

**SPRING BOOT APPLICATION PROJECT DOCUMENTATION**

**CSCB532 Практика по програмиране и интернет технологии**

**Приложение „Логистична компания“**

Изготвил:

Свилен Арсенов F105329

**Table of Contents**

[**1. Introduction**](#_Introduction)

[**2. Prerequisities**](#_Development_Environment)

**3. Getting Started**

* **\*Project Setup**
* **\*Running the Application**

**4. Configuration**

**5. Usage**

* **\*API Endpoints**
* **\*Database Interaction**

**6. Testing**

**7. Deployment**

**8. Troubleshooting**

**9. Contributing**

**10. License**

Welcome to the documentation for the Logistic Company Application. This document provides comprehensive information on how to use and understand our Spring Boot-based logistics management application. Whether you are a developer, administrator, or end-user, this documentation will guide you through the setup, configuration, and usage of the application.

## **Introduction**

The Logistic Company Application is a logistics management system built using Spring Boot, designed to streamline the operations of a logistics company. It offers a range of features to efficiently manage tasks such as order processing, route planning, shipment tracking, and inventory management. With a user-friendly interface and powerful backend functionality, our application aims to optimize logistics operations, reduce costs, and enhance customer satisfaction.

### **Key Features**

* **Order Management**: Create, update, and track orders seamlessly.
* **Route Planning**: Optimize delivery routes for cost and time efficiency.
* **Shipment Tracking**: Real-time tracking of shipments for both customers and administrators.
* **Inventory Management**: Keep track of available stock and automate replenishment.
* **User Management**: Manage user roles, permissions, and access levels.
* **Reporting**: Generate reports and analytics for informed decision-making.

**Before you begin using the Logistic Company Application, please ensure that you meet the following prerequisites:**

## **Development Environment**

* **Java Development Kit (JDK)**: You should have Java SE Development Kit 8 (JDK 8) or a higher version installed on your development machine.
* **Integrated Development Environment (IDE)**: We recommend using a Java IDE such as IntelliJ IDEA, Eclipse, or Visual Studio Code for development.
* **Gradle**: Make sure it is installed on your system.
* **Git**: You may need Git for version control and to clone the application repository if necessary.
* **Database**: The Logistic Company Application may require a relational database such as MySQL, PostgreSQL, or H2 for storing data. Ensure you have the database installed and configured.
* **Spring Boot**: The application is built on the Spring Boot framework. Familiarize yourself with Spring Boot concepts and configuration.
* **Web Server**: Deploy the application on a web server like Apache Tomcat or use Spring Boot's embedded server for development purposes.

## **Software Dependencies**

* **Spring Boot Dependencies**: The project includes various Spring Boot dependencies. These are managed via Maven, and you don't need to install them manually.
* **External Libraries**: The application uses external libraries for features such as database connectivity, web APIs, and JSON serialization. These dependencies are resolved automatically through Maven.

## **Configuration**

* **Database Configuration**: You may need to configure the database connection details in the application's application.properties or application.yml file.
* **Logging**: Configure logging settings to capture application logs effectively.
* **Security**: Adjust security settings and authentication mechanisms as per your requirements.

## **Additional Considerations**

* **Network and Ports**: Ensure that the necessary network ports are open and accessible for running the application, especially if it interacts with external services or clients.
* **Security**: Implement appropriate security measures, such as firewalls and SSL certificates, to protect sensitive data.

With these prerequisites in place, you can proceed with the setup and configuration of the Logistic Company Application. Please refer to the subsequent sections of this documentation for detailed instructions on installation, configuration, and usage.

In this section, we will guide you through setting up the project and running the Spring Boot application.

## **Project Setup**

1. **Clone the repository**: Clone the project repository from GitHub using the following command:

**[shell]**

git clone https://github.com/Thurnos/CSCBJava.git

1. **Import into IDE**

Import the project into your preferred Integrated Development Environment (IDE). We recommend using IntelliJ IDEA or Eclipse for the best development experience.

1. **Build the project :**

Open the project in your IDE and build it using Maven. You can either use the IDE's built-in support for Maven or run the following command in your project directory

**[shell]**

mvn clean install

## **Running the Application**

1. **Run from IDE:**

In your IDE, find the main class of your Spring Boot application (typically annotated with @SpringBootApplication) and run it. This will start the application, and you should see log output in the console indicating that the application has started.

1. **Access the Application**

The Spring Boot application will start on the default port 8080. You can access it in your web browser or using tools like Postman by navigating to:

http://localhost:8080

## **Custom Configuration**

If you need to customize any application configuration properties, you can do so by editing the application.properties or application.yml file located in the src/main/resources directory. This file allows you to configure various settings such as database connection details, server port, and more.

## **Configuration**

In this section, we'll explore the various configuration aspects of the My Spring Boot Application. Configuration allows you to tailor the behavior of your application to meet specific requirements.

## **Application Configuration**

#### application.properties or application.yml

The application's configuration properties can be found in the application.properties or application.yml file located in the src/main/resources directory. This is where you can customize settings such as the server port, database connection, logging, and more.

* **Example: Customizing Server Port**

To change the server port from the default 8080 to a different port, modify the application.properties file:

**[properties]**

server.port=8081

* **Example: Database Configuration**

Configure the database connection settings using properties like spring.datasource.url, spring.datasource.username, and spring.datasource.password:

**[properties]**

spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase

spring.datasource.username=root

spring.datasource.password=secret

## **Profiles and Environment-Specific Configuration**

Spring Boot supports profiles that allow you to define different sets of configuration properties for various environments (e.g., development, production). Profiles can be used to manage environment-specific settings easily.

* **Example: Creating a dev Profile**

To create a development profile, add the following properties to your **templates/administration** file:

**[/resources/templates/administration]**

# Development Profile

spring.profiles.active=dev

## **Custom Bean Configuration**

You can configure custom beans and components in your Spring Boot application. Use annotations like @Configuration, @Bean, and @Component to define and customize these beans.

* **Example: Creating a Custom Bean**

@Configuration

public class MyConfig {

@Bean

public MyService myService() {

return new MyService();

}

}

## **Property Files and External Configuration**

You can externalize configuration properties into separate property files (e.g., .properties or .yml) and load them dynamically at runtime. This approach allows you to change configuration without modifying the application code.

* **Example: Loading External Properties**

Create an external property file (e.g., myapp.properties) and load it in your application:

@PropertySource("classpath:myapp.properties")

@Configuration

public class ExternalConfig {

// ...

}

## **Security Configuration**

The security configuration class SecurityConfig plays a pivotal role in defining how authentication and authorization are handled in our application. Here's an overview of its key components:

## **Password Encoding**

We define a BCryptPasswordEncoder bean to handle password encoding throughout the application. Passwords are hashed using bcrypt, enhancing security.

/\*\*

\* Defines a BCryptPasswordEncoder bean to be used for password encoding throughout the application.

\* @return A BCryptPasswordEncoder instance.

\*/

@Bean

public BCryptPasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

## **Authentication Provider**

We configure a custom authentication provider using the DaoAuthenticationProvider. This provider leverages the UserService to retrieve user details and utilizes the BCryptPasswordEncoder for password encoding.

## **HTTP Security Rules**

/\*\*

\* Configures a custom authentication provider using the provided UserService for user details retrieval

\* and a BCryptPasswordEncoder for password encoding.

\* @param userService The UserService to use for loading user details.

\* @return A configured DaoAuthenticationProvider.

\*/

public DaoAuthenticationProvider authenticationProvider(UserService userService) {

DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();

authenticationProvider.setUserDetailsService(userService); // set the custom user details service

authenticationProvider.setPasswordEncoder(passwordEncoder());

return authenticationProvider;

}

We define the HTTP security rules for the application using the HttpSecurity configuration. In the provided code snippet, we specify URL access rules, configure form login settings, and define the login page.

/\*\*

\* Configures the HTTP security for the application, specifying URL access rules,

\* form login configurations, and logout settings.

\* @param http The HttpSecurity to configure.

\* @return The SecurityFilterChain resulting from the configuration.

\* @throws Exception if an error occurs during configuration.

\*/

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http.authorizeHttpRequests(configurer ->

configurer

.requestMatchers("/user/registration").permitAll()

.requestMatchers("/\*\*").permitAll()

// .anyRequest().authenticated()

)

.formLogin(form -> form

.loginPage("/user/login")

.loginProcessingUrl("/authenticateTheUser")

.usernameParameter("email")

.passwordParameter("password")

.permitAll()

);

return http.build();

}

## **Usage**

In this section, we'll explore how to interact with the database tables of the My Spring Boot Application using MySQL queries. We'll cover simple queries for basic operations and more complex queries for advanced data retrieval and manipulation.

## **Delivery Type Management API Endpoints:**

This module is responsible for managing delivery types within the application, including creating new delivery types and retrieving all existing ones. It uses DeliveryTypeController for handling API requests and DeliveryTypeService for executing business logic.

## **Controller Layer**

### **DeliveryTypeController**

* **Base URL:** /api/delivery-types

#### **Endpoints**

1. **List All Delivery Types**
   * **URL:** /api/delivery-types
   * **Method:** GET
   * **Description:** Fetches all delivery types currently available in the database.
   * **Response:** A list of DeliveryType entities.
2. **Create New Delivery Type**
   * **URL:** /api/delivery-types
   * **Method:** POST
   * **Description:** Saves a new delivery type to the database based on the provided entity.
   * **Request Body:** DeliveryType entity (JSON format).
   * **Response:** The created DeliveryType entity, including its generated ID.

## **Service Layer**

### **DeliveryTypeService**

#### **Methods**

1. **getAllDeliveryTypes**
   * **Returns:** A list of all DeliveryType entities from the database. This method fetches the complete list of delivery types, making them available for further operations or display.
2. **createDeliveryType**
   * **Parameters:** DeliveryType entity to save.
   * **Returns:** The saved DeliveryType entity, now including a generated ID and any other modifications made during the persistence process.
3. **getAllWithDeliveryTypeDTO**
   * **Returns:** A list of DeliveryTypeDTO objects. This method assumes the existence of a method in DeliveryTypeRepository for fetching delivery types and converting them directly into DTOs, though it's not explicitly shown in the provided code snippets. If such functionality is desired, you will need to implement it in the repository layer, typically involving a custom query and a projection or constructor expression in JPQL to map entities directly to DTOs.

## **Client Management API Endpoints:**

The Client Management module provides functionalities to create, retrieve, update, and delete client information in the system. It leverages the ClientController for handling HTTP requests and the ClientService for business logic and data access.

## Controller Layer

### ClientController

* **Base URL:** /clients

#### Endpoints

1. **List All Clients**
   * **URL:** /clients/list
   * **Method:** GET
   * **Description:** Retrieves all clients from the database and displays them on a client list page.
   * **Returns:** A view named /client/list-clients populated with all clients.
2. **Create Client (JSON)**
   * **URL:** /clients
   * **Method:** POST
   * **Description:** Creates a new client based on JSON data provided in the request body.
   * **Request Body:** Client object (JSON format).
   * **Returns:** The created Client object.
3. **Create Client (Form Submission)**
   * **URL:** /clients/clients/create
   * **Method:** POST
   * **Description:** Creates a new client based on form data provided in the request.
   * **Request Body:** Client object (form data).
   * **Returns:** Redirects to /clients/list, displaying the updated client list.
4. **Update Client**
   * **URL:** /clients/update
   * **Method:** POST
   * **Description:** Updates an existing client based on form data provided in the request. The client is identified by its ID.
   * **Request Body:** Client object (form data) including the ID of the client to be updated.
   * **Returns:** Redirects to /clients/list, displaying the updated client list.
5. **Delete Client**
   * **URL:** /clients/delete
   * **Method:** DELETE
   * **Description:** Deletes a client identified by the id query parameter.
   * **Query Parameters:**
     + id: The ID of the client to delete.
   * **Returns:** A string indicating the client was deleted.

## Service Layer

### ClientService

#### Methods

1. **getAllClients**
   * **Returns:** A list of all Client entities stored in the database.
2. **createClient**
   * **Parameters:** Client object to be saved.
   * **Returns:** The saved Client entity, including any modifications made during the save process (e.g., generated ID).
3. **deleteClientById**
   * **Parameters:** clientId - The ID of the client to delete.
   * **Returns:** true if the client was found and deleted, false otherwise.
4. **saveClient**
   * **Parameters:** Client object to be updated or saved.
   * **Description:** Saves or updates a Client entity in the database.
5. **getAllClientsWithClientDTO**
   * **Returns:** A list of all clients converted to ClientDTO objects. (Assuming the method exists in ClientRepository to support this operation.)

## **Employee Management API Endpoints:**

This module is dedicated to managing employee records within the application. It encompasses creating new employee profiles, retrieving details of existing employees, updating employee information, and deleting employee records. The EmployeeController handles API requests related to employees, while EmployeeService executes the underlying business logic.

## Controller Layer

### EmployeeController

* **Base URL:** /employees

#### Endpoints

1. **List All Employees**
   * **URL:** /employees/list
   * **Method:** GET
   * **Description:** Retrieves a comprehensive list of all employees, displaying them on a dedicated list page.
   * **Response:** A view named /employee/list-employees, populated with employee details.
2. **Create New Employee (Form Submission)**
   * **URL:** /employees/create
   * **Method:** POST
   * **Description:** Submits a form to create a new employee record. The form data is processed and the new employee is saved to the database.
   * **Form Data:** Employee entity details.
   * **Response:** Redirects to /employees/list, showing the updated list of employees.
3. **Edit Employee**
   * **URL:** /employees/edit/{id}
   * **Method:** GET
   * **Description:** Fetches details of a specific employee for editing, based on the provided employee ID.
   * **Path Variable:** id - The ID of the employee to edit.
   * **Response:** A view named /employee/edit-employee, pre-filled with the employee's current information.
4. **Update Employee**
   * **URL:** /employees/edit
   * **Method:** POST
   * **Description:** Processes the submission of the edited employee form, updating the employee record in the database.
   * **Request Body:** UpdateEmployeeDTO containing updated employee details.
   * **Response:** Redirects to /employees/list, displaying the list of employees with updated information.
5. **Delete Employee**
   * **URL:** /employees/delete/{id}
   * **Method:** GET
   * **Description:** Deletes the employee record corresponding to the specified ID.
   * **Path Variable:** id - The ID of the employee to delete.
   * **Response:** Redirects to /employees/list, showing the current list of employees after deletion.

## Service Layer

### EmployeeService

#### Methods

1. **getAllEmployees**
   * **Returns:** A list of all Employee entities stored in the database. This method enables the retrieval of comprehensive employee details for display or further processing.
2. **createEmployee**
   * **Parameters:** Employee entity to be saved.
   * **Returns:** The newly saved Employee entity, complete with a generated ID and any other changes made during the save process.
3. **findEmployeesByCompanyId**
   * **Parameters:** companyId - The ID of the company to filter employees by.
   * **Returns:** A list of employees associated with the specified company.
4. **saveEmployee**
   * **Parameters:** Employee entity to be updated or saved.
   * **Description:** Persists the provided Employee entity to the database, updating existing records or creating new ones as necessary.
5. **deleteEmployee**
   * **Parameters:** id - The ID of the employee to remove.
   * **Description:** Deletes the specified employee from the database, based on the provided ID.

## **Logistic Company Management API Endpoints:**

This module facilitates managing logistic companies within the application, encompassing a range of operations such as retrieving logistic company details, creating new logistic company records, updating existing ones, and calculating delivery costs for specific date ranges. The LogisticCompanyController interfaces with the frontend to handle web requests, while LogisticCompanyService implements the core business logic.

## Controller Layer

### LogisticCompanyController

* **Base URL:** /logistic-companies

#### Endpoints

1. **List All Logistic Companies**
   * **URL:** /logistic-companies/list
   * **Method:** GET
   * **Description:** Displays a list of all logistic companies.
   * **Response:** Renders the list-logistic-companies view with logistic company details.
2. **Create Logistic Company**
   * **URL:** /logistic-companies/create
   * **Method:** GET
   * **Description:** Shows the form for creating a new logistic company.
   * **Response:** Renders the create-logistic-companies form view.
3. **Save Logistic Company**
   * **URL:** /logistic-companies/save
   * **Method:** POST
   * **Description:** Processes the submission of the logistic company creation form.
   * **Form Data:** LogisticCompany entity details.
   * **Response:** Redirects to /logistic-companies/list.
4. **Edit Logistic Company**
   * **URL:** /logistic-companies/edit/{id}
   * **Method:** GET
   * **Description:** Shows the edit form for a specific logistic company.
   * **Path Variable:** id - The ID of the logistic company to edit.
   * **Response:** Renders the edit-logistic-company form view.
5. **Update Logistic Company**
   * **URL:** /logistic-companies/edit
   * **Method:** POST
   * **Description:** Processes the submission of the logistic company edit form.
   * **Form Data:** LogisticCompanyDTO with updated details.
   * **Response:** Redirects to /logistic-companies/list.
6. **Delete Logistic Company**
   * **URL:** /logistic-companies/delete/{id}
   * **Method:** GET
   * **Description:** Deletes a specific logistic company.
   * **Path Variable:** id - The ID of the logistic company to delete.
   * **Response:** Redirects to /logistic-companies/list.
7. **Logistic Company Delivery Costs**
   * **URL:** /logistic-companies/logistic-companies/delivery-costs
   * **Method:** GET
   * **Description:** Retrieves logistic companies along with their delivery costs for a specified date range.
   * **Query Parameters:** startDate, endDate - The date range for calculating delivery costs.
   * **Response:** Returns a list of LogisticCompanyWithCostDTO detailing each company's delivery costs in the specified range.

## Service Layer

### LogisticCompanyService

#### Methods

1. **getAllWithLogisticCompanyDTO**
   * **Returns:** A list of LogisticCompanyDTO objects representing all logistic companies, sorted and converted to DTOs for presentation.
2. **createCompany**
   * **Parameters:** LogisticCompany entity to be saved.
   * **Returns:** The newly saved LogisticCompany entity, including any modifications made during the save process.
3. **findLogisticCompanyDeliveryCostsBetween**
   * **Parameters:** startDate, endDate - The date range for which to calculate delivery costs.
   * **Returns:** A list of LogisticCompanyWithCostDTO objects, each including the logistic company's ID, name, and total delivery costs calculated for the specified date range.
4. **findByIdWithLogisticCompanyDTO**
   * **Parameters:** id - The ID of the logistic company to find.
   * **Returns:** A LogisticCompanyDTO object for the specified logistic company.
5. **deleteCompany**
   * **Parameters:** companyId - The ID of the logistic company to delete.
   * **Description:** Deletes the specified logistic company from the database.
6. **saveCompany**
   * **Parameters:** LogisticCompany entity to update or save.
   * **Returns:** The updated or newly saved LogisticCompany entity.

## **Shipment Management API Endpoints:**

This module provides comprehensive management of shipment records within the application. It covers the creation of new shipments, listing all shipments, updating shipment details, and managing the shipment status. ShipmentController handles web requests for shipment operations, while ShipmentService implements the core business logic.

## Controller Layer

### ShipmentController

* **Base URL:** /shipments

#### Endpoints

1. **List All Shipments**
   * **URL:** /shipments/list
   * **Method:** GET
   * **Description:** Displays a list of all shipments.
   * **Response:** Renders the list-shipments view with shipment details.
2. **Create Shipment**
   * **URL:** /shipments/create
   * **Method:** GET
   * **Description:** Shows the form for creating a new shipment.
   * **Response:** Renders the create-shipment form view.
3. **Save Shipment**
   * **URL:** /shipments/create
   * **Method:** POST
   * **Description:** Processes the submission of the shipment creation form.
   * **Form Data:** Shipment entity details.
   * **Response:** Redirects to /shipments/list.
4. **Edit Shipment**
   * **URL:** /shipments/edit/{id}
   * **Method:** GET
   * **Description:** Shows the form for editing an existing shipment.
   * **Path Variable:** id - The ID of the shipment to edit.
   * **Response:** Renders the edit-shipment form view.
5. **Update Shipment**
   * **URL:** /shipments/edit
   * **Method:** POST
   * **Description:** Processes the submission of the shipment edit form.
   * **Form Data:** Updated Shipment entity details.
   * **Response:** Redirects to /shipments/list.
6. **Delete Shipment**
   * **URL:** /shipments/delete/{id}
   * **Method:** GET
   * **Description:** Deletes a specific shipment.
   * **Path Variable:** id - The ID of the shipment to delete.
   * **Response:** Redirects to /shipments/list.
7. **List Not-Delivered Shipments**
   * **URL:** /shipments/not-delivered
   * **Method:** GET
   * **Description:** Displays a list of shipments that have not been delivered yet.
   * **Response:** Renders a view listing shipments with a status other than "Delivered".

## Service Layer

### ShipmentService

#### Methods

1. **getAllShipments**
   * **Returns:** A list of all Shipment entities from the database, converted to ShipmentDTO for presentation.
2. **createShipment**
   * **Parameters:** Shipment entity to be saved.
   * **Returns:** The newly saved Shipment entity, including any modifications made during the save process.
3. **findShipmentsNotDelivered**
   * **Returns:** A list of ShipmentDTO objects representing shipments that have not been delivered, based on shipment status.
4. **updateShipment**
   * **Parameters:** Updated Shipment entity details.
   * **Returns:** The updated Shipment entity.
5. **deleteShipment**
   * **Parameters:** shipmentId - The ID of the shipment to delete.
   * **Description:** Removes the specified shipment from the database.

## **User Management API Endpoints:**

The User Management module is designed to facilitate operations related to user accounts within the application. It includes functionalities for listing users, creating new user accounts, editing user details, and assigning roles to users. The UserController manages web requests for these operations, while UserService implements the core business logic.

## Controller Layer

### UserController

* **Base URL:** /users

#### Endpoints

1. **Create User**
   * **URL:** /users
   * **Method:** POST
   * **Description:** Creates a new user with the specified details provided in the request body.
   * **Request Body:** User entity (JSON format).
   * **Response:** The created User object.
2. **List All Users**
   * **URL:** /users/list
   * **Method:** GET
   * **Description:** Displays a list of all users along with their assigned roles.
   * **Response:** Renders the list-users view with user and role details.
3. **Edit User**
   * **URL:** /users/edit/{id}
   * **Method:** GET
   * **Description:** Shows the form for editing a user's details, identified by their ID.
   * **Path Variable:** id - The ID of the user to edit.
   * **Response:** Renders the edit-user form view with pre-filled user details and available roles.
4. **Update User**
   * **URL:** /users/update
   * **Method:** POST
   * **Description:** Updates a user with the given details submitted through the form.
   * **Form Data:** Updated User entity details.
   * **Response:** Redirects to the list-users view.

## Service Layer

### UserService

#### Methods

1. **createUser**
   * **Parameters:** User entity to be created.
   * **Returns:** The newly created User entity, including any automatic updates applied during the save process (e.g., generated ID).
2. **findALlUsersWithRoles**
   * **Returns:** A list of UserRolesDTO objects containing users and their associated roles. This method retrieves all users from the repository, enriching them with role information for presentation.
3. **findUserWithRoles**
   * **Parameters:** id - The ID of the user.
   * **Returns:** A UserRolesDTO object containing detailed information about the user and their roles, based on the provided user ID.
4. **saveUser**
   * **Parameters:** User entity to be updated or saved.
   * **Description:** Persists the provided User entity to the database, updating existing records or creating new ones as necessary.

## **Basic Queries**

To retrieve all clients from the clients table, use the following MySQL query , or replace clients with any other table name you are looking to extract

**[sql]**

SELECT \* FROM clients;

## **Inserting a New Client**

To insert a new client into the clients table, you can use the following MySQL query as an example:

**[sql]**

INSERT INTO clients (users\_user\_id, client\_first\_name, client\_last\_name, client\_address, client\_phone\_number, logistic\_company\_id)

VALUES (1, 'John', 'Doe', '123 Main St', '+1-555-123-4567', 1001);

## **More Complex Queires**

To retrieve clients of a specific logistic company (e.g., company with logistic\_company\_id 1001), you can use a SQL JOIN query:

**[sql]**

SELECT c.\*

FROM clients c

INNER JOIN logistic\_companies lc ON c.logistic\_company\_id = lc.logistic\_company\_id

WHERE lc.logistic\_company\_id = 1001;

To retrieve information of all Shipments that have been sent but have NOT been Delivered yet:

**[sql]**

SELECT \*

FROM shipments s

INNER JOIN shipment\_status\_catalog ssc ON s.shipment\_status\_id = ssc.shipment\_status\_catalog\_id

WHERE ssc.shipment\_status\_name != ‘Delivered’;

To retrieve information of a company’s income in a time frame:

[To calculate the income of each company , we are calculating the delivery cost of each sent shipment by this company]

The following query is calculating the total income of company with ID -> 1 in the frame between 2023-2024

**[sql]**

SELECT

lc.logistic\_company\_id AS company\_id,

lc.logistic\_company\_name,

SUM(s.shipment\_total) AS total\_delivery\_cost

FROM

shipments s

INNER JOIN logistic\_companies lc ON s.logistic\_company\_id = lc.logistic\_company\_id

WHERE

s.shipment\_created BETWEEN '2023-01-01' AND '2024-03-31'

GROUP BY

lc.logistic\_company\_id,

lc.logistic\_company\_name;

## **Updating Client Infor**

## **mation**

To update the information of a client (e.g., client with users\_user\_id 1), you can use an UPDATE query:

**[sql]**

UPDATE clients

SET client\_address = '456 Elm St', client\_phone\_number = '+1-555-987-6543'

WHERE users\_user\_id = 1;

## **Deleting a Client**

To delete a client (e.g., client with users\_user\_id 2), use a DELETE query:

## **Getting all the information for an employee , working in a company**

**[sql]**

DELETE FROM clients WHERE users\_user\_id = 2;

**[sql]**

SELECT

e.users\_user\_id,

e.employee\_first\_name,

e.employee\_last\_name,

e.employee\_birth\_date,

pc.position\_name,

ob.office\_branch\_name,

lc.logistic\_company\_name

FROM

employees e

JOIN

logistic\_companies lc ON e.logistic\_company\_id = lc.logistic\_company\_id

JOIN

office\_branches ob ON e.office\_branch\_id = ob.office\_branch\_id

JOIN

positions\_catalog pc ON e.employee\_position = pc.position\_catalog\_id

WHERE

lc.logistic\_company\_id = 1; -- Replace ? with the company ID you're interested in

## **Testing**

In this section, we'll discuss the testing strategies and specific tests for our My Spring Boot Application. Proper testing ensures that your application functions as expected and helps identify and fix issues early in the development process.

## **User Integration Tests**

#### Creating a User <a name="create-user"></a>

To test the creation of a user, we use an integration test that sends a POST request to the /api/users endpoint to create a new user. The test verifies that the user is successfully created with the expected properties.

* **Test Class:** UserControllerIntegrationTest
* **Method:** testCreateUser

#### Retrieving All Users <a name="retrieve-all-users"></a>

To test retrieving all users from the /api/users endpoint, we use an integration test that sends a GET request and expects a list of users in the response. The test checks if the returned list contains the expected number of users.

* **Test Class:** UserControllerIntegrationTest
* **Method:** testGetAllUsers

## **User Role Controller Tests**

#### Creating a User Role <a name="create-user-role"></a>

Testing the creation of a user role involves sending a POST request to the /api/user-roles endpoint and verifying that the role is created with the expected attributes.

* **Test Class:** UserRoleControllerTest
* **Method:** testCreateUserRole

## **Shipment Status Catalog Controller Tests**

#### Creating a Shipment Status Catalog <a name="create-shipment-status-catalog"></a>

In this test, we send a POST request to the /api/shipment-status-catalogs endpoint to create a new shipment status catalog entry. We ensure that the created entry matches the expected values.

* **Test Class:** ShipmentStatusCatalogControllerTest
* **Method:** testCreateShipmentStatusCatalog

## **Employee Controller Tests**

#### Retrieving All Employees <a name="retrieve-all-employees"></a>

This integration test checks the /api/employees endpoint to retrieve all employees. It verifies that the response contains the expected number of employees and their attributes.

* **Test Class:** EmployeeControllerTest
* **Method:** testGetAllEmployees

#### Creating an Employee <a name="create-employee"></a>

To test the creation of an employee, this integration test sends a POST request to the /api/employees endpoint and ensures that the created employee matches the provided attributes.

* **Test Class:** EmployeeControllerTest
* **Method:** testCreateEmployee

## **Office Branch Controller Tests**

#### Creating an Office Branch <a name="create-office-branch"></a>

This test focuses on creating an office branch using the /api/office-branches endpoint. It verifies that the created office branch matches the provided attributes.

* **Test Class:** OfficeBranchControllerTest
* **Method:** testCreateOfficeBranch

## **Client Service Test**

#### Retrieving All Clients <a name="retrieve-all-clients-service"></a>

In this service test, we retrieve all clients using the ClientService and validate that the returned list contains the expected clients.

* **Test Class:** ClientServiceTest
* **Method:** testGetAllClients