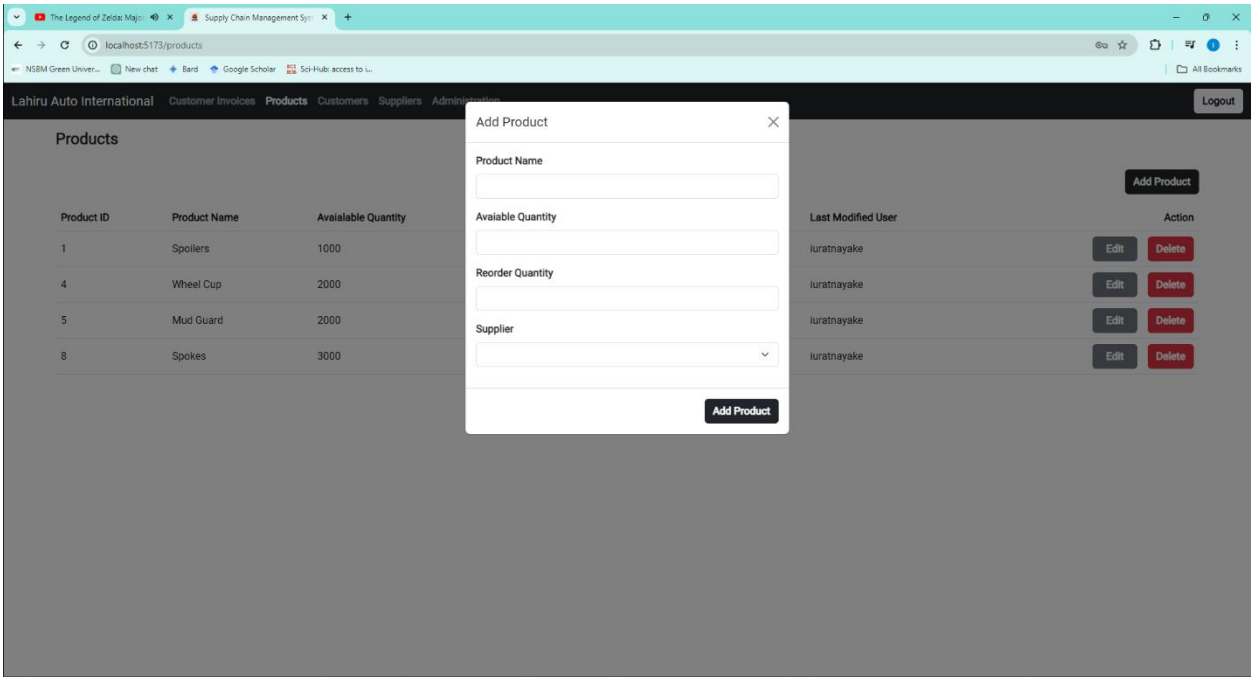
- The schema design assumes that the relationships are managed through IDs (supplierId, userId). If using a full ORM setup, these might be replaced or augmented with actual entity mappings using annotations like @ManyToOne or @JoinColumn.

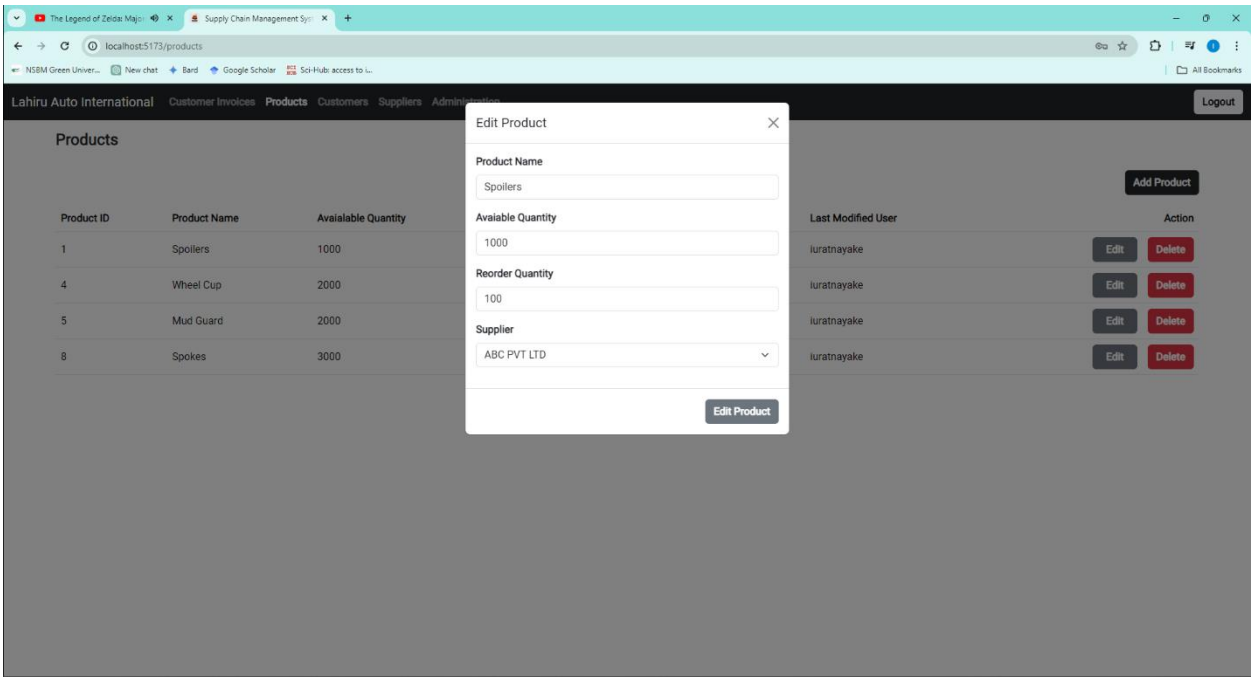


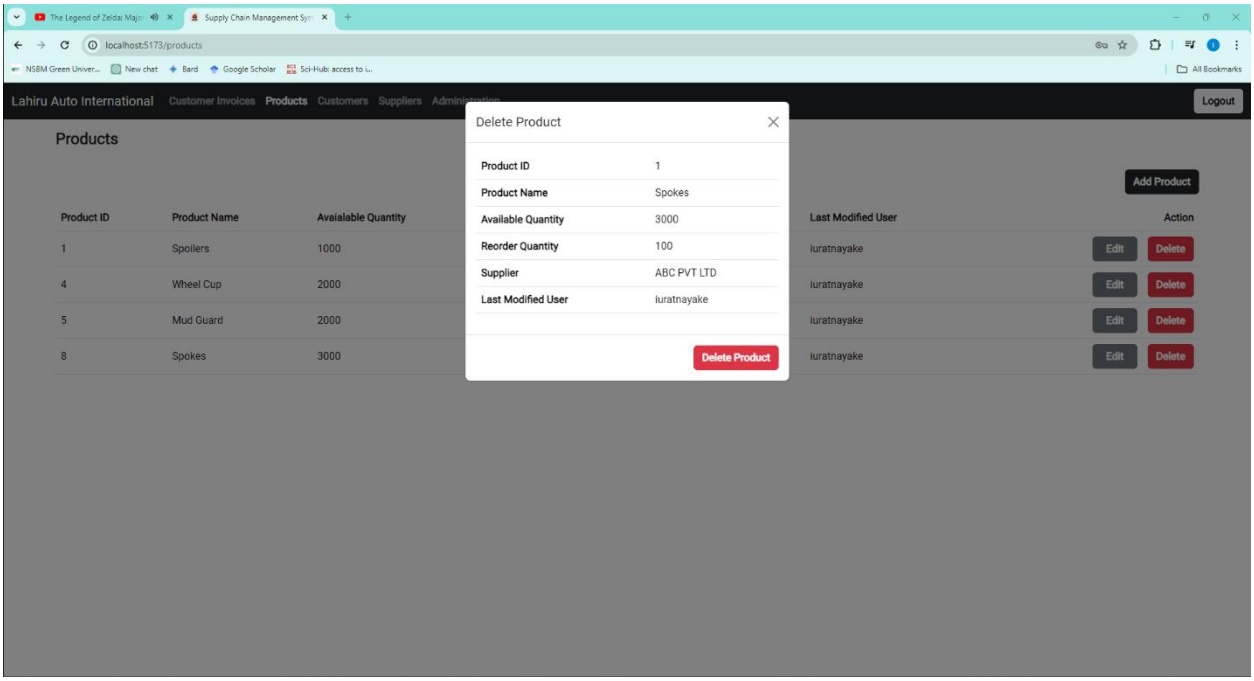


FrontEnd Design for Product API


localhost:5173/products
Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Administration
Logout
Products
Add Product
Product ID
Product Name
Avaialable Quantity
Re-order Quantity
Supplier
Last Modified User
Action
1
Spoilers
1000
100
ABC PVT LTD
iuratnayake
Edit
Delete
4
Wheel Cup
2000
100
ABC PVT LTD
iuratnayake
Edit
Delete
5
Mud Guard
2000
150
ABC PVT LTD
iuratnayake
Edit
Delete
8
Spokes
3000
100
ABC PVT LTD
iuratnayake
Edit
Delete


localhost:5173/products
Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Logout
Products
Add Product
Product ID
Product Name
Avaialable Quantity
Last Modified User
Action
1
Spoilers
1000
iuratnayake
Edit
Delete
4
Wheel Cup
2000
iuratnayake
Edit
Delete
5
Mud Guard
2000
iuratnayake
Edit
Delete
8
Spokes
3000
iuratnayake
Edit
Delete
Add Product
Product Name
Avaible Quantity
Reorder Quantity
Supplier
Add Product

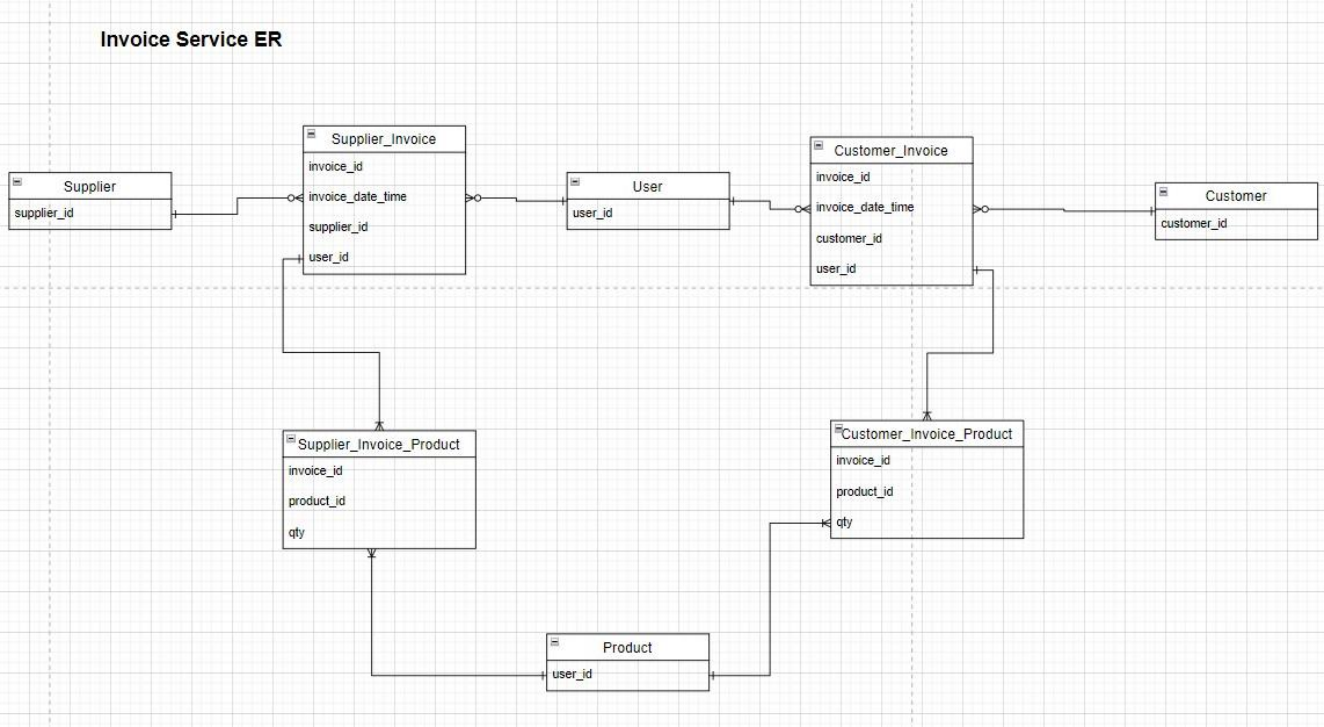

localhost:5173/products
Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Logout
Products
Add Product
Product ID
Product Name
Avaialable Quantity
Last Modified User
Action
1
Spoilers
1000
iuratnayake
Edit
Delete
4
Wheel Cup
2000
iuratnayake
5
Mud Guard
2000
iuratnayake
8
Spokes
3000
iuratnayake
Edit Product
Product Name
Spoilers
Avaiable Quantity
1000
Reorder Quantity
100
Supplier
ABC PVT LTD
Edit Product


localhost:5173/products
Lahiru Auto International
Customer Invoices
Products
Customers
Suppliers
Logout
Products
Add Product
Product ID
Product Name
Avaialable Quantity
Last Modified User
Action
1
Spoilers
1000
iuratnayake
Edit
Delete
4
Wheel Cup
2000
iuratnayake
5
Mud Guard
2000
iuratnayake
8
Spokes
3000
iuratnayake
Delete Product
Product ID
1
Product Name
Spokes
Available Quantity
3000
Reorder Quantity
100
Supplier
ABC PVT LTD
Last Modified User
iuratnayake
Delete Product

Invoice Service

1. System Architecture

The Lahiru Auto Intanational POS system architecture adopts a microservices approach to ensure scalability, modularity, and fault tolerance. By decomposing the application into smaller, independent services, it enables flexible development, deployment, and operation in dynamic retail environments.



2. Microservices

2.1 Customer Invoice Service

Functionality

- Manages customer invoices, including creation, retrieval, and deletion.
- Handles operations related to customer invoice products, such as associating products with invoices.

Responsibilities:

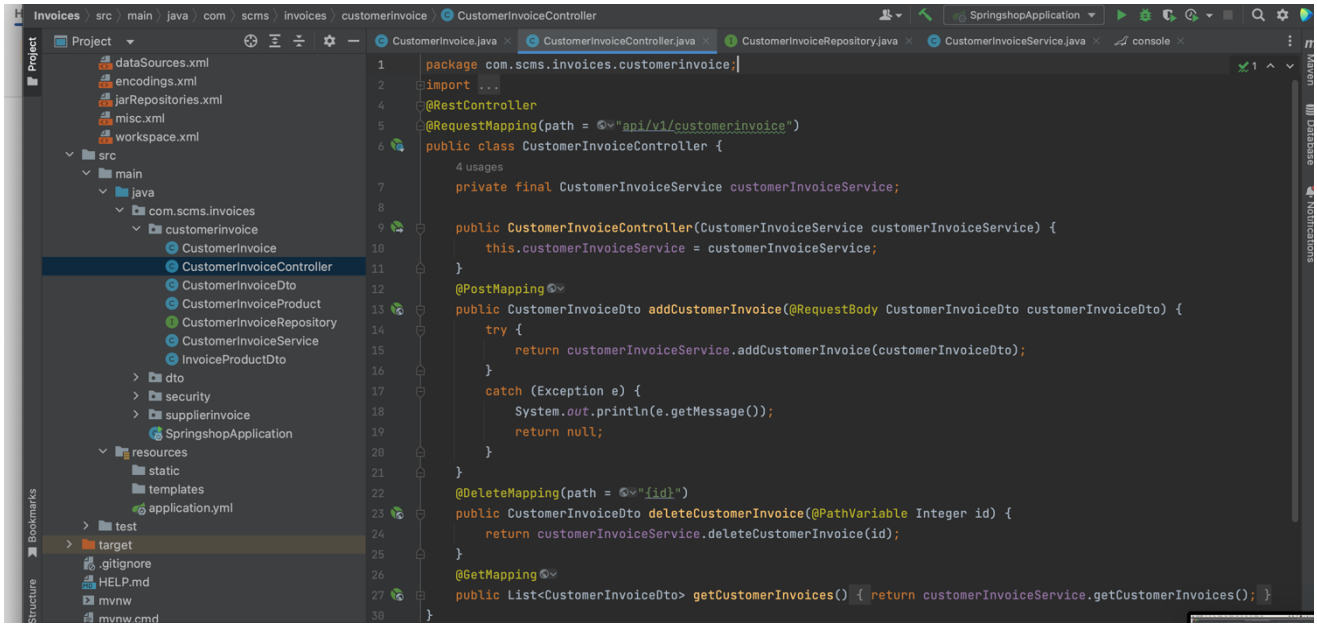
- Accepts requests from clients to create, retrieve, or delete customer invoices.
- Interacts with the database to persist customer invoice data.
- Ensures data integrity and consistency when associating products with invoices.

Endpoints:

- Create Customer Invoice: POST /api/v1/customerinvoice
- Retrieve All Customer Invoices: GET /api/v1/customerinvoice
- Delete Customer Invoice: DELETE /api/v1/customerinvoice/{id}

Interactions:

- Communicates with the database to store and retrieve customer invoice information.
- Receives requests from clients and provides responses accordingly, ensuring smooth interaction.



2.2. Supplier Invoice Service

Functionality:

- Manages supplier invoices, including creation, retrieval, and deletion.

- Handles operations related to supplier invoice products, such as associating products with invoices.

Responsibilities:

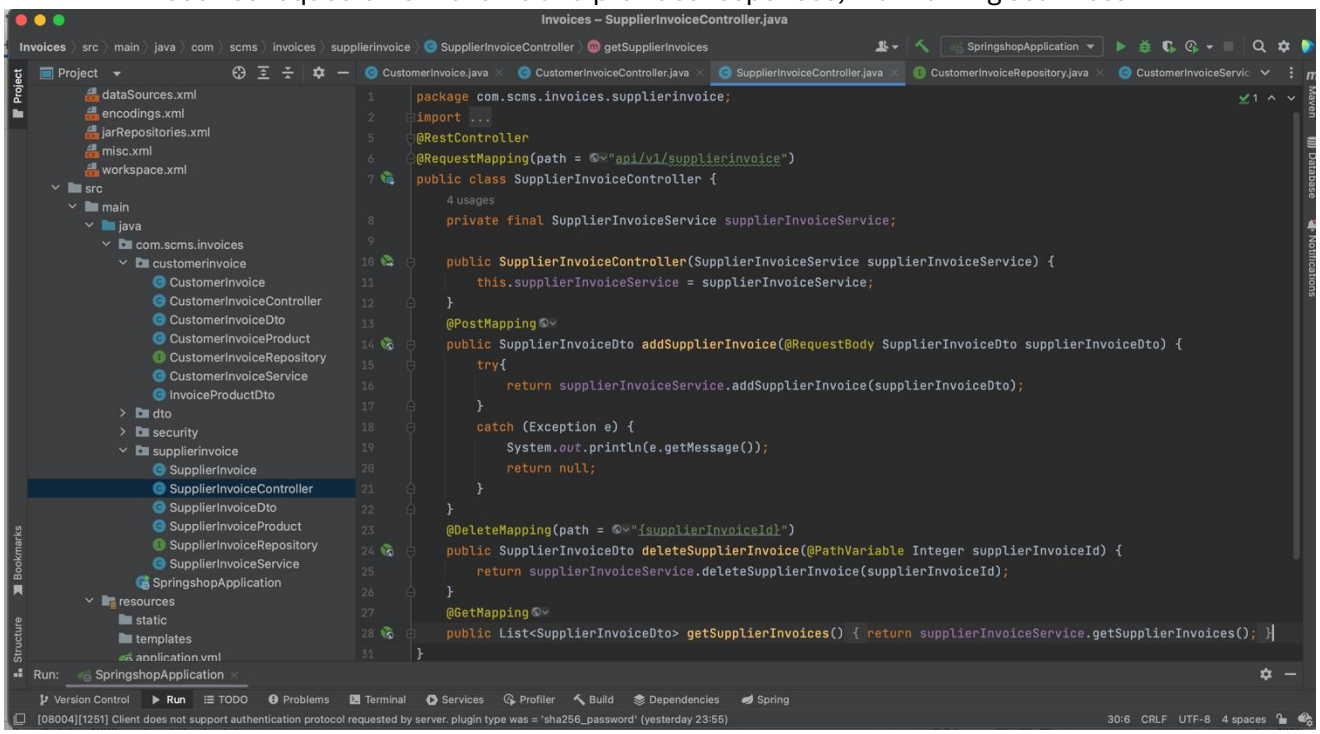
- Accepts requests from clients to create, retrieve, or delete supplier invoices.
- Interacts with the database to persist supplier invoice data.
- Ensures data consistency and accuracy when managing supplier invoice products.

Endpoints:

- Create Supplier Invoice: POST /api/v1/supplierinvoice
- Retrieve All Supplier Invoices: GET /api/v1/supplierinvoice
- Delete Supplier Invoice: DELETE /api/v1/supplierinvoice/{id}

Interactions:

- Communicates with the database to store and retrieve supplier invoice information.
- Receives requests from clients and provides responses, maintaining seamless



4. Database Schema

The database schema of the Point of Sale (POS) System plays a crucial role in storing and organizing data related to customer and supplier invoices. Let's delve deeper into the structure and purpose of each entity in the schema:

CustomerInvoice:

customerInvoiceId (Primary Key)

customerId

userId

invoiceDate

CustomerInvoiceProduct:

customerInvoiceId (Foreign Key)

productId

quantity

4.1. Customer Invoice Entity

- **customerInvoiceId:** A unique identifier for each customer invoice. This serves as the primary key of the table, ensuring each record's uniqueness.
- **customerId:** Represents the ID of the customer associated with the invoice. This attribute establishes a relationship between the invoice and the customer who made the purchase.
- **userId:** Indicates the ID of the user who created the invoice. This attribute helps in tracking the user responsible for generating the invoice.
- **invoiceDate:** Stores the date when the invoice was created. This attribute captures the timestamp of the invoice creation for reference and reporting purposes.

4.2. Customer Invoice Product Entity

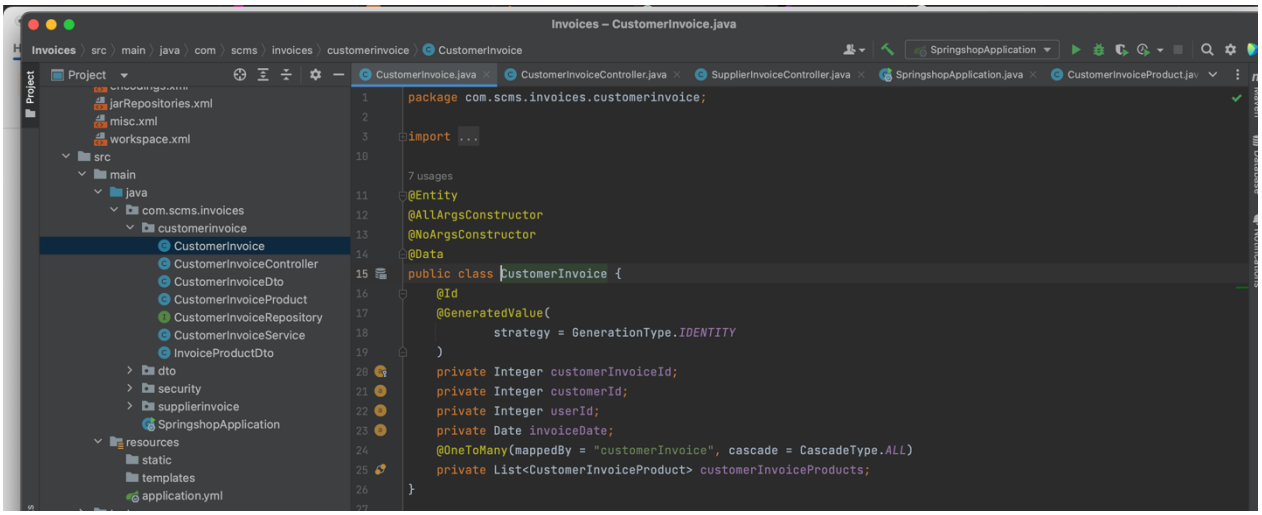
- **customerInvoice:** A foreign key that establishes a many-to-one relationship with the Customer Invoice entity. This relationship links each product to its corresponding invoice.
- **productId**: Represents the ID of the product associated with the invoice. This attribute identifies the product included in the invoice.
- **qty:** Indicates the quantity of the product included in the invoice. This attribute specifies how many units of the product were purchased by the customer.

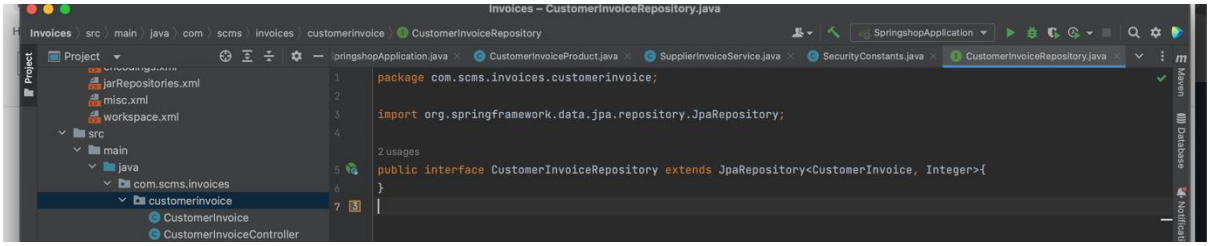
4.3. Supplier Invoice Entity

- **supplierInvoiceId:** A unique identifier for each supplier invoice. Similar to the customer invoice, this attribute serves as the primary key of the table.
- **supplierId:** Represents the ID of the supplier associated with the invoice. This attribute establishes a relationship between the invoice and the supplier who provided the goods or services.
- **userId:** Indicates the ID of the user who created the supplier invoice. This attribute helps in tracking the user responsible for generating the invoice.
- **invoiceDate**: Stores the date when the supplier invoice was created. Similar to the customer invoice, this attribute captures the timestamp of the invoice creation.

4.4. Supplier Invoice Product Entity

- **supplierInvoice:** A foreign key that establishes a many-to-one relationship with the Supplier Invoice entity. This relationship links each product to its corresponding supplier invoice.
- **productId:** Represents the ID of the product associated with the supplier invoice. This attribute identifies the product supplied by the supplier.
- **qty:** Indicates the quantity of the product included in the supplier invoice. This attribute specifies how many units of the product were supplied by the supplier.





5. RESTful API Endpoints

The Point of Sale (POS) System provides a set of RESTful API endpoints to enable communication between clients and microservices. These endpoints facilitate the creation, retrieval, and deletion of customer and supplier invoices. Let's explore each endpoint in detail, along with its functionality and usage:

Customer Invoice Service Endpoints:

1. Create Customer Invoice:

- Endpoint: POST /api/v1/customerinvoice
- Functionality: This endpoint allows clients to create a new customer invoice by sending a POST request with the necessary invoice details, including customer ID, user ID, and invoice date.
- Usage: Clients can submit a request to this endpoint with the required JSON payload containing invoice information. Upon successful creation, the endpoint returns the details of the newly created invoice.

2. Retrieve All Customer Invoices:

- Endpoint: GET /api/v1/customerinvoice
- Functionality: This endpoint retrieves all existing customer invoices stored in the system.
- Usage: Clients can send a GET request to this endpoint to fetch a list of all customer invoices. The response contains details of each invoice, such as invoice ID, customer ID, user ID, and invoice date.

3. Delete Customer Invoice:

- Endpoint: DELETE /api/v1/customerinvoice/{id}
- Functionality: Clients can use this endpoint to delete a specific customer invoice by providing its ID in the request path.
- Usage: To delete a customer invoice, clients send a DELETE request to this endpoint with the ID of the invoice to be deleted as part of the URL path. Upon successful deletion, the endpoint returns a confirmation response.

Supplier Invoice Service Endpoints:

1. Create Supplier Invoice:

- Endpoint: POST /api/v1/supplierinvoice
- Functionality: This endpoint enables clients to create a new supplier invoice by sending a POST request with the required invoice details, such as supplier ID, user ID, and invoice date.
- Usage: Clients can submit a request to this endpoint with the necessary invoice information in the request body. Upon successful creation, the endpoint returns the details of the newly created supplier invoice.

2. Retrieve All Supplier Invoices:

- Endpoint: GET /api/v1/supplierinvoice
- Functionality: This endpoint retrieves all existing supplier invoices stored in the system.
- Usage: Clients can send a GET request to this endpoint to fetch a list of all supplier invoices. The response contains details of each invoice, including invoice ID, supplier ID, user ID, and invoice date.

3. Delete Supplier Invoice:

- Endpoint: DELETE /api/v1/supplierinvoice/{id}
- Functionality: Clients can use this endpoint to delete a specific supplier invoice by providing its ID in the request path.
- Usage: To delete a supplier invoice, clients send a DELETE request to this endpoint with the ID of the invoice to be deleted as part of the URL path. Upon successful deletion, the endpoint returns a confirmation response.

6. Service Interaction Diagram

The service interaction diagram illustrates the communication flow between the client, microservices, and the database in the Point of Sale (POS) System during common scenarios involving customer and supplier invoices. Let's explore the interaction diagram in detail, incorporating the functionality of both the Customer Invoice Service and Supplier Invoice Service:

Creating a Customer Invoice

1. **Client**:
    - Initiates the request to create a new customer invoice by sending a POST request to the **/api/v1/customerinvoice** endpoint of the Customer Invoice Service.
2. **Customer Invoice Service**:
    - Receives the POST request from the client at the designated endpoint.
    - Processes the request payload, which contains the details of the new customer invoice.
    - Interacts with the database to persist the customer invoice data, including customer ID, user ID, and invoice date.
    - Generates a response indicating the success or failure of the operation and includes details of the newly created invoice, if applicable.
3. **Database**:
    - Stores the customer invoice data received from the Customer Invoice Service, ensuring data integrity and consistency.
4. **Client**:
    - Receives the response from the Customer Invoice Service, indicating the outcome of the operation.
    - If successful, the client proceeds with further actions based on the response. If unsuccessful, appropriate error handling mechanisms are employed.

Scenario: Deleting a Supplier Invoice

1. **Client**:
    - Sends a DELETE request to the **/api/v1/supplierinvoice/{id}** endpoint of the Supplier Invoice Service, specifying the ID of the supplier invoice to be deleted.
2. **Supplier Invoice Service**:
    - Receives the DELETE request from the client at the designated endpoint.
    - Identifies the supplier invoice to be deleted based on the provided ID.
    - Interacts with the database to remove the specified supplier invoice and its associated product details.

- Generates a response indicating the success or failure of the deletion operation.

3. **Database**:
    - Deletes the specified supplier invoice and associated product details from the database upon receiving the deletion request from the Supplier Invoice Service.
4. **Client**:
    - Receives the response from the Supplier Invoice Service, indicating the outcome of the deletion operation.
    - Reacts accordingly based on the response, proceeding with subsequent actions or error handling as necessary.

7. Deployment Strategy

The deployment strategy outlines how the Point of Sale (POS) System, comprising customer and supplier invoice services, is deployed and managed in a production environment. Let's delve into the deployment strategy, considering factors such as scalability, fault tolerance, and ease of maintenance:

1. Microservices Deployment:

- **Containerization**: Both the Customer Invoice Service and Supplier Invoice Service are packaged as container images using Docker. Containerization ensures consistency across different environments and simplifies deployment.
- **Orchestration**: Containerized microservices are deployed and managed using container orchestration platforms like Kubernetes. Kubernetes automates deployment, scaling, and management of containerized applications, ensuring high availability and fault tolerance.

2. High Availability and Scalability:

- **Replication**: Multiple instances of each microservice are deployed across Kubernetes clusters to ensure high availability. Kubernetes automatically manages load balancing and failover, distributing traffic evenly among healthy instances.
- **Horizontal Scaling**: Kubernetes enables horizontal scaling of microservices by dynamically adjusting the number of running instances based on resource utilization and incoming traffic. This ensures that the system can handle varying workloads efficiently.

3. Service Discovery and Load Balancing:

- **Service Discovery**: Kubernetes provides built-in service discovery mechanisms, allowing microservices to discover and communicate with each other using service names rather than hardcoded IP addresses. This simplifies inter-service communication and promotes decoupling.
- **Load Balancing**: Kubernetes automatically performs load balancing across multiple instances of microservices, distributing incoming requests evenly to ensure optimal performance and resource utilization.

4. Continuous Integration and Deployment (CI/CD):

- **Pipeline Automation**: CI/CD pipelines are implemented using tools like Jenkins or GitLab CI/CD. These pipelines automate the build, testing, and deployment process, ensuring rapid and reliable delivery of updates to the production environment.

- **Automated Testing**: Comprehensive automated testing, including unit tests, integration tests, and end-to-end tests, is integrated into the CI/CD pipeline to maintain code quality and prevent regressions.

7. Security:

- **Network Policies**: Kubernetes network policies are configured to enforce fine-grained control over network traffic between microservices, ensuring secure communication and preventing unauthorized access.
- **Secrets Management**: Kubernetes secrets are utilized to securely store sensitive information such as database credentials, API keys, and TLS certificates, minimizing the risk of exposure and unauthorized access.