Logistic Company Project Documentation

Overview

The Logistic Company project is a Java-based application designed to manage logistics operations, including package handling, user management, and office administration. It follows a structured package organization for maintainability and scalability.

Project Structure

1. Configuration (config)

This package contains configuration classes essential for application setup.

- **ApplicationBeanConfiguration** – Manages application-wide bean configurations.
- **ApplicationSecurityConfiguration** – Handles security-related configurations.

2. Initialization (init)

- **AppInitializer** – Initializes the application settings and configurations.

3. Model (model)

This package contains data representation objects.

- **binding** – Contains data binding models:
  - PackageAddBindingModel
  - UserRegisterBindingModel
- **dto** – Data Transfer Objects (DTOs) for various entities:
  - LogisticCompanyDto
  - OfficeDto
  - PackageDto
  - RoleDto
  - UserDto
- **enums** – Enumeration classes for predefined constant values:
  - PackagePaidStatus
  - PackageType
  - Role
  - State
- **service** – Service models for business logic representation:

    - PackageServiceModel
    - UserServiceModel
- **view** – View models for data presentation:
    - ClientPackageDetailsView
    - UserBalanceViewModel
    - UserViewModel
- **Entities:**
    - BaseEntity
    - LogisticCompany
    - Office
    - Package
    - RoleEntity
    - UserEntity

4. Repository (repository)

Handles database interactions using Spring Data JPA.

- LogisticCompanyRepository
- OfficeRepository
- PackageRepository
- RoleRepository
- UserRepository

5. Service (service)

Contains business logic implementations.

- **exceptions** – Handles application exceptions.
- **impl** – Implements business services:
    - LogisticCompanyServiceImpl
    - LogisticCompanyUserServiceImpl
    - OfficeServiceImpl
    - PackageServiceImpl
    - RoleServiceImpl
    - UserServiceImpl
- **Interfaces**:
    - LogisticCompanyService
    - OfficeService
    - PackageService
    - RoleService
    - UserService

6. Web (web)

Contains controllers for handling HTTP requests.

- **handler** – Manages request handling:

    - HomeController
    - PackageController
    - UserController
    - UserInfoRestController

7. Resources (resources)

Contains static files and templates for the application.

- **static** – Stores static assets such as CSS, JavaScript, and images.
- **templates** – Stores HTML templates for frontend rendering.
- **application.properties** – Configuration properties for the application.

8. Test (test)

Contains unit tests for the application.

- **Test classes:**
    - LogisticCompanyApplicationTests
    - LogisticCompanyServiceTests
    - LogisticCompanyUserServiceTests
    - OfficeServiceTests
    - PackageServiceTests
    - UserServiceTests

Detailed Description

1. Config

a. ApplicationBeanConfiguration Class

Overview

The ApplicationBeanConfiguration class is a Spring configuration class that defines essential beans for the application.

Annotations

- **@Configuration**: Marks this class as a Spring configuration class.

Beans Provided

ModelMapper

- **Purpose**: Simplifies object mapping between DTOs and entities.
- **Usage**: Injected in services to transform objects.

PasswordEncoder

- **Implementation**: Pbkdf2PasswordEncoder
- **Purpose**: Securely hashes user passwords.
- **Usage**: Injected in authentication services for password encryption.

b. ApplicationSecurityConfiguration Class

Overview

The ApplicationSecurityConfiguration class is responsible for configuring security settings in the application. It defines authentication and authorization rules, login/logout configurations, and user role-based access controls.

Annotations

- **@Configuration**: Declares this class as a Spring configuration component for security settings.

Key Features

Authentication & Authorization

- **Uses**: UserDetailsService for authentication.
- **Password Encoding**: Secured using PasswordEncoder.
- **Role-Based Access**:
  - Public access to login, register, and home pages.
  - Admin-only access to company and employee management endpoints.
  - Authenticated access to all other resources.

Login & Logout

- **Login Page**: Custom login form at /users/login.
- **Success Redirect**: Redirects authenticated users to the home page.
- **Logout Settings**:
  - Logs out users via /users/logout.
  - Invalidates session and deletes cookies.

2. Init

a. AppInitializer Class

Overview

The AppInitializer class is responsible for initializing essential application data during startup. It ensures that roles, users, packages, offices, and logistic companies are properly set up if they do not already exist in the database.

Annotations

- **@Component**: Marks this class as a Spring component, allowing it to be automatically detected and managed by Spring's dependency injection.

Dependencies

The class relies on the following repositories and services:

- **Repositories**: RoleRepository, UserRepository, PackageRepository, OfficeRepository, LogisticCompanyRepository
- **Services**: RoleService, UserService, PackageService, OfficeService, LogisticCompanyService

Initialization Process

The run method checks if the database is empty for each entity and calls the corresponding service to initialize default data (test data):

1. **Roles**: Initialized by roleService.initializeRoles().
2. **Users**: Initialized by userService.initializeUsers().
3. **Packages**: Initialized by packageService.initializePackages().
4. **Logistic Companies**: Initialized by logisticCompanyService.initializeLogisticCompanies().
5. **Offices**: Initialized by officeService.initializeOffices().

3. Model

a. Binding

PackageAddBindingModel Class

Overview

The PackageAddBindingModel class is a data binding model used for collecting user input when adding a new package. It ensures that the input meets validation requirements before being processed.

Annotations and Validations

- **@Size(min = 3, max = 20) @NotBlank**: Ensures the address is between 3 and 20 characters and is not empty.
- **@Positive @NotNull**: Ensures weight and price are positive numbers and not null.
- **@NotNull**: Ensures the package type is selected.

Fields

- **Address**: The destination address of the package.
- **Weight**: The weight of the package, must be a positive value.
- **Price**: The cost of the package, must be a positive value.
- **Package Type**: Enum representing the type of package.

UserRegisterBindingModel Class

Overview

The UserRegisterBindingModel class is a data binding model used for user registration, ensuring valid input data before processing.

Annotations and Validations

- **@Size(min = 5, max = 20) @NotBlank**: Ensures the username is between 5 and 20 characters and is not empty.
- **@Size(min = 3, max = 20) @NotBlank**: Ensures first name, last name, password, and confirm password meet length requirements and are not empty.
- **@Email @NotBlank**: Ensures the email is valid and not empty.
- **@Positive @NotNull**: Ensures age is a positive number and not null.
- **@DateTimeFormat(pattern = "yyyy-MM-dd") @PastOrPresent @NotNull**: Ensures the birth date is in the past or present and is not null.
- **@NotNull @NotEmpty**: Ensures the country is provided.

Fields

- **Username**: The user's chosen username.
- **First Name**: The user's first name.
- **Last Name**: The user's last name.

- **Password**: The user's password.
- **Confirm Password**: Field for confirming the password.
- **Email**: The user's email address.
- **Age**: The user's age, must be positive.
- **Born On**: The user's birth date.
- **Country**: The user's country of residence.

b. DTO

LogisticCompanyDto Class

Overview

The LogisticCompanyDto class is a Data Transfer Object (DTO) used for transferring logistic company data between different layers of the application.

Fields

- **Name**: The name of the logistic company.
- **Revenue**: The total revenue of the logistic company.
- **Offices**: A list of OfficeDto objects representing the company's offices.
- **User Entities**: A list of UserDto objects representing users associated with the company.

Usage

- Used to transfer logistic company data between the service and controller layers.
- Helps in reducing unnecessary entity exposure.

OfficeDto Class

Overview

The OfficeDto class is a Data Transfer Object (DTO) used for transferring office-related data between different layers of the application.

Fields

- **Address**: The physical address of the office.
- **Phone**: The contact phone number of the office.
- **LogisticCompanyDto**: A reference to the logistic company to which the office belongs.
- **Packages**: A list of PackageDto objects representing the packages associated with the office.
- **User Entities**: A list of UserDto objects representing users working in the office.

Usage

- Used to transfer office-related data between the service and controller layers.
- Helps in reducing unnecessary entity exposure and improving data encapsulation.

PackageDto Class

Overview

The PackageDto class is a Data Transfer Object (DTO) used for encapsulating and transferring package-related data across different layers of the application.

Fields

- **Sender (UserDto)**: The user who sends the package.
- **Receiver (UserDto)**: The user who receives the package.
- **Courier (UserDto)**: The courier responsible for delivering the package.
- **Address (String)**: The delivery address of the package.
- **Weight (Double)**: The weight of the package.
- **Price (Double)**: The price charged for the package delivery.
- **Registration Date (LocalDate)**: The date when the package was registered in the system.
- **Arrival Date (LocalDate)**: The estimated or actual date of arrival at its destination.
- **State (State)**: The current status of the package.
- **Type (PackageType)**: The category of the package.
- **Package Paid Status (PackagePaidStatus)**: Indicates whether the package has been paid for or not.

Usage

- Ensures structured and efficient data transfer between the service and controller layers.
- Reduces direct entity exposure, improving system security and encapsulation.
- Helps in converting package-related data into a standardized format for further processing.

RoleDto Class

Overview

The RoleDto class is a Data Transfer Object (DTO) that encapsulates role-related data, primarily representing user roles in the system.

Fields

- **Role (Role)**: Enum value representing the assigned role of a user (e.g., ADMIN, USER, COURIER).

Usage

- Used to transfer role-related data between different layers of the application.
- Helps in managing user permissions and access levels.
- Encapsulates role information, preventing direct exposure of internal entities.

UserDto Class

Overview

The UserDto class is a Data Transfer Object (DTO) that encapsulates user-related data for secure and structured data exchange within the application.

Fields

- **Username (String)**: The unique identifier for the user.
- **First Name (String)**: The first name of the user.
- **Last Name (String)**: The last name of the user.
- **Password (String)**: The user's password (should be stored securely).
- **Email (String)**: The email address of the user.
- **Balance (Double)**: The account balance associated with the user.
- **Age (Integer)**: The age of the user.
- **Born On (LocalDate)**: The birth date of the user.
- **Country (String)**: The country of residence for the user.
- **Roles (Set<RoleDto>)**: A collection of roles assigned to the user, used for managing access control.

Usage

- Facilitates secure user data exchange between different layers of the application.
- Helps in managing user profiles, authentication, and authorization.
- Prevents direct exposure of entity models, enhancing data encapsulation.

c. enums

PackagePaidStatus Enum

Overview

The PackagePaidStatus enum represents the payment status of a package.

Values

- **PAID**: The package has been paid for.
- **NON_PAID**: The package has not been paid for.

PackageType Enum

Overview

The PackageType enum defines the different statuses of a package during its lifecycle.

Values

- **SENT**: The package has been sent from the sender.
- **ACCEPTED**: The package has been received at its destination.

Role Enum

Overview

The Role enum defines different roles that users can have within the system.

Values

- **ADMIN**: A user with administrative privileges.
- **OFFICE_EMPLOYEE**: A user working at a logistics office.
- **COURIER_EMPLOYEE**: A courier responsible for package delivery.
- **CLIENT**: A customer using the logistics services.

State Enum

Overview

The State enum defines the delivery status of a package.

Values

- **DELIVERED**: The package has been successfully delivered.
- **NOT_DELIVERED**: The package has not been delivered.

d. Service

PackageServiceModel Class

Overview

The PackageServiceModel class is a service model used for handling package-related business logic and data transfer within the service layer.

Fields

- **id**: The unique identifier of the package.
- **address**: The delivery address of the package.
- **weight**: The weight of the package in kilograms.
- **price**: The cost of shipping the package.
- **PackageType**: The type of package, as defined in the PackageType enum.

Usage

- Used for internal business logic and service-level operations related to packages.
- Acts as a bridge between the database entities and DTOs.

UserServiceModel Class

Overview

The UserServiceModel class is a service model used for handling user-related business logic and data transfer within the service layer.

Fields

- **id**: The unique identifier of the user.
- **username**: The username of the user.
- **firstName**: The first name of the user.
- **lastName**: The last name of the user.
- **password**: The encrypted password of the user.
- **email**: The email address of the user.
- **age**: The age of the user.
- **bornOn**: The birthdate of the user.
- **country**: The country of residence of the user.

Usage

- Used for internal business logic and service-level operations related to users.
- Helps in transforming user data between entities and DTOs while maintaining security and efficiency.

d. View

ClientPackageDetailsView Class

Overview

The ClientPackageDetailsView class is a view model that represents package details visible to clients.

Fields

- **id**: The unique identifier of the package.
- **receiver**: The recipient's name.
- **courier**: The assigned courier's name.
- **address**: The delivery address of the package.
- **weight**: The weight of the package in kilograms.
- **price**: The cost of shipping the package.
- **registrationDate**: The date when the package was registered in the system.
- **arrivalDate**: The estimated or actual date of arrival.
- **state**: The current state of the package (e.g., DELIVERED, NOT_DELIVERED).
- **type**: The type of package (e.g., SENT, ACCEPTED).
- **packagePaidStatus**: The payment status of the package (PAID or NON_PAID).

Usage

- Used to display package details to clients in the user interface.

UserBalanceViewModel Class

Overview

The UserBalanceViewModel class is a view model that represents a user's account balance.

Fields

- **balance**: The available balance in the user's account.

Usage

- Used to display the user's balance in the frontend.
- Supports fluent setter methods for easier modification.

UserViewModel Class

Overview

The UserViewModel class is a view model that represents the user's basic information.

Fields

- **username**: The unique username of the user.
- **firstName**: The first name of the user.
- **lastName**: The last name of the user.

- **email**: The email address of the user.
- **age**: The age of the user.
- **bornOn**: The birthdate of the user.
- **country**: The country of residence of the user.

Usage

- Used to display user-related information in the frontend.
- Supports fluent setter methods for easier modification.

BaseEntity Class

Overview

The BaseEntity class is a superclass for all entity models, providing a common identifier.

Fields

- **id**: A unique identifier, automatically generated using GenerationType.IDENTITY.

Annotations

- @MappedSuperclass: Indicates that this class is a superclass that other entity classes inherit from.
- @Id: Marks id as the primary key.
- @GeneratedValue(strategy = GenerationType.IDENTITY): Specifies that the ID is auto-incremented by the database.

Usage

- All entity classes should extend BaseEntity to inherit the id field.

LogisticCompany Class

Overview

The LogisticCompany class represents a logistics company, which has multiple offices and users.

Fields

- **name**: The name of the logistics company (unique and required).

- **revenue**: The total revenue generated by the company.
- **offices**: A list of offices associated with the company.
- **userEntities**: A list of users associated with the company.

Annotations

- @Entity: Marks this class as a JPA entity.
- @Table(name = "logistic_companies"): Specifies the database table name.
- @Column(name = "name", nullable = false, unique = true): Ensures name is unique and required.
- @OneToMany(fetch = FetchType.EAGER): Defines a one-to-many relationship with Office.
- @OneToMany: Defines a one-to-many relationship with UserEntity.

Usage

- Used to store and manage logistic company information.

Office Class

Overview

The Office class represents a logistics office that belongs to a logistic company and handles packages and users.

Fields

- **phone**: The contact number of the office (required).
- **address**: The physical address of the office (required).
- **logisticCompany**: The company that owns the office.
- **packages**: A list of packages handled by the office.
- **userEntities**: A list of users working at the office.

Annotations

- @Entity: Marks this class as a JPA entity.
- @Table(name = "offices"): Specifies the database table name.
- @Column(name = "phone", nullable = false): Ensures phone is required.
- @Column(name = "address", nullable = false): Ensures address is required.
- @ManyToOne: Defines a many-to-one relationship with LogisticCompany.
- @OneToMany: Defines a one-to-many relationship with Package and UserEntity.

Usage

- Used to store and manage office-related data in the system.

Package Class

Overview

The Package class represents a shipment handled by the logistics system.

Fields

- **sender**: The user who sends the package.
- **receiver**: The user who receives the package.
- **courier**: The courier responsible for delivering the package.
- **address**: The delivery address (required).
- **weight**: The weight of the package (required).
- **price**: The cost of shipping.
- **registrationDate**: The date the package was registered.
- **arrivalDate**: The estimated arrival date.
- **state**: The delivery status of the package.
- **type**: The type of package.
- **packagePaidStatus**: The payment status of the package.

Usage

- Used to track and manage package-related data.

RoleEntity Class

Overview

The RoleEntity class represents a user role in the system.

Fields

- **role**: The role assigned to a user (e.g., ADMIN, USER).

Usage

- Used to manage user roles within the system.

UserEntity Class

Overview

The UserEntity class represents a system user, such as customers, couriers, and administrators.

Fields

- **username**: The unique username of the user (required).
- **firstName**: The user's first name (required).
- **lastName**: The user's last name.
- **password**: The user's password (required).
- **email**: The unique email address of the user (required).
- **balance**: The available balance for package payments.
- **age**: The user's age.
- **bornOn**: The user's date of birth.
- **country**: The country of the user.
- **roles**: A set of roles assigned to the user.

Usage

- Used to store and manage user-related data in the system.

4. Repository Layer

Overview

The repository layer provides an abstraction for database operations, leveraging Spring Data JPA to interact with the persistence layer. It contains interfaces that define database access methods for different entities.

LogisticCompanyRepository

Overview

Handles database operations for LogisticCompany entities.

Methods

- **Extends JpaRepository<LogisticCompany, Long>**: Provides CRUD operations for LogisticCompany.

Annotations

- @Repository: Marks this interface as a Spring Data repository.

OfficeRepository

Overview

Handles database operations for Office entities.

Methods

- **Extends JpaRepository<Office, Long>**: Provides CRUD operations for Office.

Annotations

- @Repository: Marks this interface as a Spring Data repository.

PackageRepository

Overview

Handles database operations for Package entities, providing query methods for filtering packages based on different criteria.

Methods

- findAllByType(PackageType packageType): Retrieves all packages of a specific type.
- findAllByStateAndType(State state, PackageType type): Retrieves packages filtered by state and type.
- findAllByStateAndTypeAndReceiver_Id(State state, PackageType type, Long receiverId): Retrieves packages filtered by state, type, and receiver ID.
- findAllByTypeAndSender_Id(PackageType type, Long senderId): Retrieves packages of a specific type sent by a particular user.
- findAllBySender_Username(String senderUsername): Retrieves all packages sent by a user based on username.
- findAllByTypeAndReceiver_Id(PackageType type, Long receiverId): Retrieves all packages of a specific type received by a user.

Annotations

- @Repository: Marks this interface as a Spring Data repository.

RoleRepository

Overview

Handles database operations for RoleEntity, which represents user roles.

Methods

- findByRole(Role roleEnum): Retrieves a role entity by role type.

Annotations

- @Repository: Marks this interface as a Spring Data repository.

UserRepository

Overview

Handles database operations for UserEntity, allowing retrieval of users based on various attributes.

Methods

- findByUsername(String username): Retrieves a user by their username.
- findByEmail(String email): Retrieves a user by their email.
- findUserEntitiesByRoles(Set<Role> roles): Retrieves users who have specific roles using a custom JPQL query.

Annotations

- @Repository: Marks this interface as a Spring Data repository.
- @Query: Custom query to fetch users by roles.

Usage

The repository interfaces are used by the service layer to perform database operations efficiently without requiring manual SQL queries.

5. Service Layer

Overview

The service layer contains the business logic of the application. It interacts with the repository layer to perform operations on database entities and provides data processing before returning results to the controllers.

LogisticCompanyService

Overview

Handles operations related to LogisticCompany, including creation, retrieval, updating, and deletion.

Methods

- createCompany(LogisticCompanyDto companyDto): Creates a new logistic company.
- getCompanies(): Retrieves a list of all logistic companies.
- updateCompany(LogisticCompanyDto companyDto, Long id): Updates an existing logistic company.
- deleteCompany(Long id): Deletes a logistic company by ID.
- getRevenueForTimePeriod(Long companyId, LocalDate start, LocalDate end): Calculates the revenue of a company for a specific period.
- initializeLogisticCompanies(): Initializes logistic companies in the system.

OfficeService

Overview

Handles operations related to Office, including creation, retrieval, updating, and deletion.

Methods

- createOffice(OfficeDto officeDto): Creates a new office.
- getOffices(): Retrieves a list of all offices.
- updateOffice(OfficeDto officeDto, Long id): Updates an existing office.
- deleteOffice(Long id): Deletes an office by ID.
- initializeOffices(): Initializes offices in the system.

PackageService

Overview

Handles operations related to Package, including package creation, retrieval, updating, and status management.

Methods

- createPackage(PackageServiceModel packageServiceModel, String userName): Creates a new package.
- getPackages(): Retrieves a list of all packages.
- updatePackage(PackageDto packageDto, Long id): Updates an existing package.
- deletePackage(Long id): Deletes a package by ID.
- getAllRegisteredPackages(): Retrieves all registered packages.
- getAllRegisteredPackagesByReceiver(Long receiverId): Retrieves all registered packages for a specific receiver.
- getAllRegisteredNotDeliveredPackages(): Retrieves all registered packages that are not delivered.
- getAllSentNotDeliveredPackages(): Retrieves all sent packages that are not delivered.
- getAllPackagesSentByClient(Long clientId): Retrieves all packages sent by a specific client.
- getAllPackagesReceivedByClient(Long clientId): Retrieves all packages received by a specific client.
- calculatePrice(Package pack): Calculates the price of a package.
- acceptPackage(Long packageId, String employeeUsername): Marks a package as accepted by an employee.
- initializePackages(): Initializes packages in the system.
- findAllClientPackagesDetails(String username): Retrieves all package details for a specific client.

RoleService

Overview

Handles role management in the system.

Methods

- initializeRoles(): Initializes roles in the system.

UserService

Overview

Handles operations related to UserEntity, including user registration, retrieval, updating, and balance management.

Methods

- createUser(UserDto userDto): Creates a new user.
- getUsers(): Retrieves a list of all users.
- updateUser(UserDto userDto, Long id): Updates an existing user.
- deleteUser(Long id): Deletes a user by ID.
- getAllEmployees(): Retrieves all employees.
- getAllClients(): Retrieves all clients.
- pay(String username, Long packageId): Processes a payment for a package.
- isUserExistingByEmailOrUsername(String email, String username): Checks if a user exists by email or username.
- registerUser(UserServiceModel userServiceModel): Registers a new user.
- initializeUsers(): Initializes users in the system.
- getLoggedUserInfo(String name): Retrieves balance information for a logged-in user.
- getUserInfo(String name): Retrieves general user information.
- addMoneyToUser(String name): Adds money to a user's account.

Usage

The service layer is used by the controllers to execute business logic and interact with the persistence layer efficiently.

a. Implementation Layer (Impl)

Overview

The implementation layer provides concrete implementations for service interfaces. These implementations contain business logic, database interactions, and logging.

LogisticCompanyServiceImpl

Overview

Implements the LogisticCompanyService interface to manage logistic company-related operations, including CRUD operations and revenue calculations.

Dependencies

- LogisticCompanyRepository: Interacts with the database for LogisticCompany entities.
- OfficeRepository: Manages office-related operations for a logistic company.
- UserRepository: Provides user data related to the logistic company.
- ModelMapper: Converts DTOs to entities and vice versa.
- Logger: Logs important actions and events.

Methods

- **createCompany(LogisticCompanyDto companyDto)**
  - Converts a DTO to a LogisticCompany entity.
  - Saves the company to the database.
  - Logs the creation event.
- **getCompanies()**
  - Retrieves all logistic companies from the database.
  - Maps each entity to a DTO.
- **updateCompany(LogisticCompanyDto companyDto, Long id)**
  - Finds a logistic company by ID.
  - Updates the entity with new values from the DTO.
  - Saves the updated company to the database.
  - Logs the update event.
- **deleteCompany(Long id)**
  - Deletes a logistic company by ID.
  - Logs the deletion event.
- **getRevenueForTimePeriod(Long companyId, LocalDate start, LocalDate end)**
  - Finds a logistic company by ID.
  - Calculates total revenue for the given period by summing package prices.
  - Returns null if the company is not found.
- **initializeLogisticCompanies()**
  - Retrieves predefined employees from the database.
  - Creates a sample logistic company (Speedy).
  - Associates employees with the company.
  - Saves the company to the database.

Usage

This implementation is used by controllers to execute business logic related to logistic companies.

LogisticCompanyUserServiceImpl

Overview

This class implements UserDetailsService to handle user authentication and authorization within the logistic system. It loads user details from the database and converts them into Spring Security's UserDetails format.

Dependencies

- UserRepository: Fetches user information from the database.
- UserDetailsService: Spring Security interface for loading user-specific data.

Methods

- **loadUserByUsername(String username)**
    - Fetches a UserEntity from the database using UserRepository.
    - Throws UsernameNotFoundException if the user is not found.
    - Calls mapToUserDetails(UserEntity user) to convert the entity into a UserDetails object.
- **mapToUserDetails(UserEntity user)** *(private method)*
    - Maps UserEntity to a Spring Security UserDetails object.
    - Converts user roles into GrantedAuthority objects with the format ROLE_ROLE_NAME.
    - Returns a User object with username, password, and authorities.

Usage

This implementation is used by Spring Security to authenticate users based on their username and roles.

OfficeServiceImpl

Overview

This class implements the OfficeService interface and provides business logic for managing office-related operations within the logistic system. It interacts with repositories to perform CRUD operations on offices and initializes default office data.

Dependencies

- OfficeRepository: Handles database operations related to Office entities.
- UserRepository: Retrieves user entities associated with offices.
- PackageRepository: Fetches packages assigned to an office.

- LogisticCompanyRepository: Manages LogisticCompany entities.
- ModelMapper: Converts between Office entities and OfficeDto objects.
- Logger: Logs important system events.

Methods

- **createOffice(OfficeDto officeDto)**
  - Converts an OfficeDto to an Office entity.
  - Saves the entity in the database.
  - Logs the creation event.
- **getOffices()**
  - Retrieves all offices from the database.
  - Converts each Office entity to an OfficeDto.
  - Returns a list of DTOs.
- **updateOffice(OfficeDto officeDto, Long id)**
  - Finds an office by its ID, throwing an ObjectNotFoundException if it does not exist.
  - Maps the new values from the DTO to the existing office entity.
  - Saves the updated office to the database.
  - Logs the update event.
- **deleteOffice(Long id)**
  - Deletes the office entity by ID.
  - Logs the deletion event.
- **initializeOffices()**
  - Fetches predefined logistic companies, packages, and users from the repository.
  - Creates a new Office entity with associated employees and packages.
  - Saves the office and updates the logistic company.

Usage

This service is responsible for managing office-related business logic and database interactions, including creating, updating, deleting, and initializing office data.

PackageServiceImpl Class

Overview

This class implements the PackageService interface, providing business logic for package-related operations. It interacts with repositories to manage packages, including creation, retrieval, updates, and deletion. Additionally, it includes functions for tracking package statuses, calculating costs, and initializing package data.

Dependencies

- PackageRepository: Handles database operations for Package entities.

- UserRepository: Retrieves users related to a package (sender, receiver, courier).
- OfficeRepository: Retrieves office locations for price calculation.
- ModelMapper: Converts between Package entities and DTOs.
- Logger: Logs important actions in package management.

Constants

- ADDRESS_TAX = 1.15: Applied when the destination is outside registered offices.
- WEIGHT_OVER_20_TAX = 1.05: Applied to packages heavier than 20 kg.

Methods

- **createPackage(PackageServiceModel packageServiceModel, String userName)**
    - Retrieves the sender user by userName.
    - Maps the PackageServiceModel to a Package entity.
    - Sets default values (price, dates, statuses).
    - Saves the package and logs creation.
- **getPackages()**
    - Retrieves all packages.
    - Converts them to PackageDto and returns the list.
- **updatePackage(PackageDto packageDto, Long id)**
    - Finds a package by id.
    - Updates package details using ModelMapper.
    - Saves the updated package and logs the update.
- **deletePackage(Long id)**
    - Deletes a package by ID and logs the deletion.
- **getAllRegisteredPackages()**
    - Retrieves all ACCEPTED packages.
- **getAllRegisteredPackagesByReceiver(Long receiverId)**
    - Finds a receiver by receiverId.
    - Checks if the receiver is an OFFICE_EMPLOYEE.
    - Retrieves all NOT_DELIVERED and ACCEPTED packages for the receiver.
- **getAllRegisteredNotDeliveredPackages()**
    - Retrieves all NOT_DELIVERED packages of type ACCEPTED.
- **getAllSentNotDeliveredPackages()**
    - Retrieves all NOT_DELIVERED packages of type SENT.
- **getAllPackagesSentByClient(Long clientId)**
    - Retrieves all SENT packages by a client.
- **getAllPackagesReceivedByClient(Long clientId)**
    - Retrieves all ACCEPTED packages received by a client.
- **calculatePrice(Package pack)**
    - Applies an extra tax if the package is heavier than 20 kg.
    - Applies an address tax if the destination is outside known office addresses.
    - Rounds the price up.
- **acceptPackage(Long packageId, String employeeUsername)**
    - Finds a package by packageId.
    - Finds an employee by employeeUsername.
    - Sets the employee as the receiver and marks the package as ACCEPTED.
- **initializePackages()**

    - Initializes a few test packages with predefined users and addresses.
    - Saves them to the database.
    - Prints various package-related reports.
- **findAllClientPackagesDetails(String username)**
    - Retrieves all packages sent by a user.
    - Maps them to ClientPackageDetailsView, including sender, receiver, courier, address, weight, price, and status details.

Key Features

- Uses ModelMapper for efficient object transformation.
- Provides methods for both administrative (CRUD operations) and client-side package tracking.
- Implements tax calculations based on package weight and destination.
- Ensures employees have the correct role when handling package operations.
- Logs important actions for better traceability.

This class is central to package management in the logistic system, handling everything from pricing to delivery status updates.

RoleServiceImpl Class

Overview

This class implements the RoleService interface, managing the initialization of user roles within the system. It interacts with the RoleRepository to store role-related data.

Dependencies

- RoleRepository: Handles database operations for RoleEntity objects.

Methods

- **initializeRoles()**
    - Creates predefined roles (ADMIN, CLIENT, OFFICE_EMPLOYEE, COURIER_EMPLOYEE).
    - Saves them to the database using roleRepository.saveAll().

UserServiceImpl Class

Overview

This class implements the UserService interface, providing user management functionalities such as creation, retrieval, updating, authentication, and balance management. It interacts with UserRepository, RoleRepository, and PackageRepository for user-related operations.

Dependencies

- UserRepository: Handles user-related database operations.
- RoleRepository: Manages user roles.
- PackageRepository: Retrieves packages associated with users.
- ModelMapper: Converts between UserEntity and DTOs.
- PasswordEncoder: Encodes passwords before saving.
- LogisticCompanyUserServiceImpl: Loads user details for authentication.

Methods

- **createUser(UserDto userDto)**
  - Maps UserDto to UserEntity.
  - Saves the user to the repository.
  - Logs user creation.
- **getUsers()**
  - Retrieves all users.
  - Converts them to UserDto.
  - Returns the list.
- **updateUser(UserDto userDto, Long id)**
  - Finds a user by ID.
  - Updates user details.
  - Saves the updated user.
  - Logs the update.
- **deleteUser(Long id)**
  - Deletes a user by ID.
  - Logs the deletion.
- **getAllEmployees()**
  - Retrieves all employees (OFFICE_EMPLOYEE, COURIER_EMPLOYEE).
  - Returns them as a list of UserDto.
- **getAllClients()**
  - Retrieves all users with the CLIENT role.
  - Returns them as a list of UserDto.
- **pay(String username, Long packageId)**
  - Retrieves user by username and package by packageId.
  - Checks if the package is NOT_DELIVERED and user has sufficient balance.
  - Deducts balance and updates package status to PAID.
  - Saves updated user and package.
- **isUserExistingByEmailOrUsername(String email, String username)**
  - Checks if a user exists with the given email or username.
- **registerUser(UserServiceModel userServiceModel)**
  - Assigns a default CLIENT role.
  - Encodes the password before saving.
  - Saves the user and authenticates them.
- **initializeUsers()**
  - Initializes predefined users with different roles (ADMIN, CLIENT, OFFICE_EMPLOYEE, COURIER_EMPLOYEE).
  - Saves them to the database.

    - Prints employees and clients.
- **getLoggedUserInfo(String name)**
    - Retrieves user balance details based on the logged-in username.
- **getUserInfo(String username)**
    - Retrieves user details by username and maps to UserViewModel.
- **addMoneyToUser(String username)**
    - Adds a fixed amount (1000) to the user's balance.
    - Saves the updated user.

Key Features

- Implements CRUD operations for user management.
- Handles authentication and balance transactions.
- Uses ModelMapper for efficient object conversion.
- Ensures proper role assignment during registration.
- Logs important actions for better traceability.

This class plays a crucial role in managing user operations within the logistic system, ensuring secure authentication, role assignments, and financial transactions.

6. Web

a. Handler

GlobalExceptionHandler

Description:

The GlobalExceptionHandler class is responsible for handling exceptions globally across the application. It ensures that specific exceptions are caught and appropriate responses are returned to the user, improving the error-handling process.

Annotations:

- @ControllerAdvice – Marks this class as a global exception handler that applies to all controllers.
- @ExceptionHandler(ObjectNotFoundException.class) – Specifies that this method handles ObjectNotFoundException.

Methods:

- **handleException(ObjectNotFoundException e) : ModelAndView**
    - **Description:** Handles the ObjectNotFoundException by returning a custom error page.
    - **Parameters:**
        - e – The exception instance containing the error details.
    - **Returns:**

        - ModelAndView – A view object containing the error page (object-not-found) and the HTTP status (404 Not Found).
    - **Functionality:**
        - Creates a ModelAndView instance with the object-not-found view.
        - Sets the HTTP response status to 404 Not Found.
        - Adds an error message retrieved from the exception.
        - Returns the ModelAndView object to be displayed to the user.

Example Usage:
If a requested object (e.g., user, package) is not found in the database, an ObjectNotFoundException is thrown. The GlobalExceptionHandler catches this exception and returns a user-friendly error page with the corresponding message.

HomeController

Description:
The HomeController handles the routing for the home page of the application.

Annotations:

- @Controller – Marks this class as a Spring MVC controller.

Methods:

- **index() : String**
    - **Description:** Maps the root URL ("/") to the index.html view.
    - **Returns:** "index" – The name of the view to be displayed.
- **home() : String**
    - **Description:** Maps the /home endpoint to the home.html view.
    - **Returns:** "home" – The name of the view to be displayed.

PackageController

Description:
The PackageController manages package-related functionalities, such as viewing, purchasing, and adding packages.

Annotations:

- @Controller – Marks this class as a Spring MVC controller.
- @RequestMapping("/packages") – Maps all endpoints in this controller under /packages.

Dependencies:

- PackageService – Handles package-related business logic.
- UserService – Manages user-related operations.
- ModelMapper – Used for converting between DTOs and entity models.

Methods:

- **myPackages(Model model, Principal principal) : String**
  - **Description:** Displays all packages associated with the logged-in user.
  - **Parameters:**
    - model – The model object used to pass attributes to the view.
    - principal – Contains details about the currently authenticated user.
  - **Returns:** "my-packages" – The name of the view.
- **payPackageConfirm(Long id, RedirectAttributes redirectAttributes, Principal principal) : String**
  - **Description:** Handles package purchase logic.
  - **Parameters:**
    - id – The ID of the package to be purchased.
    - redirectAttributes – Stores flash attributes for redirection.
    - principal – Represents the currently authenticated user.
  - **Returns:** Redirects to "my-packages" if payment fails, otherwise redirects to "/".
- **add() : String**
  - **Description:** Displays the package creation page.
  - **Returns:** "add-package" – The name of the view.
- **addConfirm(PackageAddBindingModel packageAddBindingModel, BindingResult bindingResult, RedirectAttributes redirectAttributes, Principal principal) : String**
  - **Description:** Handles the addition of new packages.
  - **Parameters:**
    - packageAddBindingModel – Contains package details submitted by the user.
    - bindingResult – Stores validation results.
    - redirectAttributes – Stores flash attributes for redirection.
    - principal – Represents the currently authenticated user.
  - **Returns:** Redirects to "my-packages" if successful, otherwise redirects to "add-package".
- **packageAddBindingModel() : PackageAddBindingModel**
  - **Description:** Provides a default PackageAddBindingModel object for binding in forms.
  - **Returns:** A new instance of PackageAddBindingModel.

UserController

Description:

The UserController handles user authentication, registration, and profile management.

Annotations:

- @Controller – Marks this class as a Spring MVC controller.
- @RequestMapping("/users") – Maps all endpoints in this controller under /users.

Dependencies:

- UserServiceImpl – Handles user-related business logic.
- ModelMapper – Used for mapping DTOs and entity models.

Methods:

- **login() : String**
  - **Description:** Displays the login page.
  - **Returns:** "login" – The name of the login view.
- **loginFailed(String username, RedirectAttributes redirectAttributes) : String**
  - **Description:** Handles failed login attempts.
  - **Parameters:**
    - username – The username entered by the user.
    - redirectAttributes – Stores flash attributes for redirection.
  - **Returns:** Redirects to "login" with error messages.
- **register(Model model) : String**
  - **Description:** Displays the registration page.
  - **Parameters:**
    - model – Used to pass attributes to the view.
  - **Returns:** "register" – The name of the view.
- **registerConfirm(UserRegisterBindingModel userRegisterBindingModel, BindingResult bindingResult, RedirectAttributes redirectAttributes) : String**
  - **Description:** Handles user registration logic.
  - **Parameters:**
    - userRegisterBindingModel – Contains user registration details.
    - bindingResult – Stores validation results.
    - redirectAttributes – Stores flash attributes for redirection.
  - **Returns:** Redirects to "register" if validation fails, otherwise redirects to "/".
- **profile(Model model, Principal principal) : String**
  - **Description:** Displays the user profile page.
  - **Parameters:**
    - model – Used to pass attributes to the view.
    - principal – Contains details about the currently authenticated user.
  - **Returns:** "profile" – The name of the profile view.
- **addMoney(Principal principal) : String**
  - **Description:** Adds money to the user's account.
  - **Parameters:**
    - principal – Represents the currently authenticated user.
  - **Returns:** Redirects to "profile" after adding money.
- **userRegisterBindingModel() : UserRegisterBindingModel**
  - **Description:** Provides a default UserRegisterBindingModel object for binding in forms.
  - **Returns:** A new instance of UserRegisterBindingModel.

UserInfoRestController

Description:

The UserInfoRestController is a RESTful controller responsible for handling API requests related to user information. It provides an endpoint to retrieve the logged-in user's balance details.

Annotations:

- @RestController – Marks this class as a Spring MVC REST controller, meaning all methods return data directly instead of rendering views.

Dependencies:

- UserService – Handles user-related business logic.

Methods:

- **getLoggedUserInfo(Principal principal) : ResponseEntity<UserBalanceViewModel>**
    - **Description:** Retrieves the balance information of the currently authenticated user.
    - **Endpoint:** GET /api/user/info
    - **Parameters:**
        - principal – Represents the currently authenticated user.
    - **Returns:**
        - 200 OK – If the request is successful, returns the user's balance details wrapped in a UserBalanceViewModel object.

LogisticCompanyApplication

Description:

The LogisticCompanyApplication class is the main entry point of the application. It is responsible for bootstrapping and starting the Spring Boot application.

Annotations:

- @SpringBootApplication – A convenience annotation that combines:
    - @Configuration (indicating this class contains Spring configurations)
    - @EnableAutoConfiguration (enabling automatic configuration)
    - @ComponentScan (scanning for Spring components in the package and sub-packages)

Methods:

- **main(String[] args)**
    - **Description:** The main method that starts the Spring Boot application.

- **Functionality:**
  - Calls SpringApplication.run(LogisticCompanyApplication.class, args), which launches the embedded server and initializes the Spring context.

7. Tests

LogisticCompanyServiceTests

Description:

This test class verifies the functionality of the LogisticCompanyServiceImpl service, ensuring correct behavior when managing logistic companies.

Annotations:

- @ExtendWith(MockitoExtension.class): Enables Mockito-based testing in JUnit 5.
- @Mock: Marks dependencies that will be mocked using Mockito.
- @InjectMocks: Injects mocked dependencies into the class under test.
- @BeforeEach: Initializes test data before each test.
- @Test: Marks a method as a test case.

Test Cases:

1. **createCompany_ShouldSaveAndReturnCompany()**
   - Tests that a company is correctly mapped, saved, and returned.
   - Uses Mockito's when and verify to ensure correct interactions.
2. **getCompanies_ShouldReturnListOfCompanies()**
   - Tests retrieval of a list of logistic companies.
   - Ensures correct mapping of entities to DTOs.
3. **updateCompany_ShouldUpdateAndReturnCompany()**
   - Verifies that an existing company is updated correctly.
4. **updateCompany_ShouldThrowException_WhenCompanyNotFound()**
   - Ensures that updating a non-existent company throws ObjectNotFoundException.
5. **deleteCompany_ShouldDeleteCompany()**
   - Confirms that a company is successfully deleted by its ID.
6. **getRevenueForTimePeriod_ShouldReturnRevenue()**
   - Ensures that revenue is correctly calculated for a given time period.
7. **getRevenueForTimePeriod_ShouldReturnNull_WhenCompanyNotFound()**
   - Verifies that revenue calculation returns null if the company does not exist.

LogisticCompanyUserServiceTests

Test Cases:

1. **testLoadUserByUsername_UserExists**
   - Ensures correct user details are loaded when user exists.
   - Mocks userRepository.findByUsername().
2. **testLoadUserByUsername_UserNotFound**
   - Ensures an exception is thrown when the user is not found.
   - Uses assertThrows for UsernameNotFoundException.

OfficeServiceTests

Test Cases:

1. **createOffice_ShouldSaveAndReturnOffice**
   - Ensures office creation saves correctly.
   - Uses modelMapper to map DTO to entity.
   - Verifies repository method calls.
2. **getOffices_ShouldReturnListOfOffices**
   - Fetches all offices and ensures correctness.
   - Uses modelMapper to map entities to DTOs.
   - Verifies repository method calls.
3. **updateOffice_ShouldUpdateAndReturnOffice**
   - Ensures an existing office can be updated.
   - Mocks repository lookup and save operations.
   - Verifies correct field persistence.
4. **updateOffice_ShouldThrowExceptionWhenOfficeNotFound**
   - Ensures an exception is thrown when an office update is attempted on a non-existent office.
5. **deleteOffice_ShouldDeleteOffice**
   - Ensures office deletion works correctly.
   - Uses verify to check repository method invocation.

PackageServiceTests

Test Cases:

1. **testCreatePackage_UserNotFound**

- Ensures an exception is thrown when creating a package for a non-existent user.
- Uses assertThrows for ObjectNotFoundException.

2. **testUpdatePackage**
   - Ensures packages can be updated successfully.
   - Mocks repository lookup and save operations.
   - Verifies correct field persistence.
3. **testUpdatePackage_PackageNotFound**
   - Ensures an exception is thrown when trying to update a non-existent package.
4. **testDeletePackage**
   - Ensures package deletion works correctly.
   - Uses verify to check repository method invocation.
5. **testGetPackages**
   - Fetches all packages and ensures correctness.
   - Uses modelMapper to map entities to DTOs.
6. **testCalculatePrice_WeightOver20**
   - Ensures correct price calculation when weight exceeds 20kg.
   - Applies additional charges if necessary.
7. **testCalculatePrice_AddressNotInOffices**
   - Ensures extra cost is applied when the package address is outside office locations.
8. **testGetAllRegisteredPackages**
   - Fetches all registered packages of type ACCEPTED.
   - Uses modelMapper to map entities to DTOs.

UserServiceTests

Test Cases:

1. **createUser_ShouldSaveAndReturnUser**
   - Ensures that creating a user correctly saves it in the repository.
   - Uses modelMapper to map DTO to entity.
   - Verifies that userRepository.save() is called once.
2. **getUsers_ShouldReturnListOfUsers**
   - Ensures fetching users returns a correct list.
   - Uses modelMapper to map entities to DTOs.
   - Verifies repository method calls.
3. **updateUser_ShouldUpdateAndReturnUser**
   - Ensures an existing user can be updated.
   - Mocks repository lookup and save operations.
   - Verifies correct field persistence.
4. **deleteUser_ShouldDeleteUser**
   - Ensures deletion of a user from the repository.

    - Uses verify to check repository method invocation.
5. **payPackage_ShouldSucceed_WhenSufficientBalance**
    - Ensures a user can successfully pay for a package if balance is sufficient.
    - Mocks repository calls for user and package retrieval.
    - Verifies updated balance and package payment status.
6. **payPackage_ShouldFail_WhenInsufficientBalance**
    - Ensures payment fails when user has insufficient balance.
    - Verifies that package payment status remains unchanged.
7. **isUserExistingByEmailOrUsername_ShouldReturnTrue_WhenUserExists**
    - Ensures the method returns true when a user exists by either email or username.
    - Mocks repository lookups.
8. **isUserExistingByEmailOrUsername_ShouldReturnFalse_WhenUserDoesNotExist**
    - Ensures the method returns false when neither email nor username exists in the repository.
    - Mocks repository lookups.
9. **addMoneyToUser_ShouldIncreaseBalance**
    - Ensures adding money to a user correctly updates the balance.
    - Mocks repository lookups and save operations.