



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.....	2
Team Overview.....	2
Member contribution	2
Challenges Faced and how they were overcome.....	2
Architecture	2
Microservices	3
Administration.....	3
Authentication.....	5
Authentication for other Microservices	7
Authorization	7
User Management.....	7
Supplier Microservice	12
Customer Microservice.....	18
Front End.....	18
ER Diagram.....	22
Backend Code	23
Product Service	30
Invoice Service.....	44

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 Jeewanthy	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.

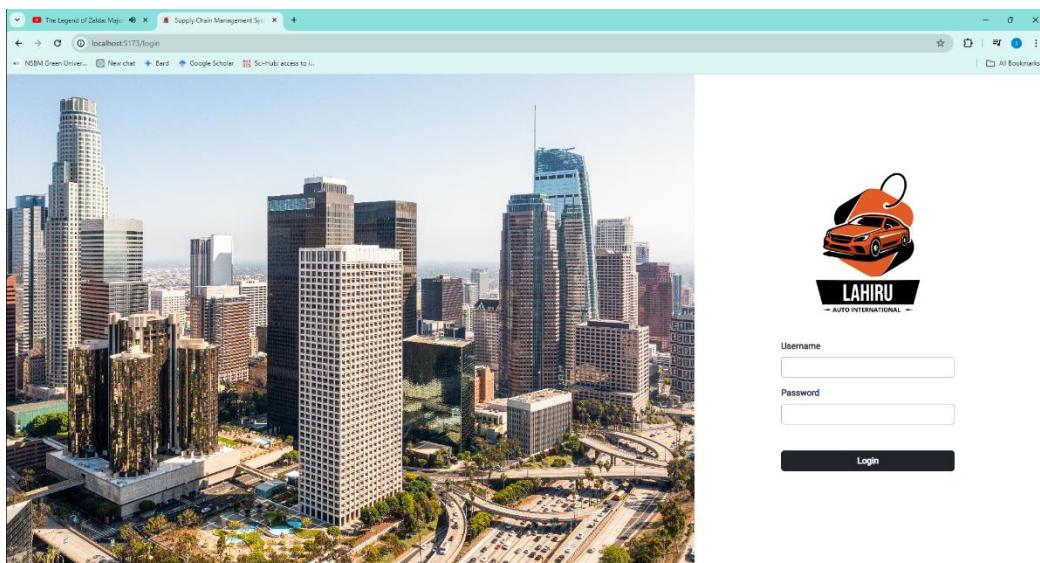
For the front end an MVP architecture was used. The front end was built using React and React Router. The entire system was designed using the Rest Architectural pattern.

Microservices

Administration

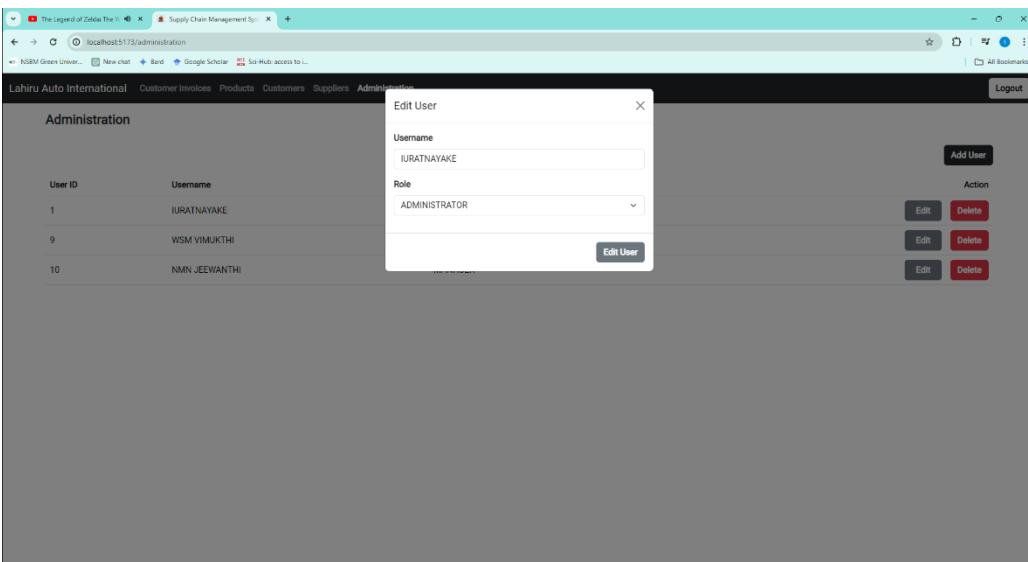
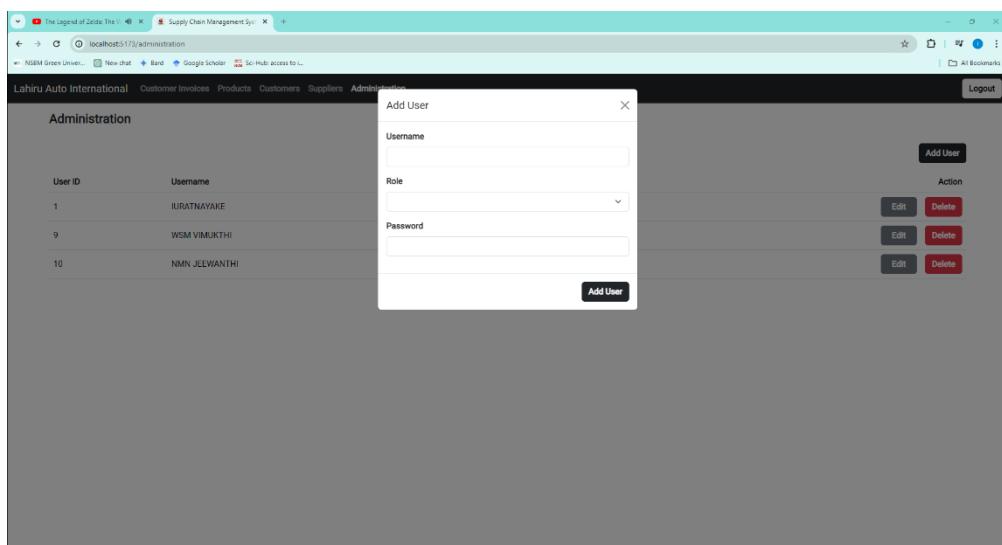
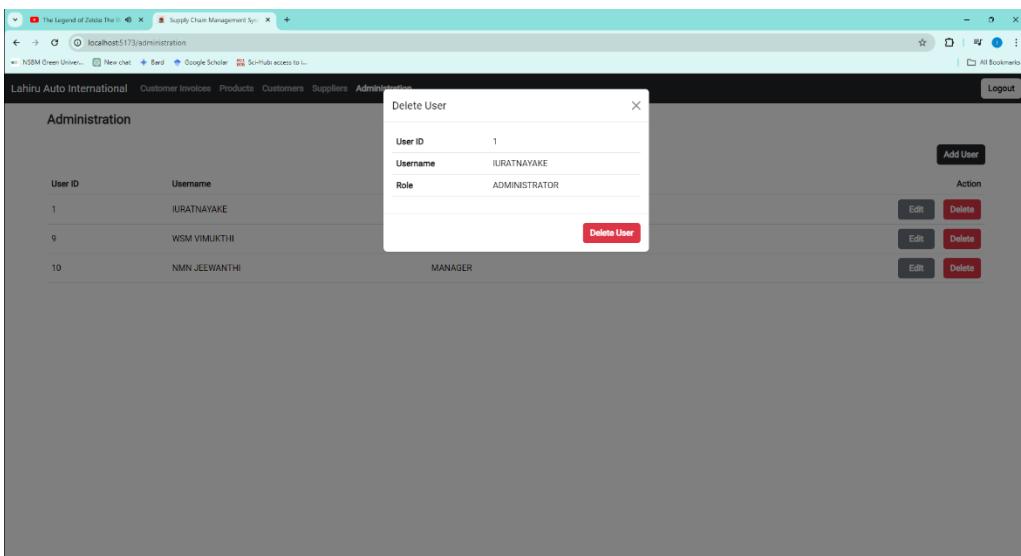
The Administration microservice is responsible for Authentication, Authorization and User Management. It provides functionality for login users into the system, checking if the user is currently logged in, authorization of actions, adding new users, deleting, and updating users and fetching user details.

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.

A screenshot of a web browser showing the administration page of the supply chain management system. The top navigation bar includes links for 'Customer Invoices', 'Products', 'Customers', 'Suppliers', and 'Administration'. The 'Administration' link is highlighted. The main content area is titled 'Administration' and displays a table of user data. The table has columns for 'User ID', 'Username', 'Role', and 'Action'. There are three rows of data:

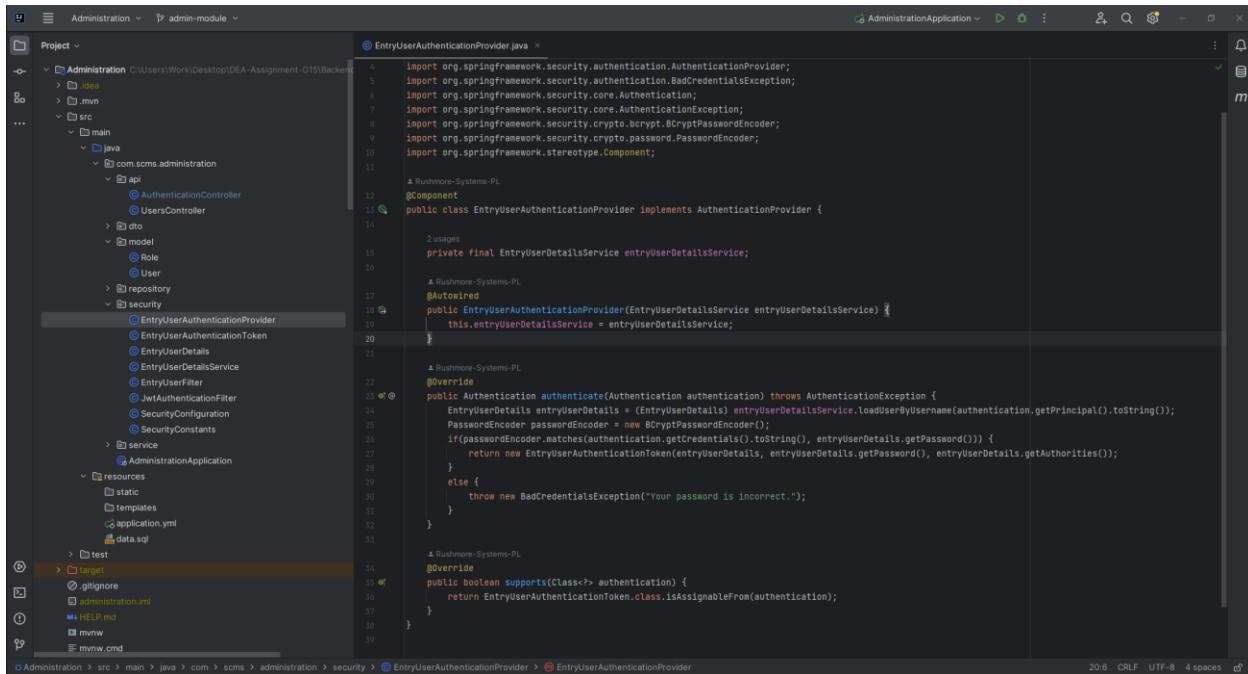
User ID	Username	Role	Action
1	JURATNAYAKE	ADMINISTRATOR	Edit Delete
9	WSM VIMUKTHI	CASHIER	Edit Delete
10	MNN JEEWANTHI	MANAGER	Edit Delete

A small 'Add User' button is located at the top right of the table. A 'Logout' button is visible in the top right corner of the browser window.



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.



The screenshot shows the IntelliJ IDEA interface with the 'EntryUserAuthenticationProvider.java' file open in the editor. The code implements the `AuthenticationProvider` interface, using `BCryptPasswordEncoder` for password hashing. It includes annotations like `@Component`, `@Autowired`, and `@Override`. The code handles password matching and returns a `BadCredentialsException` if the password is incorrect. The code editor shows syntax highlighting and line numbers from 1 to 39.

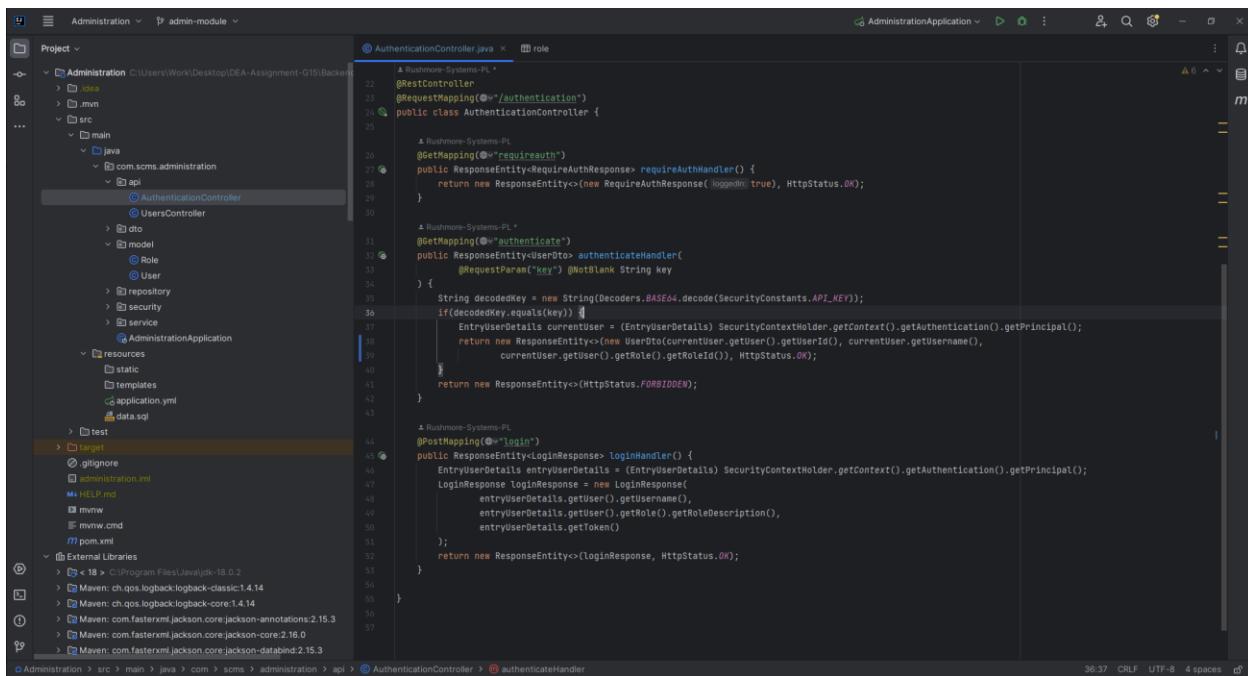
```
import org.springframework.security.authentication.AuthenticationProvider;
import org.springframework.security.authentication.BadCredentialsException;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Component;

public class EntryUserAuthenticationProvider implements AuthenticationProvider {

    private final EntryUserDetailsService entryUserDetailsService;

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        EntryUserDetails entryUserDetails = (EntryUserDetails) entryUserDetailsService.loadUserByUsername(authentication.getPrincipal().toString());
        PasswordEncoder passwordEncoder = new BCryptPasswordEncoder();
        if(passwordEncoder.matches(authentication.getredentials().toString(), entryUserDetails.getPassword())) {
            return new EntryUserAuthenticationToken(entryUserDetails, entryUserDetails.getPassword(), entryUserDetails.getAuthorities());
        }
        else {
            throw new BadCredentialsException("Your password is incorrect.");
        }
    }

    @Override
    public boolean supports(Class<?> authentication) {
        return EntryUserAuthenticationToken.class.isAssignableFrom(authentication);
    }
}
```



The screenshot shows the IntelliJ IDEA interface with the 'AuthenticationController.java' file open in the editor. The controller uses `RestController` and `RequestMapping` annotations to handle requests. It has two main endpoints: one for requiring authentication and another for authenticating users. The 'authenticate' endpoint takes a key parameter and checks it against a decoded API key. If successful, it returns a user's role and ID. The 'login' endpoint handles login responses and returns a user's details. The code editor shows syntax highlighting and line numbers from 18 to 97.

```
@RestController
@RequestMapping("/authentication")
public class AuthenticationController {

    @GetMapping("requireAuth")
    public ResponseEntity<RequireAuthResponse> requireAuthHandler() {
        return new ResponseEntity<RequireAuthResponse>(new RequireAuthResponse(loggedIn: true), HttpStatus.OK);
    }

    @GetMapping("authenticate")
    public ResponseEntity<User> authenticateHandler(
        @RequestParam("key") @NotBlank String key
    ) {
        String decodedKey = new String(Base64.getDecoder().decode(SecurityConstants.API_KEY));
        if(decodedKey.equals(key)) {
            EntryUserDetails currentUser = (EntryUserDetails) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
            return new ResponseEntity<User>(new User(currentUser.getUserId(), currentUser.getUsername(),
                currentUser.getUser().getRole().getRoleName(), currentUser.getRole().getRoleDescription(),
                currentUser.getRole().getRoleDescription(), currentUser.getRole().getRoleName(),
                currentUser.getRole().getRoleName()), HttpStatus.OK);
        }
        return new ResponseEntity<User>(HttpStatus.FORBIDDEN);
    }

    @PostMapping("login")
    public ResponseEntity<LoginResponse> loginHandler() {
        EntryUserDetails entryUserDetails = (EntryUserDetails) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        LoginResponse loginResponse = new LoginResponse(
            entryUserDetails.getUsername(),
            entryUserDetails.getUser().getUsername(),
            entryUserDetails.getUser().getRole().getRoleName(),
            entryUserDetails.getRole().getRoleName()
        );
        return new ResponseEntity<LoginResponse>(loginResponse, HttpStatus.OK);
    }
}
```

```

public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private final EntryUserDetailsService entryUserDetailsService;

    public JwtAuthenticationFilter(EntryUserDetailsService entryUserDetailsService) {
        this.entryUserDetailsService = entryUserDetailsService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        try {
            String authHeader = request.getHeader("Authorization");
            if (StringUtils.hasText(authHeader) && authHeader.startsWith("Bearer ")) {
                String token = authHeader.substring(7);
                if(StringUtils.hasText(token)) {
                    Claims claims = Jwts.parser()
                        .verifyWith(Keys.hmacSha256For(Decoders.BASE64.decode(JWT_SECRET)))
                        .build()
                        .parseSignedClaims(token)
                        .getPayload();
                    String username = claims.getSubject();
                    if(StringUtils.hasText(username)) {
                        if(!request.getServletPath().equals("/authentication/requireauth")) {
                            EntryUserDetails userdetails = (EntryUserDetails) entryUserDetailsService
                                .loadUserByUsername(username);
                            SecurityContextHolder.getContext()
                                .setAuthentication(new EntryUserAuthenticationToken(
                                    userdetails,
                                    userdetails.getPassword(),
                                    userdetails.getAuthorities()
                                ));
                            filterChain.doFilter(request, response);
                        } else {
                            throw new BadCredentialsException("Credentials absent from token.");
                        }
                    } else {
                        throw new BadCredentialsException("Token not found.");
                    }
                }
            }
        } catch(BadCredentialsException e) {
            response.setStatus(HttpStatus.FORBIDDEN.value());
            response.setContentType(MediaType.APPLICATION_JSON_VALUE);
            Error error = new Error("username", e.getMessage());
            ObjectMapper objectMapper = new ObjectMapper();
            String errorResponseString = objectMapper.writeValueAsString(error);
            response.getWriter().write(errorResponseString);
        }
    }
}

```

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

```

public class EntryUserFilter extends OncePerRequestFilter {
    private final EntryUserAuthenticationProvider entryUserAuthenticationProvider;

    public EntryUserFilter(EntryUserAuthenticationProvider entryUserAuthenticationProvider) {
        this.entryUserAuthenticationProvider = entryUserAuthenticationProvider;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
        try {
            if(request.getMethod().equals("POST")) {
                throw new HttpMethodNotSupportedException("Only supports POST requests.");
            }
            String username = request.getParameter("username");
            String password = request.getParameter("password");
            Authentication authentication = new EntryUserAuthenticationToken(username, password);
            Authentication authenticatedUser = entryUserAuthenticationProvider.authenticate(authentication);
            EntryUserDetails entryUserDetails = (EntryUserDetails) authenticatedUser.getPrincipal();
            long jwt_expiration = JWT_EXPIRATION * 60000;
            Date currentDate = new Date();
            Date expirationDate = new Date(currentDate.getTime() + jwt_expiration);
            String token = Jwts
                .builder()
                .subject(authenticatedUser.getName())
                .issuedAt(currentDate)
                .expiration(expirationDate)
                .signWith(Keys.hmacSha256For(Decoders.BASE64.decode(JWT_SECRET)))
                .compact();
            entryUserDetails.setToken(token);
            SecurityContextHolder.getContext().setAuthentication(authenticatedUser);
            filterChain.doFilter(request, response);
        } catch(UsernameNotFoundException e) {
            response.setStatus(HttpStatus.FORBIDDEN.value());
            response.setContentType(MediaType.APPLICATION_JSON_VALUE);
            Error error = new Error("username", e.getMessage());
            ObjectMapper objectMapper = new ObjectMapper();
            String errorResponseString = objectMapper.writeValueAsString(error);
            response.getWriter().write(errorResponseString);
        }
    }
}

```

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 checked for a JWT token that is sent using the Bearer Scheme. In addition to authentication, in a

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

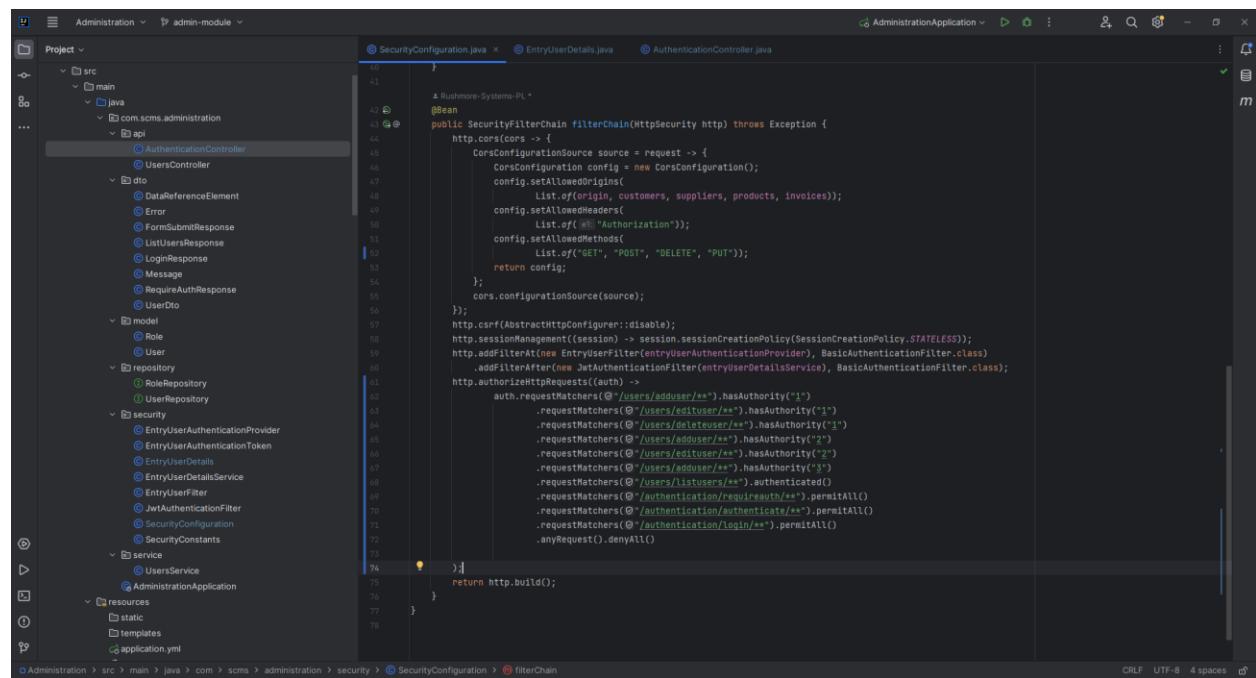
Authentication for other Microservices

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

Spring 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 SecurityConfig 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.



```
40 }
41 }
42 @Bean
43 public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
44     http.consCors -> {
45         CorsConfigurationSource source = request -> {
46             CorsConfiguration config = new CorsConfiguration();
47             config.setAllowedOrigins(
48                 List.of("origin, customers, suppliers, products, invoices"));
49             config.setAllowedHeaders(
50                 List.of("Authorization"));
51             config.setAllowedMethods(
52                 List.of("GET", "POST", "DELETE", "PUT"));
53             return config;
54         };
55         cors.configurationSource(source);
56     };
57     http.csrf(AbstractHttpConfigurer::disable);
58     http.sessionManagement((session) -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
59     http.addFilterAt(new EntryUserFilter(entryUserAuthenticationProvider), BasicAuthenticationFilter.class)
60         .addFilterAfter(new JwtAuthenticationFilter(entryUserDetailsService), BasicAuthenticationFilter.class);
61     http.authorizeHttpRequests(auth) ->
62         auth.requestMatchers("/users/adduser/**").hasAuthority("1")
63             .requestMatchers("/users/edituser/**").hasAuthority("1")
64             .requestMatchers("/users/deleteuser/**").hasAuthority("1")
65             .requestMatchers("/users/adduser/**").hasAuthority("2")
66             .requestMatchers("/users/edituser/**").hasAuthority("2")
67             .requestMatchers("/users/deleteuser/**").hasAuthority("2")
68             .requestMatchers("/users/listusers/**").authenticated()
69             .requestMatchers("/authentication/requireAuth/**").permitAll()
70             .requestMatchers("/authentication/authenticate/**").permitAll()
71             .requestMatchers("/authentication/login/**").permitAll()
72             .anyRequest().denyAll()
73     );
74 }
75 }
76 }
77 }
78 }
```

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

```

1 import org.springframework.stereotype.Service;
2 import java.util.List;
3 import java.util.Optional;
4 import javax.persistence.EntityNotFoundException;
5 import javax.persistence.EntityManager;
6 import javax.persistence.PersistenceContext;
7 import javax.persistence.Query;
8 import javax.persistence.criteria.CriteriaBuilder;
9 import javax.persistence.criteria.CriteriaQuery;
10 import javax.persistence.criteria.Root;
11 import org.hibernate.validator.internal.util.contracts.UserService;
12
13 @Service
14 public class UserService {
15     private final UserRepository userRepository;
16     private final RoleRepository roleRepository;
17
18     @Autowired
19     public UserService(UserRepository userRepository, RoleRepository roleRepository) {
20         this.userRepository = userRepository;
21         this.roleRepository = roleRepository;
22     }
23
24     /**
25      * Returns a list of all users
26      */
27     @Rushmore-Systems-PL
28     public List<User> listUsers() {
29         try {
30             List<User> users = new ArrayList<>();
31             for (User user : userRepository.findAll()) {
32                 users.add(new UserdataUser(user.getUsername(), user.getRoleId()));
33             }
34             List<DataReferenceElement> roles = new ArrayList<>();
35             for (Role role : roleRepository.findAll()) {
36                 roles.add(new DataReferenceElement(role.getId(), role.getDescription()));
37             }
38             return new ListUsersResponse(roles, users);
39         } catch (Exception e) {
40             return new ListUsersResponse(new ArrayList<>(), new ArrayList<>());
41         }
42     }
43
44     /**
45      * Adds a role to a user
46      */
47     @Rushmore-Systems-PL
48     public void addRoleToUser(@RequestBody UserdataUser user) {
49         EntityManager em = emf.createEntityManager();
50         CriteriaBuilder cb = em.getCriteriaBuilder();
51         CriteriaQuery<User> cq = cb.createQuery(User.class);
52         Root<User> root = cq.from(User.class);
53         cq.select(root).where(cb.equal(root.get("username"), user.getUsername()));
54         Query query = em.createQuery(cq);
55         User userEntity = query.getSingleResult();
56         userEntity.setRoleId(user.getRoleId());
57         em.merge(userEntity);
58         em.close();
59     }
60
61     /**
62      * Adds a user
63      */
64     @Rushmore-Systems-PL
65     @PostMapping(value = "/adduser")
66     public ResponseEntity<FormSubmitResponse> addUser(@RequestBody FormSubmitResponse formSubmitResponse) {
67         String username = formSubmitResponse.getUsername();
68         String password = formSubmitResponse.getPassword();
69         Long roleId = formSubmitResponse.getRoleId();
70         User user = userService.addUser(username, password, Long.parseLong(roleId));
71         return new ResponseEntity<FormSubmitResponse>(formSubmitResponse, HttpStatus.OK);
72     }
73
74     /**
75      * Deletes a user
76      */
77     @Rushmore-Systems-PL
78     @DeleteMapping(value = "/deleteuser/{id}")
79     public void deleteUser(@PathVariable Long id) {
80         User user = userRepository.findById(id);
81         if (user != null) {
82             userRepository.delete(user);
83         }
84     }
85
86     /**
87      * Deletes a role
88      */
89     @Rushmore-Systems-PL
90     @DeleteMapping(value = "/deleterole/{id}")
91     public void deleteRole(@PathVariable Long id) {
92         Role role = roleRepository.findById(id);
93         if (role != null) {
94             roleRepository.delete(role);
95         }
96     }
97 }

```

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

The User Controller 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.

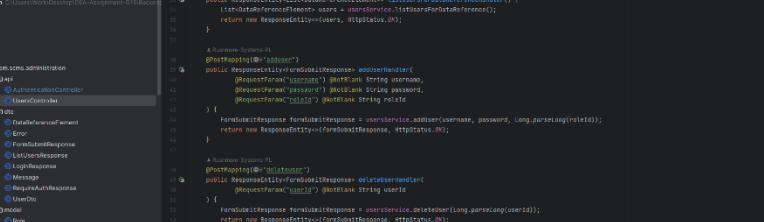
The User entity provides a domain model representation of the User table in the database. Similarly, the Role class represents the Role table, and they have a Many to One relationship as denoted by the @ManyToOne annotation.

Controller

```

1 import org.springframework.http.ResponseEntity;
2 import org.springframework.web.bind.annotation.*;
3 import java.util.List;
4
5 @RestController
6 @RequestMapping(value = "/users")
7 public class UsersController {
8
9     /**
10      * Returns a list of all users
11      */
12     @Rushmore-Systems-PL
13     @GetMapping(value = "listusers")
14     public ResponseEntity<ListUsersResponse> listUsersHandler() {
15         ListUsersResponse listUsersResponse = userService.listUsers();
16         return new ResponseEntity<ListUsersResponse>(listUsersResponse, HttpStatus.OK);
17     }
18
19     /**
20      * Adds a role to a user
21      */
22     @Rushmore-Systems-PL
23     @PostMapping(value = "listusersfordatareference")
24     public ResponseEntity<ListDataReferenceElement> listUsersForDataReferenceHandler() {
25         List<DataReferenceElement> users = userService.listUsersForDataReference();
26         return new ResponseEntity<ListDataReferenceElement>(users, HttpStatus.OK);
27     }
28
29     /**
30      * Adds a user
31      */
32     @Rushmore-Systems-PL
33     @PostMapping(value = "adduser")
34     public ResponseEntity<FormSubmitResponse> addUserHandler(
35         @RequestParam("username") @NotBlank String username,
36         @RequestParam("password") @NotBlank String password,
37         @RequestParam("roleId") @NotNull Long roleId
38     ) {
39         FormSubmitResponse formSubmitResponse = userService.addUser(username, password, Long.parseLong(roleId));
40         return new ResponseEntity<FormSubmitResponse>(formSubmitResponse, HttpStatus.OK);
41     }
42
43     /**
44      * Deletes a user
45      */
46     @Rushmore-Systems-PL
47     @DeleteMapping(value = "deleteuser/{id}")
48     public void deleteUserHandler(@PathVariable Long id) {
49         User user = userRepository.findById(id);
50         if (user != null) {
51             userRepository.delete(user);
52         }
53     }
54
55     /**
56      * Deletes a role
57      */
58     @Rushmore-Systems-PL
59     @DeleteMapping(value = "deleterole/{id}")
60     public void deleteRoleHandler(@PathVariable Long id) {
61         Role role = roleRepository.findById(id);
62         if (role != null) {
63             roleRepository.delete(role);
64         }
65     }
66 }

```



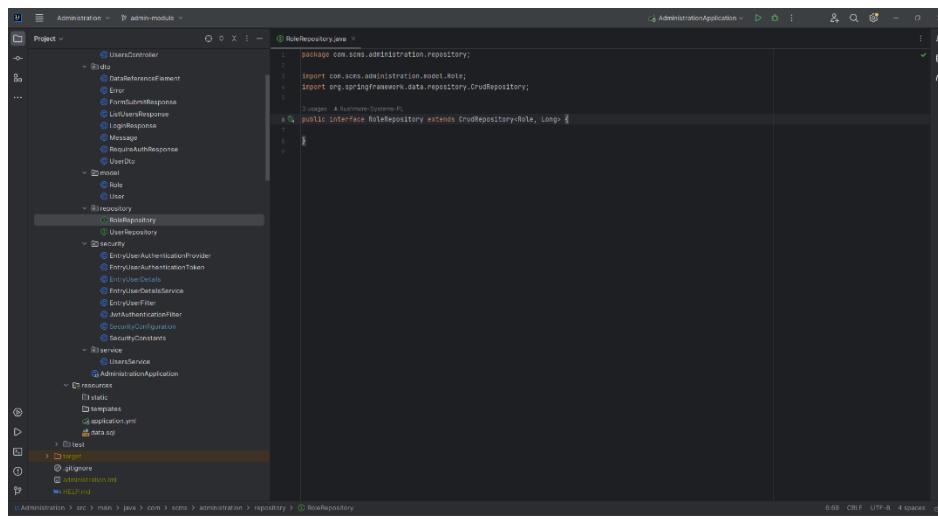
The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Tree:** Shows the `Administration` module structure. The `UsersController` class is selected in the tree.
- Code Editor:** Displays the `UsersController` class code. The code handles various HTTP requests for user management, including registration, login, password reset, and user deletion.
- Toolbars and Status Bar:** Standard IntelliJ IDEA toolbars and status bar at the bottom.

```
public ResponseEntity<User> registerUser(@RequestBody User user) {  
    return new ResponseEntity<User>(userService.addUser(user), HttpStatus.OK);  
}  
  
@PostMapping(value = "/login")  
public ResponseEntity<User> login(@RequestBody User user) {  
    return new ResponseEntity<User>(userService.login(user), HttpStatus.OK);  
}  
  
@PutMapping(value = "/password")  
public ResponseEntity<User> updatePassword(@RequestBody User user) {  
    return new ResponseEntity<User>(userService.updatePassword(user), HttpStatus.OK);  
}  
  
@DeleteMapping(value = "/{id}")  
public ResponseEntity<User> deleteUser(@PathVariable Long id) {  
    return new ResponseEntity<User>(userService.deleteUser(id), HttpStatus.OK);  
}  
  
@PutMapping(value = "/reset")  
public ResponseEntity<User> resetPassword(@RequestBody User user) {  
    return new ResponseEntity<User>(userService.resetPassword(user), HttpStatus.OK);  
}
```

Service

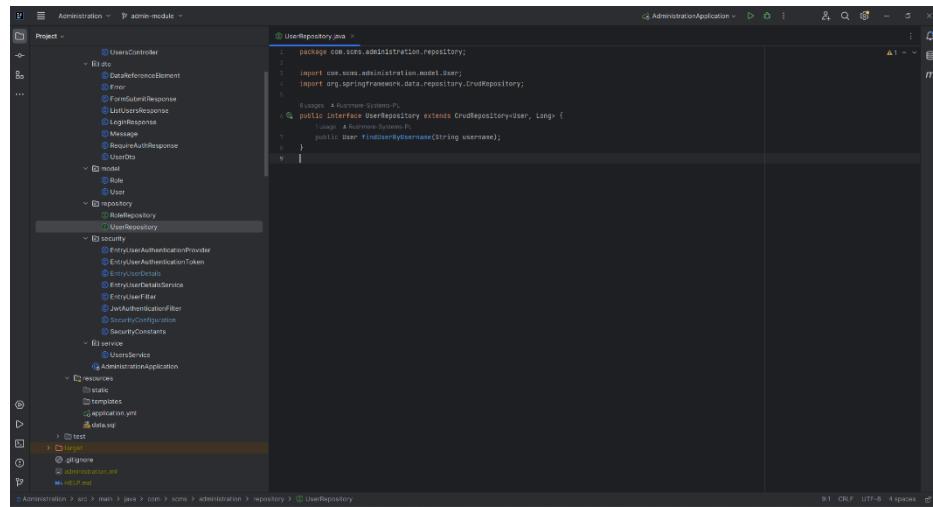
Repository



```
package com.scs.administration.repository;

import com.scs.administration.model.Role;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface RoleRepository extends CrudRepository<Role, Long> {
```

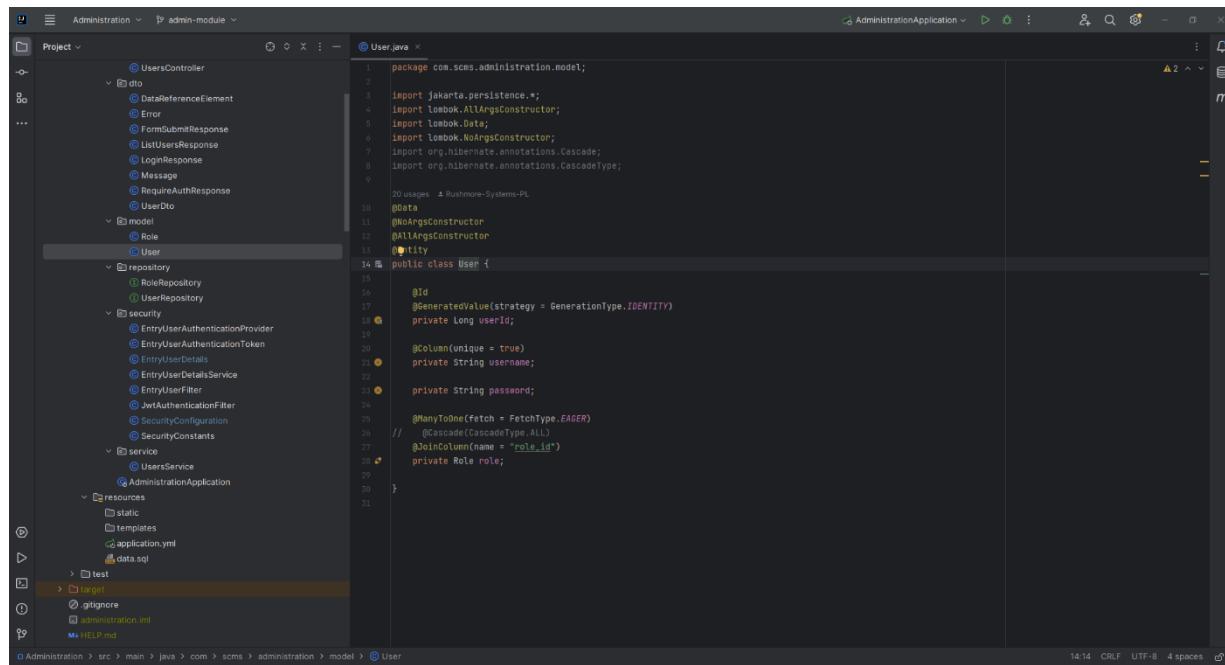


```
package com.scs.administration.repository;

import com.scs.administration.model.User;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface UserRepository extends CrudRepository<User, Long> {
    User findUserByUsername(String username);
}
```

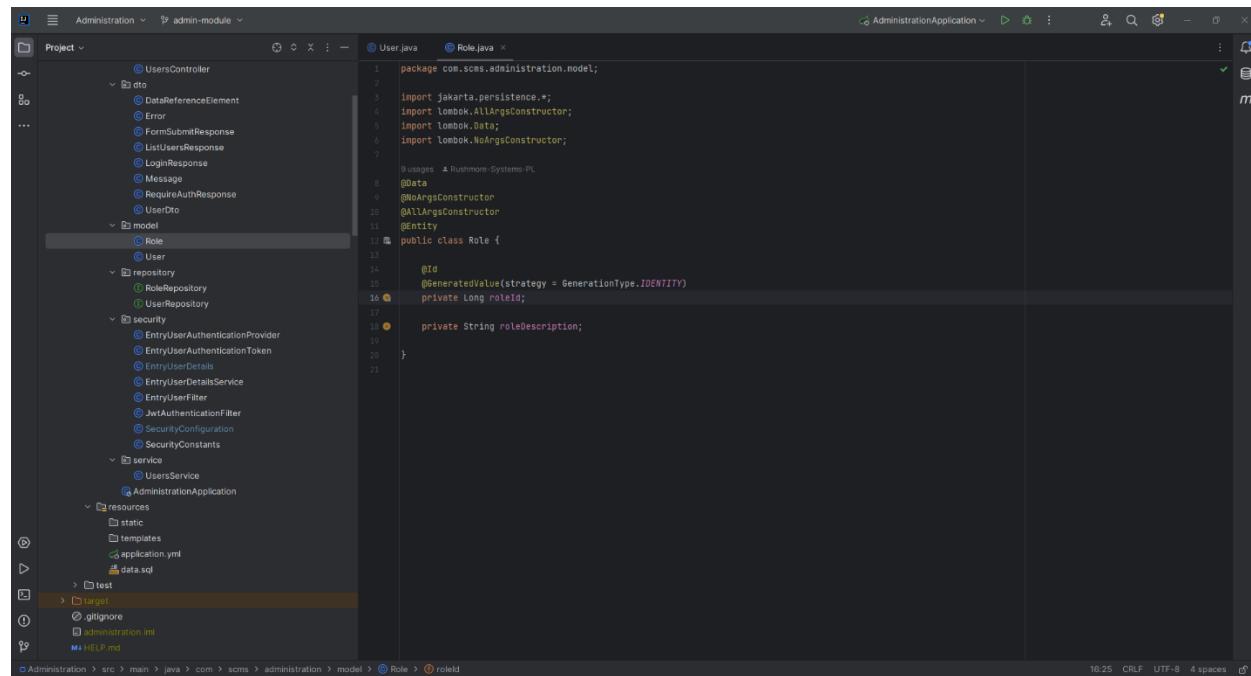
Models



The screenshot shows the User.java file in a Java IDE. The code defines a User entity with an ID, username, and password, and a many-to-one relationship with a Role. The code uses Lombok annotations like @Data, @NoArgsConstructor, and @AllArgsConstructor.

```
1 package com.scms.administration.model;
2
3 import jakarta.persistence.*;
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7 import org.hibernate.annotations.Cascade;
8 import org.hibernate.annotations.CascaseType;
9
10 import java.util.List;
11
12 @Entity
13 @Id
14 @GeneratedValue(strategy = GenerationType.IDENTITY)
15 private Long userId;
16
17 @Column(unique = true)
18 private String username;
19
20 private String password;
21
22 @ManyToOne(fetch = FetchType.EAGER)
23 @Cascade(CascadeType.ALL)
24 @JoinColumn(name = "role_id")
25 private Role role;
26
27 }
```

The project structure on the left includes UserController, Dto, Model, Repository, Security, and Service layers. Resources like application.yml and data.sql are also present.



The screenshot shows the Role.java file in a Java IDE. The code defines a Role entity with an ID and a role description. It uses Lombok annotations like @Data, @NoArgsConstructor, and @AllArgsConstructor.

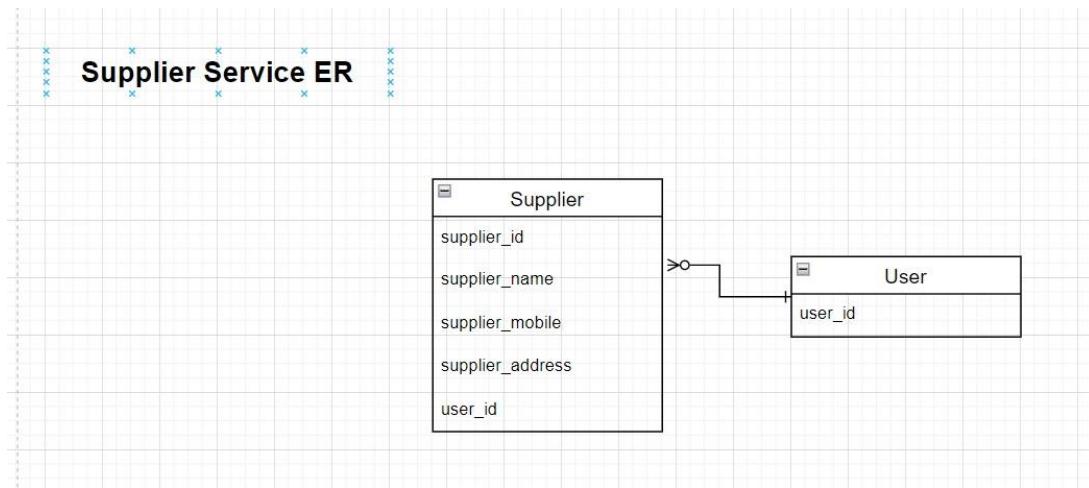
```
1 package com.scms.administration.model;
2
3 import jakarta.persistence.*;
4 import lombok.AllArgsConstructor;
5 import lombok.Data;
6 import lombok.NoArgsConstructor;
7
8 @Entity
9 @Id
10 @GeneratedValue(strategy = GenerationType.IDENTITY)
11 private Long roleId;
12
13 private String roleDescription;
14
15 }
```

The project structure on the left is identical to the first screenshot, showing the same layers and resource files.

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

The screenshot shows a web browser window with the URL `localhost:5173/suppliers`. The page title is "Suppliers". At the top right, there is a "Logout" button. Below the title, there is a table with the following data:

Supplier ID	Supplier Name	Supplier Mobile	Supplier Address	Last Modified User	Action
1	ABC (PVT) LTD	773359663	7B/5L, RADDOLUGAMA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
6	BDE (PVT) LTD	776067000	123/A, PUTTALAM	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
7	XYZ (PVT) LTD	442025000	123/B, COLOMBO	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
8	JKL (PVT) LTD	771013000	123/A, ANURADHAPURA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>

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

The screenshot shows a web browser window for 'Supply Chain Management System' at 'localhost:5173/suppliers'. The main page displays a table of existing suppliers with columns: Supplier ID, Supplier Name, Supplier Mobile, and Action. A modal dialog box titled 'Add Supplier' is open in the center. It contains fields for 'Supplier Name' (empty), 'Supplier Mobile' (empty), and 'Supplier Address' (empty). At the bottom right of the modal is a large 'Add Supplier' button. The background table shows supplier data:

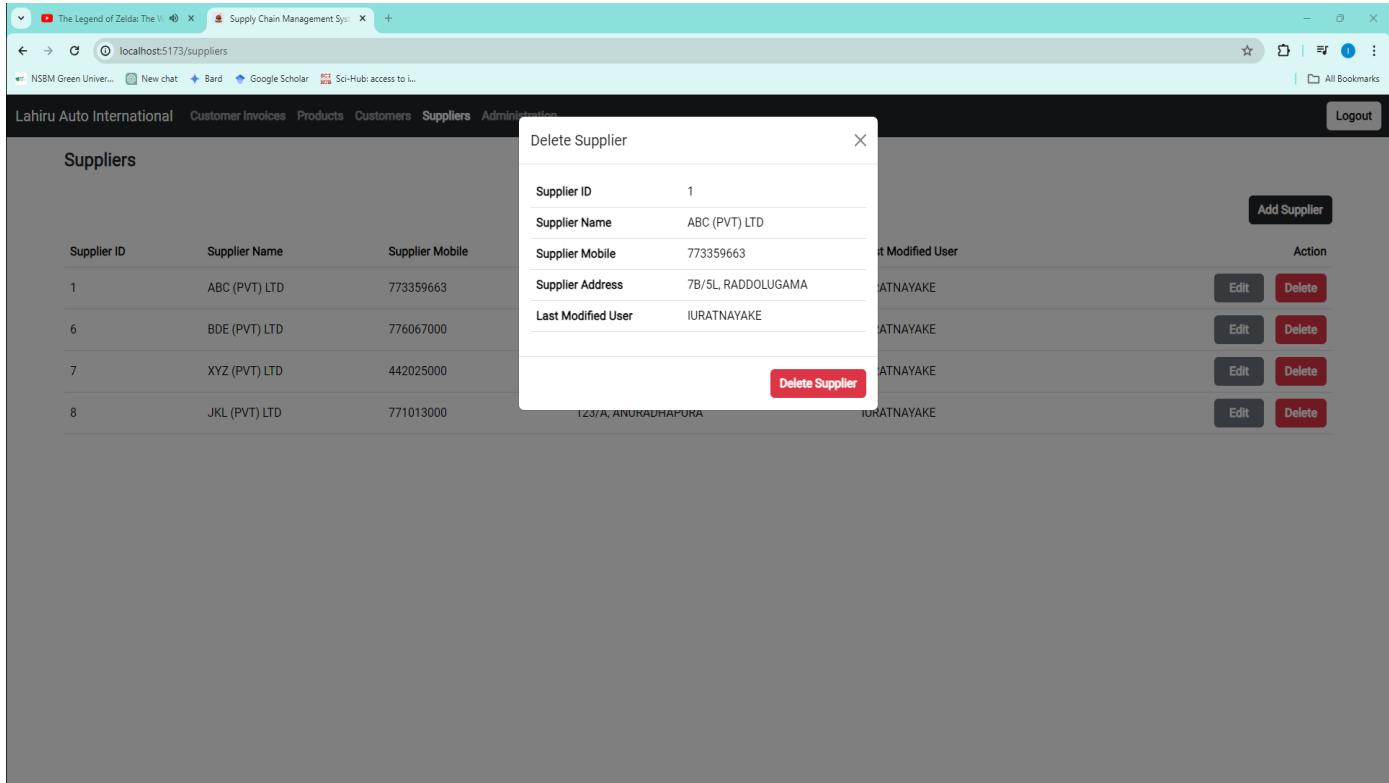
Supplier ID	Supplier Name	Supplier Mobile	Action
1	ABC (PVT) LTD	773359663	<button>Edit</button> <button>Delete</button>
6	BDE (PVT) LTD	776067000	<button>Edit</button> <button>Delete</button>
7	XYZ (PVT) LTD	442025000	<button>Edit</button> <button>Delete</button>
8	JKL (PVT) LTD	771013000	<button>Edit</button> <button>Delete</button>

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

The screenshot shows a web browser window for 'Supply Chain Management System' at 'localhost:5173/suppliers'. The main page displays a table of existing suppliers. A modal dialog box titled 'Edit Supplier' is open, showing updated information for supplier 'XYZ (PVT) LTD': Supplier Name (XYZ (PVT) LTD), Supplier Mobile (442025000), and Supplier Address (123/B, COLOMBO). At the bottom right of the modal is a 'Edit Supplier' button. The background table shows supplier data:

Supplier ID	Supplier Name	Supplier Mobile	Action
1	ABC (PVT) LTD	773359663	<button>Edit</button> <button>Delete</button>
6	BDE (PVT) LTD	776067000	<button>Edit</button> <button>Delete</button>
7	XYZ (PVT) LTD	442025000	<button>Edit</button> <button>Delete</button>
8	JKL (PVT) LTD	771013000	<button>Edit</button> <button>Delete</button>

- ❖ **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

The screenshot shows the IntelliJ IDEA interface with the Suppliers project open. The Supplier.java file is the active editor. The code defines a Supplier entity with fields for supplierId, supplierName, supplierMobile, supplierAddress, and userId, annotated with @Data, @Entity, and various @Column and @Id annotations. The project structure on the left shows the Supplier class is located under com.scms.suppliers.model. The status bar at the bottom right shows the date and time as 4/26/2024, 10:46 PM.

```
package com.scms.suppliers.model;

import ...;

public class Supplier {
    @Id
    @GeneratedValue(strategy= GenerationType.IDENTITY)
    private Long supplierId;

    @Column(unique = true)
    private String supplierName;
    private String supplierMobile;
    private String supplierAddress;
    private Long userId;
}
```

service

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

api

- SupplierController

The screenshot shows the IntelliJ IDEA interface with the Suppliers project open. The SuppliersService.java file is the active editor. The code implements a SuppliersService interface, utilizing a SupplierRepository to handle supplier operations. It includes methods for adding, editing, and deleting suppliers, along with exception handling and response generation.

```
package com.scms.suppliers.service;
import ...;
@Service
public class SuppliersService {
    private final SupplierRepository supplierRepository;
    @Autowired
    public SuppliersService(SupplierRepository supplierRepository) { this.supplierRepository = supplierRepository; }

    public FormSubmitResponse addSupplier(String supplierName, String supplierMobile, String supplierAddress, Long userId)
        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);
        }
    }

    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);
        }
    }

    public FormSubmitResponse deleteSupplier(String supplierId) {
        try{
            Long parsedSupplierId = Long.parseLong(supplierId);
            Optional<Supplier> supplierResult = supplierRepository.findById(parsedSupplierId);
        }
    }
}
```

This screenshot shows the same SuppliersService.java code as the previous one, but with a different section of the code highlighted. The code continues from the previous snippet, showing the implementation of the deleteSupplier method. It attempts to parse the supplier ID from a string and then finds the supplier by ID using the supplierRepository.

```
Long parsedSupplierId = Long.parseLong(supplierId);
Optional<Supplier> supplierResult = supplierRepository.findById(parsedSupplierId);
```

Suppliers > pom.xml (Suppliers) SupplierApplication.java SuppliersService.java Supplier.java

```
54     } catch (Exception e){
55         return new FormSubmitResponse(new Message(success: false, message: "Failed to edit Supplier."), errors: null);
56     }
57 }
58 }
59 1 usage
public FormSubmitResponse deleteSupplier(String supplierId) {
60     try{
61         Long parsedSupplierId = Long.parseLong(supplierId);
62         Optional<Supplier> supplierResult = supplierRepository.findById(parsedSupplierId);
63         if(supplierResult.isPresent()){
64             Supplier supplier = supplierResult.get();
65             supplierRepository.delete(supplier);
66             return new FormSubmitResponse(new Message(success: true, message: "Supplier successfully deleted."), errors: null);
67         }else{
68             return new FormSubmitResponse(new Message(success: false, message: "Supplier not found."), errors: null);
69         }
70     }catch(NumberFormatException e){
71         return new FormSubmitResponse(new Message(success: false, message: "invalid supplier ID format."), errors: null);
72     }catch(Exception e){
73         return new FormSubmitResponse(new Message(success: false, message: "Failed to delete Supplier."), errors: null);
74     }
75 }
76 }
77 1 usage
public ListSuppliersResponse listSuppliers(){
78     try{
79         List<SupplierDto> suppliers = new ArrayList<>();
80         for(Supplier supplier : supplierRepository.findAll()){
81             suppliers.add(new SupplierDto(supplier.getSupplierId(), supplier.getSupplierName(), supplier.getSupplierMobile(), supplier.getSupplierA
82         }
83     }
84     return new ListSuppliersResponse(suppliers);
85 }catch (Exception e){
86     return new ListSuppliersResponse(new ArrayList<>());
87 }
88 }
89 }
90 }
```

16:14 CRLF UTF-8 4 spaces

Type here to search 28°C Partly cloudy 11:20 PM 4/26/2024

Suppliers > pom.xml (Suppliers) SupplierApplication.java SuppliersService.java Supplier.java

```
70 }catch(NumberFormatException e){
71     return new FormSubmitResponse(new Message(success: false, message: "invalid supplier ID format."), errors: null);
72 }catch(Exception e){
73     return new FormSubmitResponse(new Message(success: false, message: "Failed to delete Supplier."), errors: null);
74 }
75 }
76
77 1 usage
public ListSuppliersResponse listSuppliers(){
78     try{
79         List<SupplierDto> suppliers = new ArrayList<>();
80         for(Supplier supplier : supplierRepository.findAll()){
81             suppliers.add(new SupplierDto(supplier.getSupplierId(), supplier.getSupplierName(), supplier.getSupplierMobile(), supplier.getSupplierA
82         }
83     }
84     return new ListSuppliersResponse(suppliers);
85 }catch (Exception e){
86     return new ListSuppliersResponse(new ArrayList<>());
87 }
88 }
89 }
90 }
```

16:14 CRLF UTF-8 4 spaces

Type here to search 28°C Partly cloudy 11:24 PM 4/26/2024

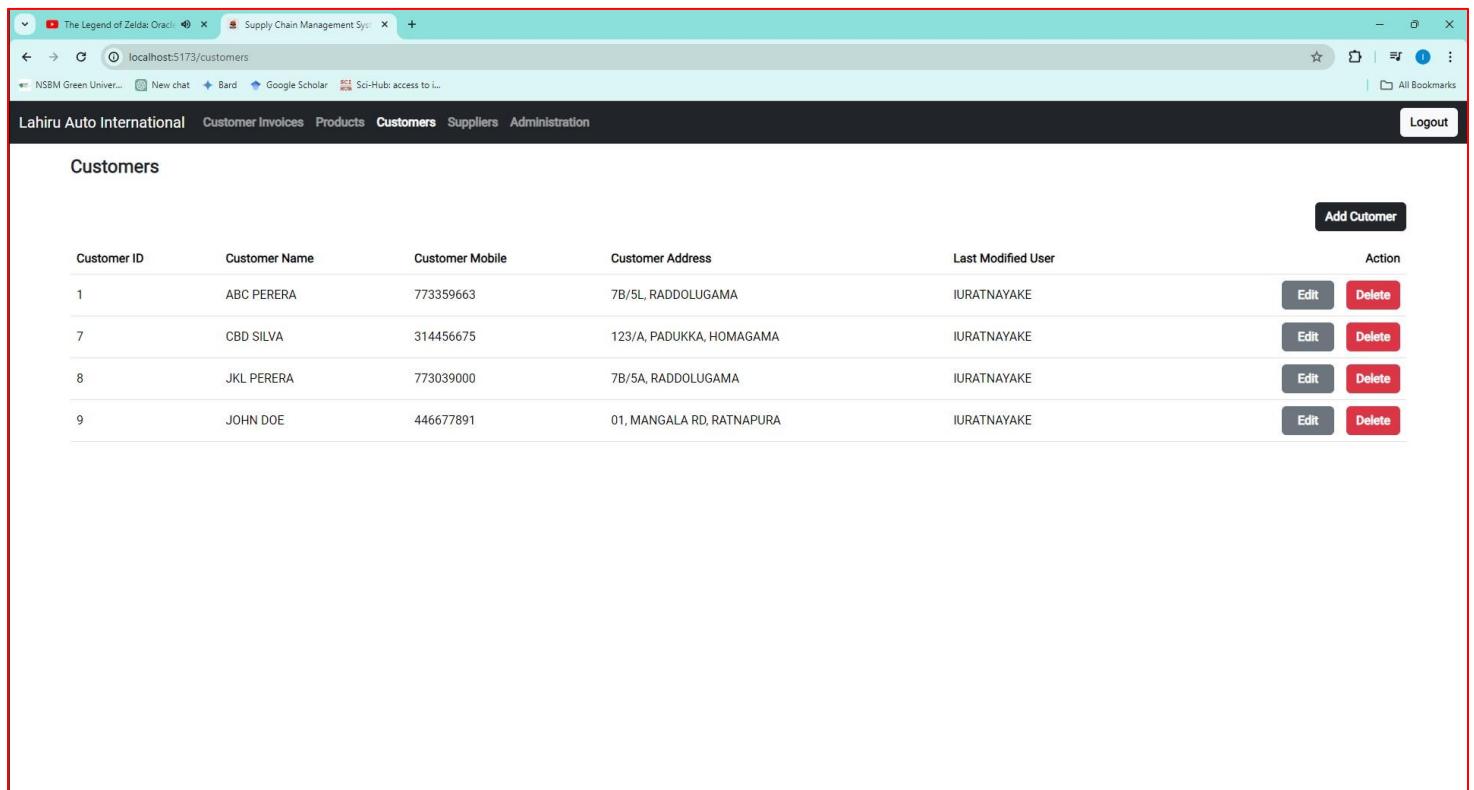
Customer Microservice

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 ID	Customer Name	Customer Mobile	Customer Address	Last Modified User	Action
1	ABC PERERA	773359663	7B/5L, RADDOLUGAMA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
7	CBD SILVA	314456675	123/A, PADUKKA, HOMAGAMA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
8	JKL PERERA	773039000	7B/5A, RADDOLUGAMA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
9	JOHN DOE	446677891	01, MANGALA RD, RATNAPURA	IURATNAYAKE	<button>Edit</button> <button>Delete</button>

Customer Create Page

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

The screenshot shows a web browser window with a red border around the main content area. The title bar says "Supply Chain Management System". The address bar shows "localhost:5173/customers". The page header includes "Lahiru Auto International" and navigation links like "Customer Invoices", "Products", "Customers", "Suppliers", "Administration", and "Logout". A modal window titled "Add Customer" is open in the center. It contains three input fields: "Customer Name" (empty), "Customer Mobile" (empty), and "Customer Address" (empty). Below these fields is a "Add Customer" button. To the right of the modal, there is a table with four rows of customer data. Each row has columns for "Customer ID", "Customer Name", "Customer Mobile", "Last Modified User", and "Action". The "Action" column for each row contains "Edit" and "Delete" buttons. The data in the table is as follows:

Customer ID	Customer Name	Customer Mobile	Last Modified User	Action
1	ABC PERERA	773359663	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
7	CBD SILVA	314456675	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
8	JKL PERERA	773039000	IURATNAYAKE	<button>Edit</button> <button>Delete</button>
9	JOHN DOE	446677891	IURATNAYAKE	<button>Edit</button> <button>Delete</button>

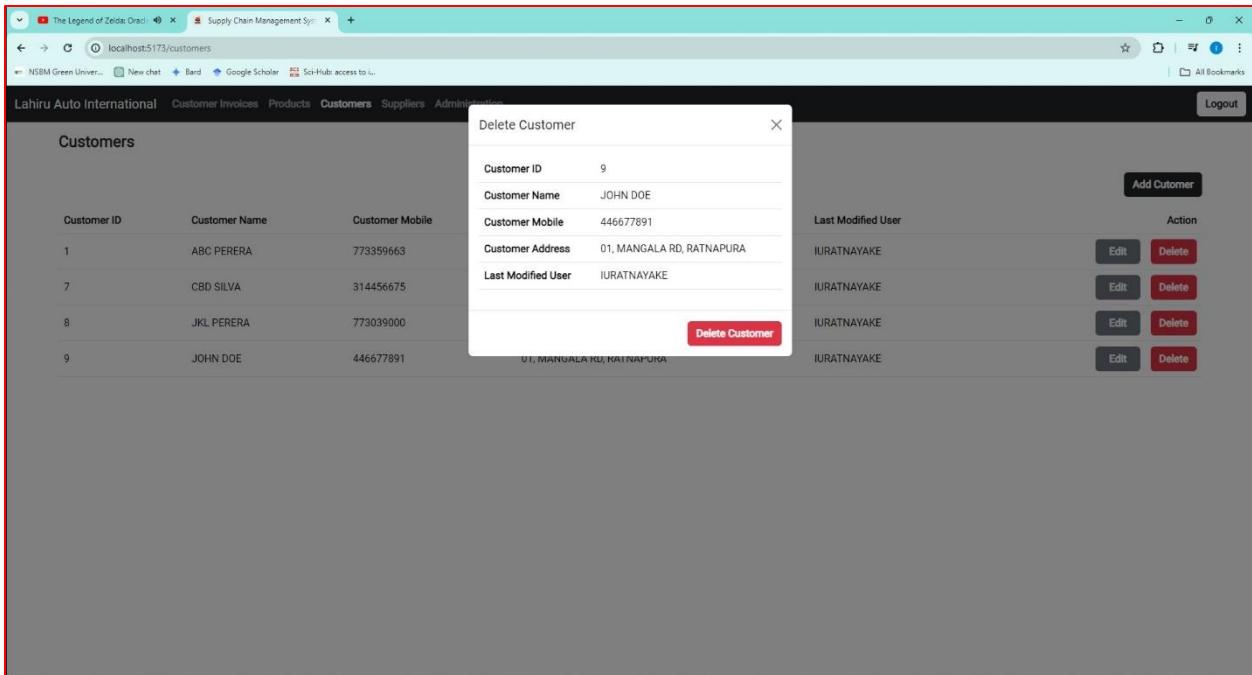
Customer Edit Page

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

The screenshot shows a web browser window with a teal header bar containing the text "Supply Chain Management System". Below the header is a navigation bar with links: "NSBM Green University", "New chat", "Bard", "Google Scholar", "Sci-Hub: access to i...", "Logout", and "All Bookmarks". The main content area has a dark header bar with the text "Lahiru Auto International" and several menu items: "Customer Invoices", "Products", "Customers", "Suppliers", "Administrations", and "Logout". Below this is a sub-header "Customers". On the left, there is a table with columns "Customer ID", "Customer Name", and "Customer Mobile". The rows show data for customers with IDs 1, 7, 8, and 9, with names like ABC PERERA, CBD SILVA, JKL PERERA, and JOHN DOE respectively. To the right of the table is an "Edit Customer" dialog box. This dialog has three input fields: "Customer Name" (containing "ABC PERERA"), "Customer Mobile" (containing "773359663"), and "Customer Address" (containing "7B/5L, RADDOLUGAMA"). At the bottom of the dialog is a "Edit Customer" button. To the right of the dialog is a table titled "Last Modified User" with four rows, each showing "IURATNAYAKE" in the "Last Modified User" column and "Edit" and "Delete" buttons in the "Action" column. The entire screenshot is framed by a red border.

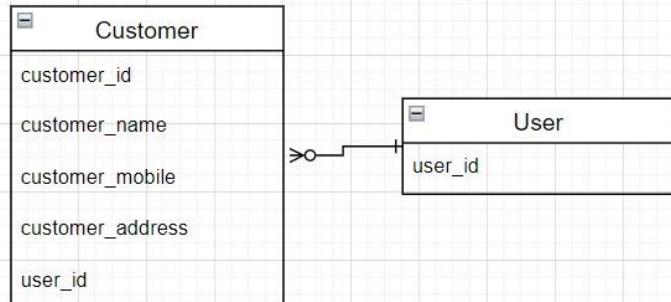
Customer delete Page

In here you can delete customer use by delete button.



ER Diagram

Customer Service ER

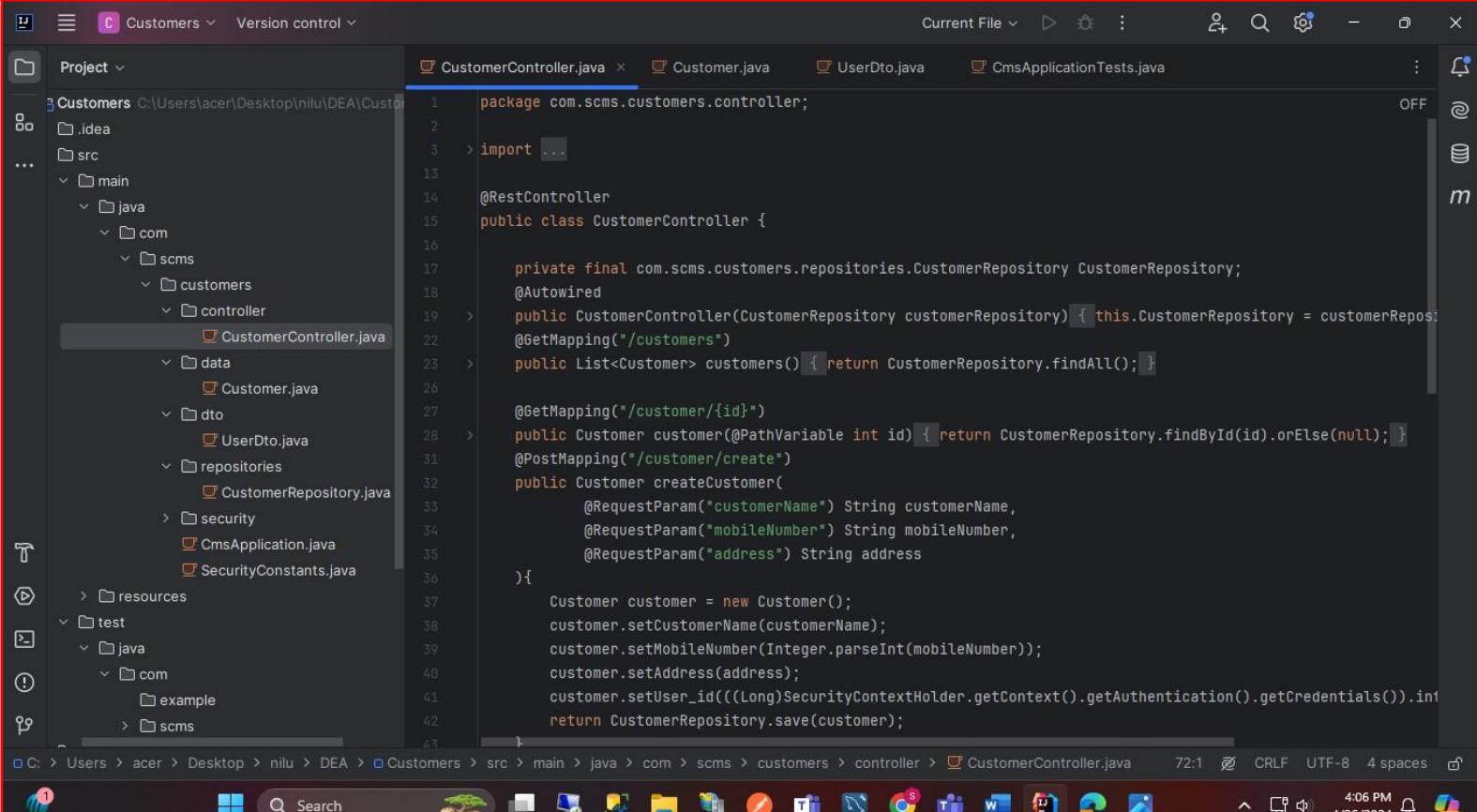


Backend Code

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.



The screenshot shows the IntelliJ IDEA interface with the 'Customers' project open. The 'CustomerController.java' file is the active editor tab. The code implements a REST controller for managing customers, utilizing annotations like @RestController, @Autowired, @GetMapping, and @PostMapping. It interacts with a CustomerRepository to perform operations like finding all customers or creating a new customer. The left sidebar displays the project structure, and the bottom shows the system tray and taskbar.

```
package com.scms.customers.controller;

import ...

@RestController
public class CustomerController {

    private final com.scms.customers.repositories.CustomerRepository customerRepository;
    @Autowired
    public CustomerController(CustomerRepository customerRepository) { this.customerRepository = customerRepository; }

    @GetMapping("/customers")
    public List<Customer> customers() { return customerRepository.findAll(); }

    @GetMapping("/customer/{id}")
    public Customer customer(@PathVariable int id) { return customerRepository.findById(id).orElse(null); }

    @PostMapping("/customer/create")
    public Customer createCustomer(
        @RequestParam("customerName") String customerName,
        @RequestParam("mobileNumber") String mobileNumber,
        @RequestParam("address") String address
    ) {
        Customer customer = new Customer();
        customer.setCustomerName(customerName);
        customer.setMobileNumber(Integer.parseInt(mobileNumber));
        customer.setAddress(address);
        customer.setUser_id(((Long)SecurityContextHolder.getContext().getAuthentication().getCredentials()).getId());
        return customerRepository.save(customer);
    }
}
```

The screenshot shows a Java application development environment with the following details:

- Project Explorer:** The project is named "Customers" located at `C:\Users\acer\Desktop\nilu\DEA\Customers`. It contains the following structure:
 - `.idea`
 - `src`:
 - `main`:
 - `java`:
 - `com`:
 - `scms`:
 - `customers`:
 - `controller`:
 - `CustomerController.java` (selected)
 - `test`:
 - `java`:
 - `com`:
 - `example`
 - `resources`
 - `test`
 - Code Editor:** The file `CustomerController.java` is open and selected. The code implements a REST controller for managing customers. It includes methods for updating and deleting customers using `@PutMapping` and `@DeleteMapping` annotations respectively. It uses `@RequestParam` annotations to map request parameters to method arguments. It also uses `@RequestContextHolder` to get the security context.
 - Toolbars and Status Bar:** The status bar at the bottom shows the current file path (`C:\Users\acer\Desktop\nilu\DEA\Customers>src>main>java>com>scms>customers>controller>CustomerController.java`), the current time (4:07 PM), and the date (4/26/2024).

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.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar displays the project structure under the "Customers" module. The "Customer.java" file is selected in the "src/main/java/com/scms/customers/data" directory.
- Editor:** The main window shows the code for **Customer.java**. The code defines a JPA entity with fields for customer ID, name, mobile number, address, and user ID. It includes standard getters and setters.
- Toolbars and Status Bar:** The top bar shows tabs for "CustomerController.java", "Customer.java", "UserDto.java", and "CmsApplicationTests.java". The status bar at the bottom shows the file path "C:/Users/acer/Desktop/nilu/DEA/Customers/src/main/java/com/scms/customers/data/Customer.java", the current time "7:24", and file statistics like "CRLF", "UTF-8", and "4 spaces".

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.

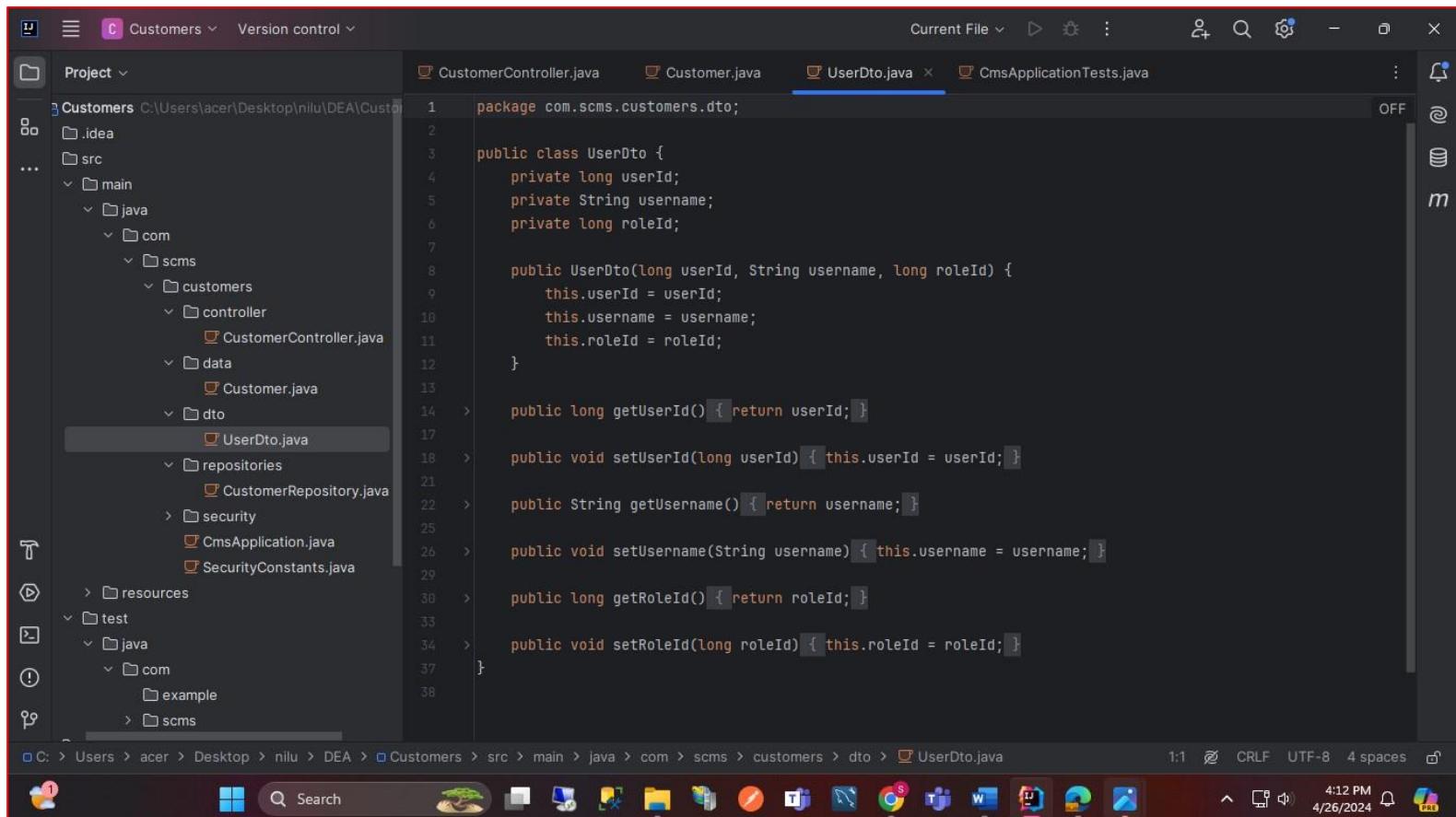
Key Features:

- 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.



The screenshot shows a Java IDE interface with the following details:

- Project Structure:** The project is named "Customers" and contains a ".idea" folder, a "src" folder with "main", "java", and "com" subfolders. Inside "com/scms/customers/dto", there is a "UserDto.java" file which is currently selected.
- Code Editor:** The code for "UserDto.java" is displayed:

```
1 package com.scms.customers.dto;
2
3 public class UserDto {
4     private long userId;
5     private String username;
6     private long roleId;
7
8     public UserDto(long userId, String username, long roleId) {
9         this.userId = userId;
10        this.username = username;
11        this.roleId = roleId;
12    }
13
14    public long getUserId() { return userId; }
15
16    public void setUserId(long userId) { this.userId = userId; }
17
18    public String getUsername() { return username; }
19
20    public void setUsername(String username) { this.username = username; }
21
22    public long getRoleId() { return roleId; }
23
24    public void setRoleId(long roleId) { this.roleId = roleId; }
25
26 }
```
- File Explorer:** Shows other files like "CustomerController.java", "Customer.java", "CmsApplicationTests.java", "CustomerRepository.java", "CmsApplication.java", and "SecurityConstants.java".
- Bottom Bar:** Includes a search bar, system icons, and a status bar showing the path "C:\Users\acer\Desktop\nilu\DEA\Customers", file encoding "UTF-8", and date/time "4/26/2024 4:12 PM".

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.

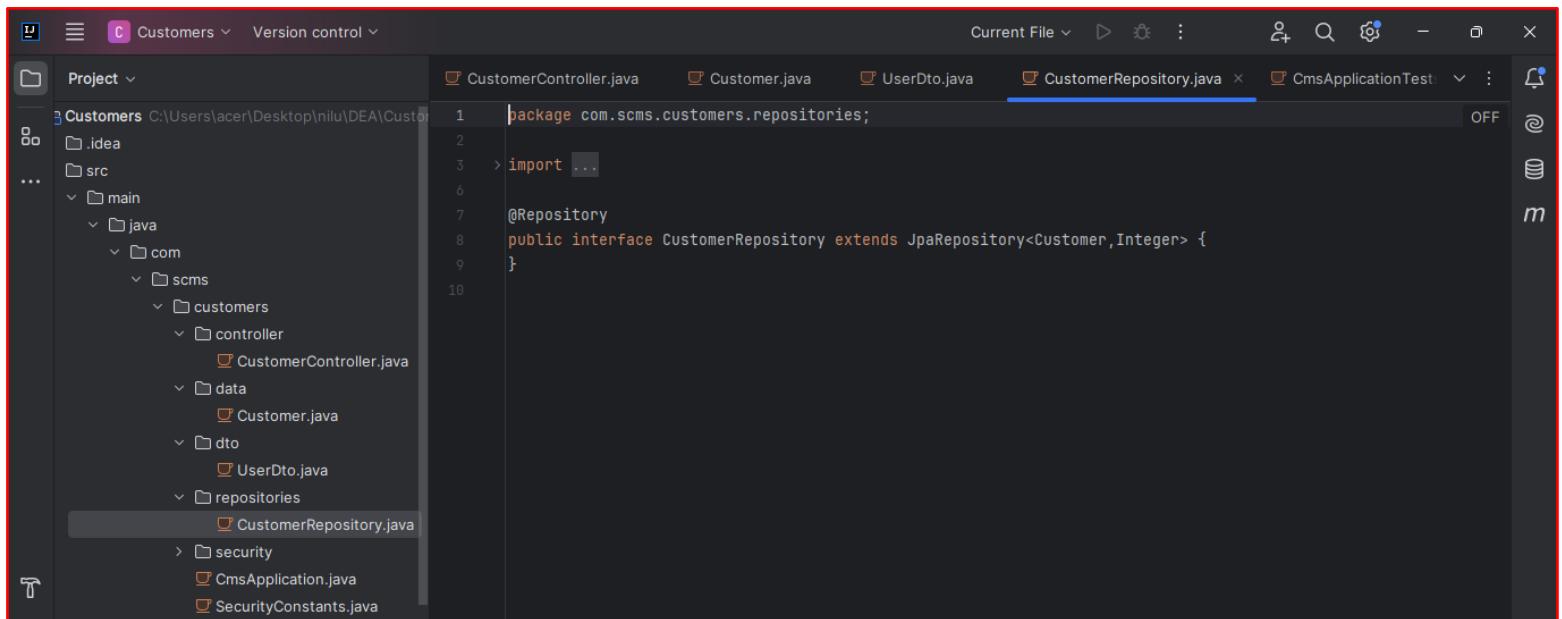
Usage:

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.



```
package com.scms.customers.repositories;
import ...;
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Integer> { }
```

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.

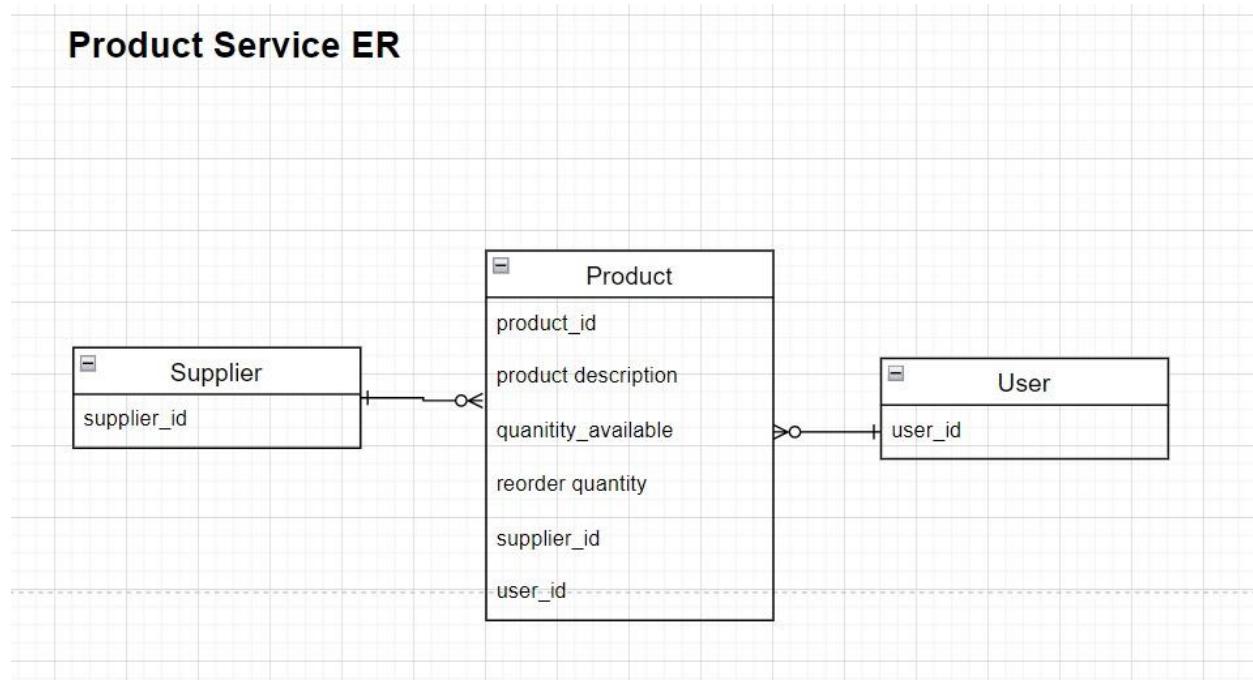
Usage:

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.

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project View:** On the left, the project structure is displayed under the "Products" project. It includes the main directory, .idea, out, and src. The src folder contains main, java, com.scms.products, controller, ProductController, dto, models, Product, repository, security, and service. The ProductController.java file is currently selected.
- Code Editor:** The right side shows the code for ProductController.java. The code defines a REST controller for products, autowiring a ProductService and mapping to a listProductsHandler method.
- Status Bar:** At the bottom, the status bar shows the path as Products > src > main > java > com > scms > products > controller > ProductController, the file name, and various system information like date and time.

```
1 package com.scms.products.controller;
2
3 import com.scms.products.dto.DataReferenceElement;
4 import com.scms.products.models.Product;
5 import com.scms.products.service.ProductService;
6 import org.springframework.beans.factory.annotation.Autowired;
7 import org.springframework.http.HttpStatus;
8 import org.springframework.http.ResponseEntity;
9 import org.springframework.web.bind.annotation.*;
10
11 import java.util.List;
12
13 @RestController
14 @RequestMapping("/{products}")
15 public class ProductController {
16
17     private final ProductService productService;
18
19     @Autowired
20     public ProductController(ProductService productService) { this.productService = productService; }
21
22     @GetMapping("/listproducts")
23     public ResponseEntity<List<Product>> listProductsHandler() {
24         List<Product> products = productService.listProducts();
25         return ResponseEntity.ok(products);
26     }
27
28 }
```

The screenshot shows a Java application structure for a 'Products' project. The 'src/main/java/com.scms.products/controller' package contains a 'ProductController' class. The code implements four REST endpoints using Spring's @PostMapping annotations:

```
29  
30  
31 @GetMapping(value = "listproductsforreference")  
public ResponseEntity<List<DataReferenceElement>> createProduct(@RequestBody Product product)  
    List<DataReferenceElement> products = productService.listProductsForReference();  
    return ResponseEntity.ok(products);  
}  
  
36 @PostMapping(value = "addproduct")  
public ResponseEntity<Product> addProductHandler(@RequestBody Product product) {  
    Product insertedProduct = productService.addProduct(product);  
    if(insertedProduct != null) {  
        return new ResponseEntity<Product>(insertedProduct, HttpStatus.OK);  
    }  
    return new ResponseEntity<Product>(null, HttpStatus.OK);  
}  
  
45 @PostMapping(value = "editproduct")  
public ResponseEntity<Product> editProductHandler(@RequestBody Product product) {  
    Product insertedProduct = productService.editProduct(product);  
    if(insertedProduct != null) {  
        return new ResponseEntity<Product>(insertedProduct, HttpStatus.OK);  
    }  
    return new ResponseEntity<Product>(null, HttpStatus.OK);  
}  
  
54 @PostMapping(value = "deleteproduct")  
public ResponseEntity<Product> deleteProductHandler(@RequestBody Product product) {  
    boolean success = productService.deleteProduct(product);  
    if(success) {  
        return new ResponseEntity<Product>(product, HttpStatus.OK);  
    }  
    return new ResponseEntity<Product>(null, HttpStatus.OK);  
}
```

This screenshot is identical to the one above, showing the same Java code for the 'ProductController' class. The code defines four REST endpoints using Spring's @PostMapping annotations to handle product creation, update, and deletion.

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.

The screenshot shows an IDE interface with the following details:

- Project View:** Shows a tree structure of the project. The `src` folder contains `main`, which further contains `java`. Inside `java`, there is a package `com.scms.products` containing `controller`, `models`, `repository`, `security`, and `service` sub-packages.
- Code Editor:** The current file is `Product.java`. The code defines a class `Product` with various annotations and fields. A specific field, `quantityAvailable`, is highlighted with a green underline.
- Toolbar:** Standard IDE toolbar with icons for file operations, search, and navigation.
- Status Bar:** Shows the file path (`Products > src > main > java > com > scms > products > models > Product`), encoding (`23:2 CRLF UTF-8`), and system status (e.g., battery level, network).

The screenshot shows an IDE interface with the following details:

- Project View:** Similar to the first screenshot, showing the `src` folder with `main`, `java`, and the `com.scms.products` package.
- Code Editor:** The current file is `ProductRepository.java`. It defines an interface `ProductRepository` extending `CrudRepository<Product, Long>`.
- Toolbar:** Standard IDE toolbar.
- Status Bar:** Shows the file path (`Products > src > main > java > com > scms > products > repository > ProductRepository`), encoding (`8:18 CRLF UTF-8`), and system status.

FrontEnd Design for Product API

The Legend of Zelda: Majora's Mask Supply Chain Management System

localhost:5173/products

Lahiru Auto International Customer Invoices Products Customers Suppliers Administration Logout

Products

						Add Product
Product ID	Product Name	Available Quantity	Re-order Quantity	Supplier	Last Modified User	Action
1	Spoilers	1000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
4	Wheel Cup	2000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
5	Mud Guard	2000	150	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
8	Spokes	3000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>

The Legend of Zelda: Majora's Mask Supply Chain Management System

localhost:5173/products

Lahiru Auto International Customer Invoices Products Customers Suppliers Administration Logout

Products

Product ID	Product Name	Available Quantity
1	Spoilers	1000
4	Wheel Cup	2000
5	Mud Guard	2000
8	Spokes	3000

Add Product

Product Name:

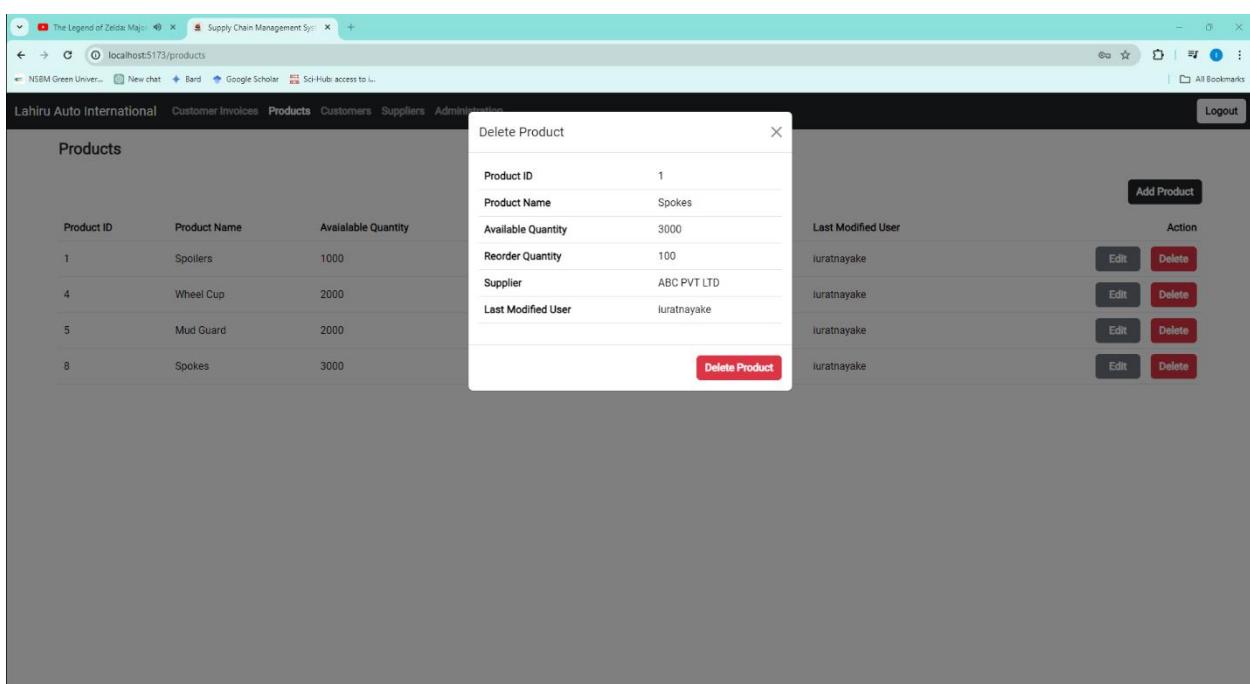
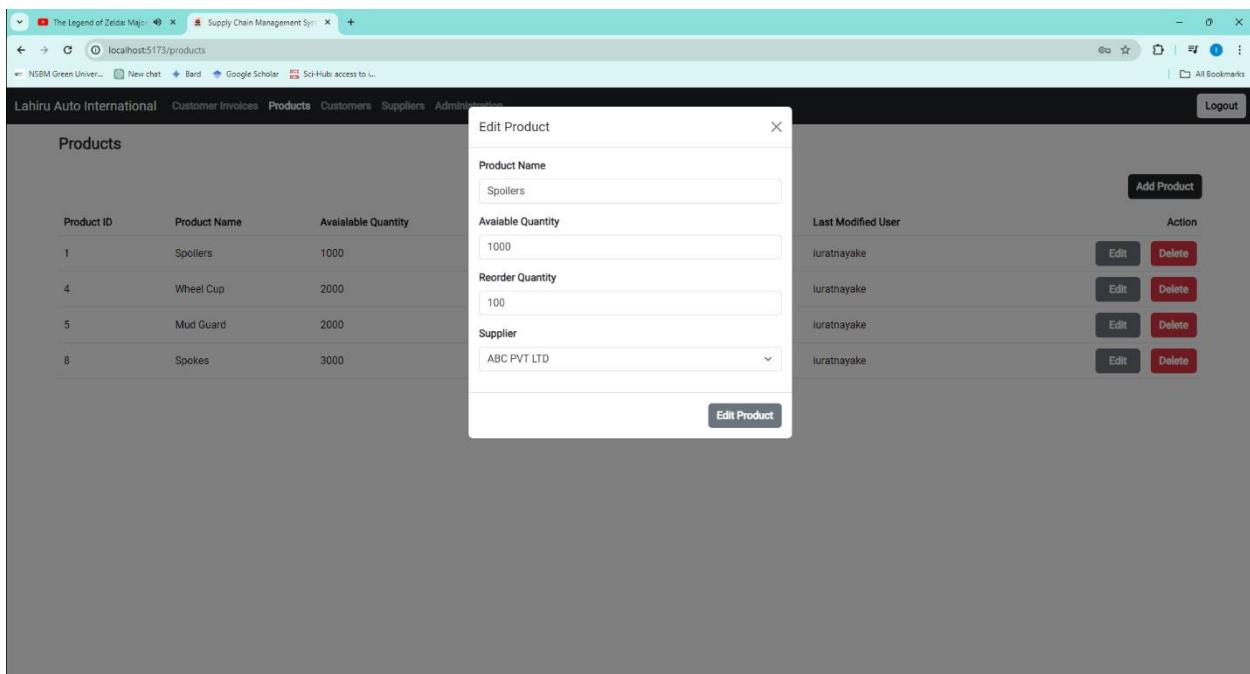
Available Quantity:

Reorder Quantity:

Supplier:

Add Product

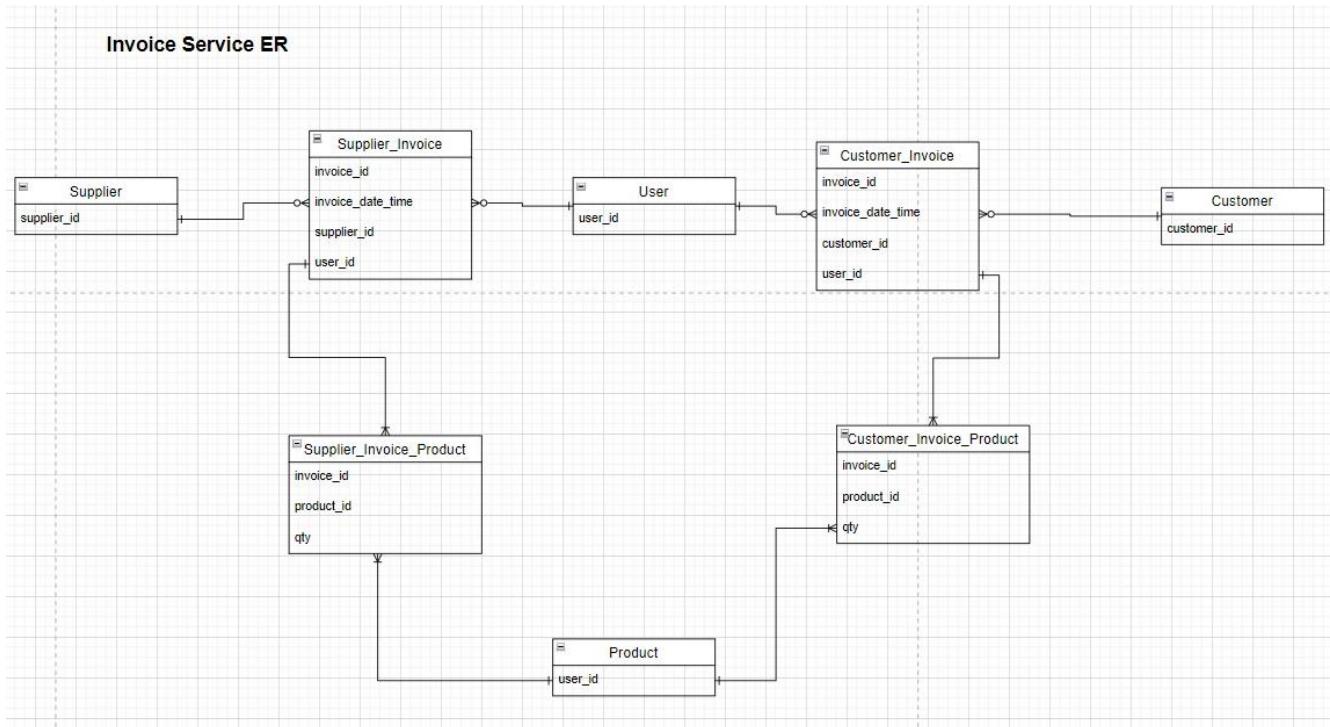
						Add Product
Product ID	Product Name	Available Quantity	Re-order Quantity	Supplier	Last Modified User	Action
1	Spoilers	1000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
4	Wheel Cup	2000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
5	Mud Guard	2000	150	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>
8	Spokes	3000	100	ABC PVT LTD	luratnayake	<button>Edit</button> <button>Delete</button>



Invoice Service

1. System Architecture

The Lahiru Auto International 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:

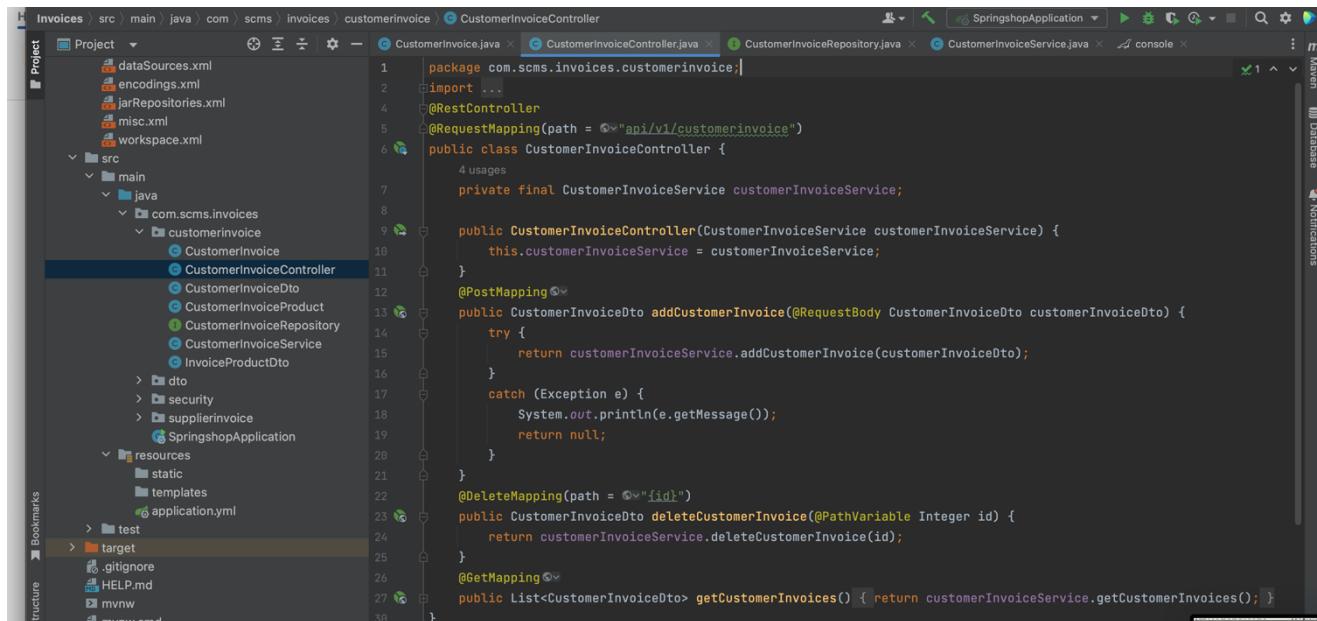
- Accepts requests from clients to create, retrieve, or delete customer invoices.
- Interacts with the database to persist customer invoice data.
- Ensures data integrity and consistency when associating products with invoices.

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.



The screenshot shows the IntelliJ IDEA interface with the project 'Invoices' open. The left sidebar displays the project structure, including 'src' (main, java, com.scms.invoices.customerinvoice), 'resources', 'test', and 'target'. The right pane shows the code editor for 'CustomerInvoiceController.java'. The code defines a REST controller for managing customer invoices:

```
package com.scms.invoices.customerinvoice;
import ...
import org.springframework.web.bind.annotation.*;
import com.scms.invoices.customerinvoice.CustomerInvoiceService;
import com.scms.invoices.customerinvoice.CustomerInvoice;
import com.scms.invoices.customerinvoice.CustomerInvoiceDto;
import com.scms.invoices.customerinvoice.CustomerInvoiceProduct;
import com.scms.invoices.customerinvoice.CustomerInvoiceRepository;
import com.scms.invoices.customerinvoice.CustomerInvoiceService;
import com.scms.invoices.customerinvoice.InvoiceProductDto;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(path = "/api/v1/customerinvoice")
public class CustomerInvoiceController {
    private final CustomerInvoiceService customerInvoiceService;
    public CustomerInvoiceController(CustomerInvoiceService customerInvoiceService) {
        this.customerInvoiceService = customerInvoiceService;
    }
    @PostMapping
    public CustomerInvoiceDto addCustomerInvoice(@RequestBody CustomerInvoiceDto customerInvoiceDto) {
        try {
            return customerInvoiceService.addCustomerInvoice(customerInvoiceDto);
        } catch (Exception e) {
            System.out.println(e.getMessage());
            return null;
        }
    }
    @DeleteMapping(path = "/{id}")
    public CustomerInvoiceDto deleteCustomerInvoice(@PathVariable Integer id) {
        return customerInvoiceService.deleteCustomerInvoice(id);
    }
    @GetMapping
    public List<CustomerInvoiceDto> getCustomerInvoices() {
        return customerInvoiceService.getCustomerInvoices();
    }
}
```

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

```

Invoices - SupplierInvoiceController.java
-----
1 package com.scms.invoices.supplierinvoice;
2 import ...
3 ...
4 @RestController
5 ...
6 @RequestMapping(path = "/api/v1/supplierinvoice")
7 public class SupplierInvoiceController {
8     ...
9     private final SupplierInvoiceService supplierInvoiceService;
10    ...
11    public SupplierInvoiceController(SupplierInvoiceService supplierInvoiceService) {
12        this.supplierInvoiceService = supplierInvoiceService;
13    }
14    ...
15    @PostMapping
16    public SupplierInvoiceDto addSupplierInvoice(@RequestBody SupplierInvoiceDto supplierInvoiceDto) {
17        try {
18            return supplierInvoiceService.addSupplierInvoice(supplierInvoiceDto);
19        } catch (Exception e) {
20            System.out.println(e.getMessage());
21            return null;
22        }
23    }
24    ...
25    @DeleteMapping(path = "/{supplierInvoiceId}")
26    public SupplierInvoiceDto deleteSupplierInvoice(@PathVariable Integer supplierInvoiceId) {
27        return supplierInvoiceService.deleteSupplierInvoice(supplierInvoiceId);
28    }
29    ...
30    @GetMapping
31    public List<SupplierInvoiceDto> getSupplierInvoices() { return supplierInvoiceService.getSupplierInvoices(); }
}

```

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:

customerInvoiceld (Primary Key)

customerId

userId

invoiceDate

CustomerInvoiceProduct:

customerInvoiceld (Foreign Key)

productId

quantity

4.1. Customer Invoice Entity

- **customerInvoiceld:** 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

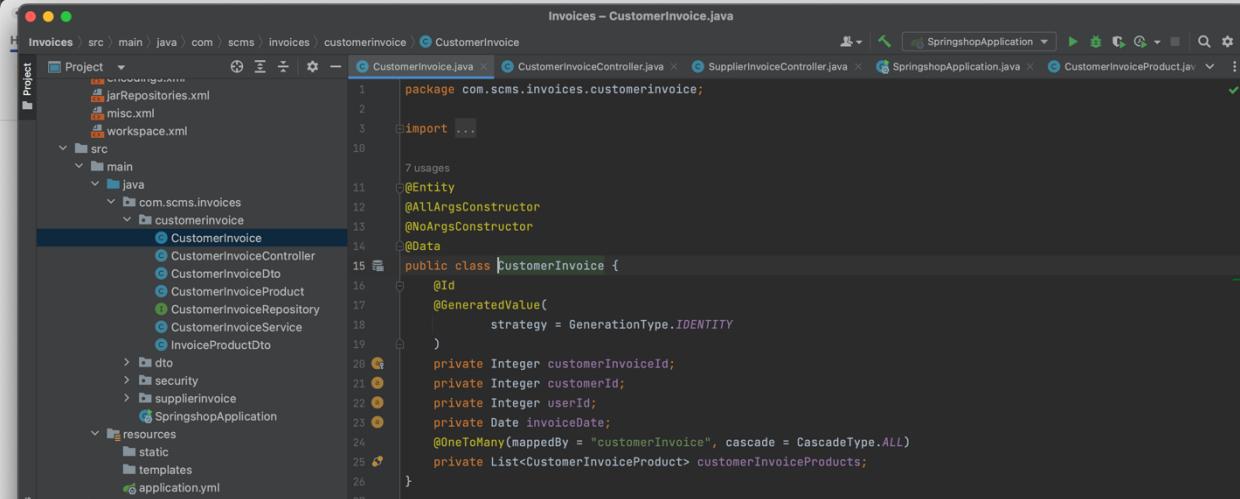
- **customerInvoice:** A foreign key that establishes a many-to-one relationship with the Customer Invoice entity. This relationship links each product to its corresponding invoice.
- **productId:** Represents the ID of the product associated with the invoice. This attribute identifies the product included 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

- **supplierInvoiceld:** 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.

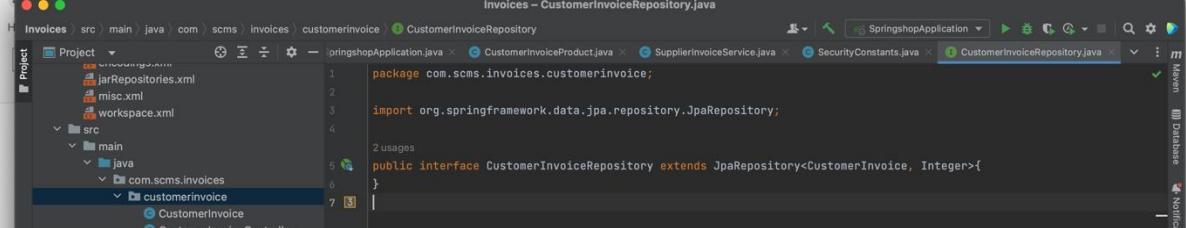


```

Invoices - CustomerInvoice.java
Invoices > src > main > java > com > scms > invoices > customerinvoice > CustomerInvoice.java

1 package com.scms.invoices.customerinvoice;
2
3 import ...
4
5 @Entity
6 @AllArgsConstructor
7 @NoArgsConstructor
8 @Data
9
10 public class CustomerInvoice {
11     @Id
12     @GeneratedValue(
13         strategy = GenerationType.IDENTITY
14     )
15     private Integer customerInvoiceId;
16     private Integer customerId;
17     private Integer userId;
18     private Date invoiceDate;
19     @OneToMany(mappedBy = "customerInvoice", cascade = CascadeType.ALL)
20     private List<CustomerInvoiceProduct> customerInvoiceProducts;
21 }
22
23
24
25
26
27
28
29

```



```

Invoices - CustomerInvoiceRepository.java
Invoices > src > main > java > com > scms > invoices > customerinvoice > CustomerInvoiceRepository.java

1 package com.scms.invoices.customerinvoice;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 public interface CustomerInvoiceRepository extends JpaRepository<CustomerInvoice, Integer>{
6 }
7

```

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.
- Database:**
 - Deletes the specified supplier invoice and associated product details from the database upon receiving the deletion request from the Supplier Invoice Service.
 - 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.