1. Project Overview

Link to frontend GitHub
Link to backend GitHub
Link to Trello

1.1. Description

This project is an online food ordering system application where users can browse restaurants, select products from a selected restaurant to add to the shopping cart and order the products to be delivered to a specified location. The application supports owner users who have different tools for restaurant management.

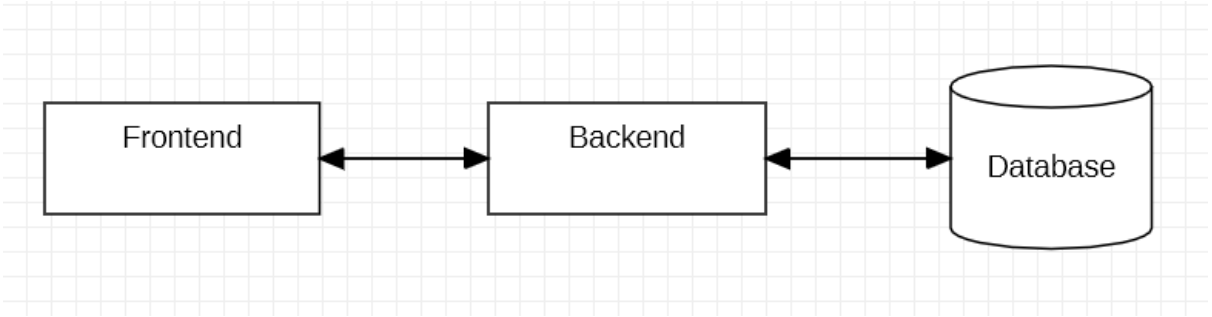
1.2. Goals and scope

We chose the project with the intention to create our own version of a food delivery application to see if there were improvements to make.

For scope we wanted to include both regular users and owner users. For regular users this project includes user registration, editable user settings and food ordering. For owner users the project includes editing an owned restaurant's general info as well as the menu of the restaurant.

2. System Architecture

- **Architecture Type**: Client-Server architecture with RESTful API
- **Components**:
  - Frontend: JafaFX for the user interface.
  - Backend: Java with Springboot for server-side logic.
  - Database: MariaDB for data storage.
- **Authentication**: JWT for authentication



3. Technologies used

- **Frontend:** Java, JavaFX
- **Backend:** Java, Spring Boot
- **Database:** MariaDB
- **Authentication:** JWT
- **Version Control:** GitHub
- **CI/CD:** Jenkins
- **Virtual Environment:** Docker, VM
- **Testing**: Junit , Postman
- **Dependency Management:** Maven

4. Software Design

Our application implements the MVC-model (Model, view, controller). We decided to create separate backend and frontend projects in order to maintain good performance and easy readability.

4.1. Major classes

Backend:
Controller classes (ProductController, RestaurantController etc) control the flow and functionality of our application.

Entity classes (ContactInfoEntity, OrderEntity etc.) are used to store and provide information about specific entities.

DTO classes (UserDTO, LoginRequestDTO etc) have been created to encapsulate data and send it from a subsystem of the application to another.

Frontend:
Controller classes (AdminMenuController, NavController etc.) are assigned to control specific FXML files. They provide necessary functionality into the frontend side of our application.

Model classes (Product, Translation etc.) are used to store data about that specific model. They mainly consist of getter and setter methods but are still a crucial part of the software.

Service classes (ProductService, UserService etc.) interact with our API to create, read, update and delete information in the application.

4.2. Major functions

Backend:
authenticateUser function (AuthController, full method not visible in screenshot)

```
mikktur +1
@PostMapping("/login")
public ResponseEntity<LoginResponseDTO> authenticateUser(@RequestBody LoginRequestDTO loginRequest) {
    try {
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        loginRequest.getUsername(),
                        loginRequest.getPassword()
                )
        );

        MyUserDetails userDetails = (MyUserDetails) authentication.getPrincipal();
        String role = userDetails.getAuthorities().iterator().next().getAuthority().replace( target: "ROLE_",  replacement: "");
        String token = jwtUtil.generateToken(userDetails.getUsername(), role);


        LoginResponseDTO response = new LoginResponseDTO(
                userDetails.getUserId(),
                success: true,
                userDetails.getUsername(),
                message: "Authentication successful",
                token,
                role
        );

        return new ResponseEntity<>(response, HttpStatus.OK);
```

deleteProductFromRestaurant (ProductService)

```java
1 usage  emotytto00 +1
@Transactional
public ResponseEntity<String> deleteProductFromRestaurant(int productId, int restaurantId, String owner) {
    RestaurantEntity restaurant = restaurantRepository.findByRestaurantID(restaurantId)
            .orElseThrow(() -> new RuntimeException("Restaurant not found"));

    if (!restaurant.getOwner().getUsername().equals(owner)) {
        return new ResponseEntity<>( body: "You are not authorized to delete products from this restaurant.", HttpStatus.UNAUTHORIZED);
    }

    ProvidesEntity providesEntity = providesRepository.findByProductIdAndRestaurantId(productId, restaurantId);

    if (providesEntity == null) {
        return new ResponseEntity<>( body: "Product not found in this restaurant.", HttpStatus.NOT_FOUND);
    }

    providesRepository.delete(providesEntity);

    return new ResponseEntity<>( body: "Product successfully deleted from the restaurant.", HttpStatus.OK);
}
```

createUser (UserController)

```java
mikktur
@PostMapping(🌐v"/create")
public ResponseEntity<CreateUserResponseDTO> createUser(@Valid @RequestBody CreateUserRequestDTO createUserRequest) {
    try {

        UserEntity createdUser = userService.createUser(createUserRequest);

        System.out.println("User created: " + createdUser.getUsername());
        CreateUserResponseDTO response = new CreateUserResponseDTO(
                createdUser.getUserId(),
                createdUser.getUsername(),
                 message: "User created successfully",
                 success: true
        );

        return new ResponseEntity<>(response, HttpStatus.CREATED);
    } catch (Exception e) {
        e.printStackTrace();
        CreateUserResponseDTO response = new CreateUserResponseDTO(
                 id: null,
                 username: null,
                 message: "User creation failed: " + e.getMessage(),
                 success: false
        );

        return new ResponseEntity<>(response, HttpStatus.BAD_REQUEST);
    }
}
```

Frontend:

userLogin (LoginController)

```java
@FXML
public void userLogin(ActionEvent event) {
    new Thread(() -> {
        try {
            Response response = userService.login(username.getText(), password.getText());
            Platform.runLater(() -> handleLoginResponse(response));
        } catch (IOException e) {
            Platform.runLater(() -> {
                System.out.println("Login failed.");
                e.printStackTrace();
                showError("Login error: " + e.getMessage());
            });
        }
    }).start();
}
```

addProductToRestaurant (ProductService)

```
1 usage  emotytto00
public void addProductToRestaurant(Product product, int restaurantId) throws IOException {
    MediaType JSON = MediaType.get("application/json; charset=utf-8");

    String json = mapper.writeValueAsString(product);

    SessionManager sessionManager = SessionManager.getInstance();
    String bearerToken = sessionManager.getToken();

    RequestBody body = RequestBody.create(json, JSON);
    Request request = new Request.Builder()
            .url(API_URL + "api/products/create/" + restaurantId)
            .post(body)
            .addHeader( name: "Authorization",  value: "Bearer " + bearerToken)
            .build();

    Response response = client.newCall(request).execute();

    if (!response.isSuccessful()) {
        throw new IOException("Failed to add product: " + response);
    }
}
```

setLocale (LocalizationManager)

```
3 usages  DarkpunkKI +1
public static void setLocale(Locale newLocale) {
    locale = newLocale;
    bundle = ResourceBundle.getBundle(BASE_NAME, locale);

    selectedLanguage.set(languageMap.getOrDefault(locale.getLanguage(),  defaultValue: "English"));
}
```

addRestaurantCard (MainMenuController)

```
6 usages  mikktur +1
private void addRestaurantCard(Restaurant restaurant) {
    try {
        FXMLLoader loader = new FXMLLoader(getClass().getResource( name: "/ofosFrontend/User/restaurant_card.fxml"));
        VBox card = loader.load();

        ImageView imageView = (ImageView) card.lookup( s: "#restaurantImage");
        Label descriptionLabel = (Label) card.lookup( s: "#restaurantDesc");

        imageView.setImage(new Image( s: URL + restaurant.getPicture()));
        descriptionLabel.setText(restaurant.getRestaurantName() + "\n" + restaurant.getRestaurantPhone());

        card.setOnMouseClicked(event -> {
            try {
                goToRestaurant(restaurant);
            } catch (IOException e) {
                e.printStackTrace();
            }
        });

        restaurantFlowPane.getChildren().add(card);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

4.3. Product management

Products are stored in our database into their own table. Restaurant owners can create products in the application while logged in on an owner account. Products can also be directly edited in the application and those changes are directly updated to the database. This way we can let the restaurant owner manage their own products by themselves and further ensure customer satisfaction.

4.4. Restaurant management

Restaurants are stored in our database. The restaurant table has fields for all the necessary information about that restaurant. Restaurant information can be updated directly in the application (while logged in on an owner account), but at this time we don't provide functionality to add new restaurants. We decided it's better this way so we can minimize bad actors on our application.

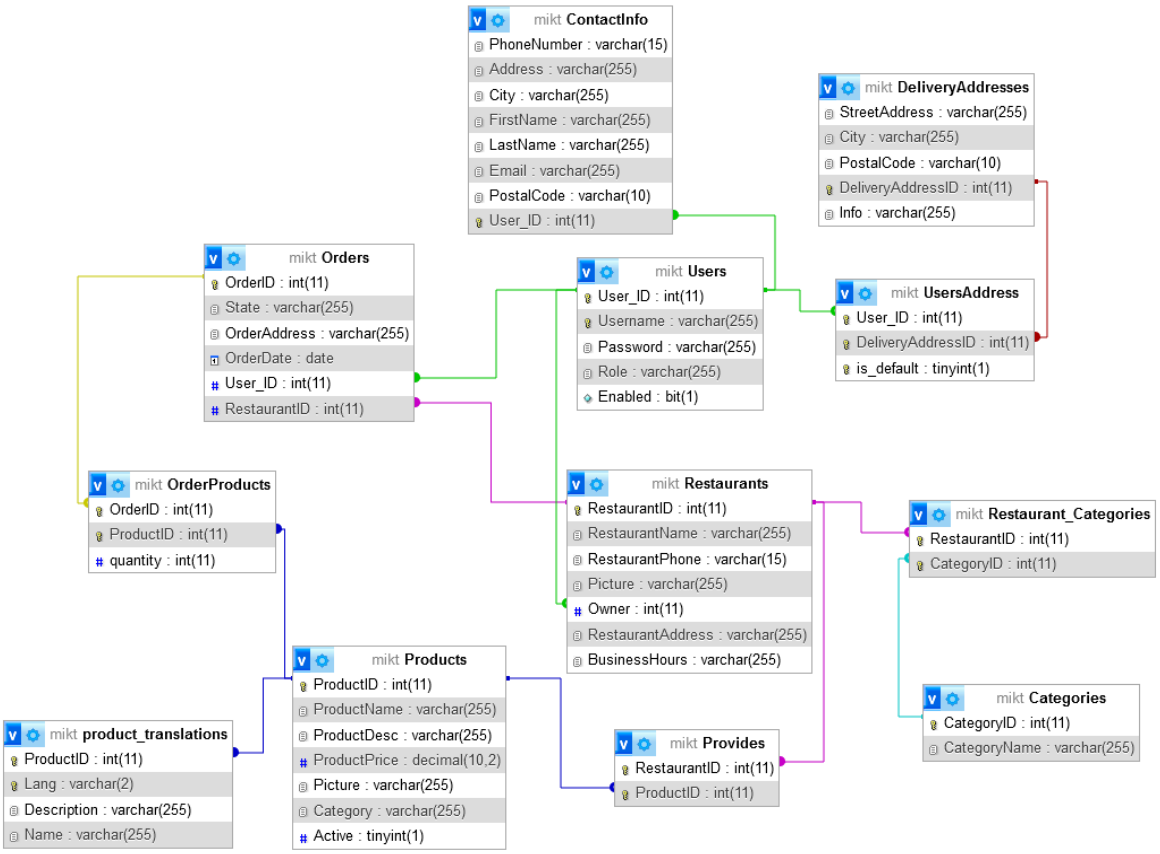
4.5. Internationalization and localization

OFOS application comes in four different languages. English (which is the default language), Finnish, Japanese and Russian. Our development team practiced Internationalization (i18n) to ensure translated text displays properly in the application. After internationalization we started to implement localization to our application. We decided to provide restaurant owners the possibility to translate products into our chosen languages. Those translations are stored in our database (product_translation table). All other translations are stored in our Resource bundle, since they are static.

4.6. FXML

OFOS application is designed with MVC-model, but we don't use view classes. Instead, our views are created with FXML files. Our design team started by designing the application in Figma, and based on that we created our individual FXML files. Each different view has its own FXML with the exception of navBar and adminNavBar. These two are loaded in the individual FXML files to streamline development and eliminate duplication.

5. Database Schema



- **Users**
  - User_ID (Int, primary key)
    - Auto generated user
  - Username (String)
  - Password (String)
  - Role (String)
  - Enabled (Boolean, true if user not banned)

- **UsersAddress**
  - User_ID (Int, references to User table, primary key)
  - DeliveryAddressID (Int, references DeliveryAddress table, primary key)
  - is_default (Boolean, true if user's default address)

- **DeliveryAddresses**
  - DeliveryAddressID (Int, primary key)
    - Auto generated
  - StreetAddress (String)

    - City (String)
    - PostalCode (String)
    - Info (String, additional information for delivery purposes)

- **ContactInfo**
    - User_ID (Int, references Users, primary key)
    - Firstname (String)
    - Lastname (String)
    - Address (String)
    - PostalCode (String)
    - PhoneNumber (String)
    - Email (String)

- **Orders** (log of user's order history)
    - OrderID (Int, primary key)
    - State (String, status of the order, e.g. "received", "preparing")
    - OrderAddress (String)
    - OrderDate (Date)
    - User_ID (Int, references Users)
    - RestaurantID (Int, references Restaurants)

- **Restaurants**
    - RestaurantID (Int, primary key)
        - Auto generated
    - RestaurantName (String)
    - RestaurantPhone (String)
    - Picture (String, URL to restaurant's logo image)
    - Owner (Int, references User_ID in Users)
    - RestaurantAddress (String)
    - BusinessHours (String, shows when the restaurant is open)

- **Provides**
    - RestaurantID (Int, references Restaurants, primary key)
    - ProductID (Int, references Products, primary key)

- **Products**
    - ProductID (Int, primary key)
        - Auto generated
    - ProductName (String)
    - ProductDesc (String, description for the product)
    - ProductPrice (Decimal)
    - Picture (String, URL to image of product)
    - Category (String)
    - Active (Boolean, true if product is actively offered at restaurant)

- **OrderProducts** (logs the products for each order)
    - OrderID (Int, references Orders, primary key)
    - ProductID (Int, references Products, primary key)

        - Quantity (Int)

- **Product_translations** (translation table for products)
    - ProductID (Int, references Products, primary key)
    - Lang (String, two letter ISO 639 language code, primary key)
    - Description (String)
    - Name (String)

- **Restaurant_Categories**
    - RestaurantID (Int, references Restaurants, primary key)
    - CategoryID (Int, references Categories, primary key)

- **Categories** (e.g. Burger, Pizza)
    - CategoryID (Int, primary key)
        - Auto generated
    - CategoryName (String)

6. API Documentation

6.1. Introduction

6.1.1. The OFOS API allows users to interact with the system for managing restaurants, menus, orders, and users.

6.2. Authentication

6.2.1. Most API endpoints require a valid JWT token passed in the Authorization header in the format Bearer <token>."

6.3. Endpoint Overview

Endpoint	Method	Description	Headers	Request Body	Response
/api /auth /login	POST	Authenticates a user by validating their username and password. If successful, the endpoint returns a JWT token along with user details for further authenticated requests.	“Content-type: “application/json”	{ “Username”: “String”, “Password”: “String” }	**Success (200 OK):** { “userId”: 1, “success” : true, “Username”: “Jimi”, “Message”: “Authentication successful”, “Token”: “Jwt-token-string”, “Role”: “Owner” } **Error (401 Unauthorized):** { “userId”: null, “Success”: false, “Message”: “Incorrect username or password.” }

/api
/Contactinfo
/{userID}
	GET	Fetches contact information for the user with the given ID.	None	None	**Success (200 OK):**
					{
					"userId": 8,
					"phoneNumber": "0401234567",
					"address": "Täällä asuu mikkee",
					"city": "Kouvola",
					"firstName": "Mikaeli",
					"lastName": "Testerinpoika",
					"email": "mikke@granlund.com",
					"postalCode": "02360"
					}
					
					Error (404 Not Found):
					No contact information found.

Endpoint	Method	Description	Headers	Request Body	Response

/api
/contactinfo
/update
	POST	Updates the contact information for the authenticated user.	“Authorization”: “Bearer <token>”	{
				“phoneNumber”: “123456789”,
				“Address”: “Kotikatu 4”,
				“City”: “Espoo”,
				“firstName”: “John”,
				“lastName”: “Doe”,
				“Email”: “john.doe@example.com”,
				“postalCode”: “12345”,
				“userId”: 1
				}
					Success (200 OK):
					“Contact information updated successfully.”
					
					Error (401 Unauthorized):
					Invalid or missing token.
					
					Error (400 Bad Request):
					Invalid contact information payload

/api
/contactinfo
/save
	POST	Saves new contact information for the authenticated user.	“Authorization”: “Bearer <token>”	{
				“phoneNumber”: “123456789”,
				“Address”: “Kotikatu 4”,
				“City”: “Espoo”,
				“firstName”: “John”,
				“lastName”: “Doe”,
				“Email”: “john.doe@example.com”,
				“postalCode”: “12345”,
				“userId”: 1
				}
					Success (200 OK):
					“Contact information saved successfully.”
					
					Error (401 Unauthorized):
					Invalid or missing token.
					
					Error (400 Bad Request):
					Invalid contact information payload.

/api
/deliveryaddress
/{id}
	GET	Retrieves all delivery addresses for the user with the given ID, including a default flag for the default address.	None	None	**Success (200 OK):**
					[
					{
					"streetAddress": "Katulankatu 3 B 81",
					"city": "Espoo",
					"postalCode": "02230",
					"deliveryAddressId": 122,
					"info": "Jee jee\n",
					"defaultAddress": true
					}
					]
					
					Error (404 Not Found):
					No address found.

Endpoint	Method	Description	Headers	Request Body	Response
/api /deliveryaddress/ save	POST	Saves a new delivery address for the authenticated user.	“Authorization”: “Bearer <token>”	{ “streetAddress”: “Kotikatu 3”, “City”: “Espoo”, “postalCode”: “12345”, “deliveryAddress Id”: 0, “Info”: “Ovikoodi on 1234”, “defaultAddress: true }	**Success (200 OK):** Delivery address saved successfully. **Error (401 Unauthorized):** Invalid or missing token. **Error (400 Bad Request):** Invalid delivery address data.
/api /deliveryaddress/ update	PUT	Updates an existing delivery address.	"Authorization": "Bearer <token>"	{ “streetAddress”: “Kotikatu 3”, “City”: “Espoo”, “postalCode”: “12345”, “deliveryAddress Id”: 0, “Info”: “Ovikoodi on 1234”, “defaultAddress: true }	**Success (200 OK):** Delivery address saved successfully. **Error (401 Unauthorized):** Invalid or missing token. **Error (400 Bad Request):** Invalid delivery address data.
/api /deliveryaddress/ delete /{addressID}	DELETE	Deletes the delivery address with the given ID for the authenticated user.	"Authorization": "Bearer <token>"	None	**Success (200 OK):** Delivery address deleted successfully. **Error (404 Not Found):** Address not found. **Error (401 Unauthorized):** Invalid or missing token.
/api /deliveryaddress/ setDefault	PUT	Sets a delivery address as the default for the authenticated user.	"Authorization": "Bearer <token>"	{ "deliveryAddress Id": 1, "userId": 123 }	**Success (200 OK):** Default delivery address set successfully. **Error (400 Bad Request):** Invalid address ID or user ID. **Error (401 Unauthorized):** Invalid or missing token. **Error (404 Not Found):** Address not found.

Endpoint	Method	Description	Headers	Request Body	Response
/images /{filename}	GET	Retrieves an image file by its filename from the products directory (/uploads/restaurants/Products/).	None	None	**Success (200 OK):** Returns the image file with appropriate headers: - Content-Disposition: inline; filename="filename.jpg" - Content-Type: image/jpeg. **Error (400 Bad Request):** Malformed URL. **404 Not Found:** File not found.
/images /restaurant /{filename}	GET	Retrieves a restaurant logo by its filename from the logos directory (/app/uploads/restaurants/Logos/).	None	None	**Success (200 OK):** Returns the image file with appropriate headers: - Content-Disposition: inline; filename="filename.jpg" - Content-Type: image/jpeg. **Error (400 Bad Request):** Malformed URL. **404 Not Found:** File not found.
/api/order	POST	Place a new order for the authenticated user.	"Authorization": "Bearer <token>"	{ "state": "string", "quantity": int, "productID": int, "deliveryAddressID": int, "restaurantID": int }	**Success (200 OK):** Order placed successfully. **Error (400 Bad Request):** "User not found" or "Order list is empty." **Error (404 Not Found):** Missing restaurant, delivery address, or product.
/api/order/{id}	GET	Retrieves all orders placed by the user with the given ID.	None	None	**Success (200 OK):** List of OrdersEntity objects. Example: [{ "orderId": 1, "restaurantName": "McDonalds", "orderDate": "2024-12-03", "state": "Completed" }] **Error (404 Not Found):** No orders found.

Endpoint	Method	Description	Headers	Request Body	Response
/api /order /products/{id}	GET	Retrieves the product details for all orders placed by the user with the given ID.	None	None	**Success (200 OK):** List of OrderProductsEntity objects. Example: [{ "productName": "Burger", "quantity": 2, "price": 5.99 }] **Error (404 Not Found):** No products found for the orders.
/api /order /{language} /history	GET	Retrieves the order history for the authenticated user in the specified language.	"Authorization": "Bearer <token>"	None	**Success (200 OK):** A map of order IDs to lists of OrderHistoryDTO. Example: { "1": [{ "productName": "Burger", "quantity": 2, "price": 5.99, "orderDate": "2024-12-03", "restaurantName": "McDonalds" }] } **Error (400 Bad Request):** Missing or invalid token.
/api /order /status /{id} /{status}	POST	Updates the status of the order with the given ID.	None	None	**Success (200 OK):** "Status updated to: {status}" **Error (400 Bad Request):** "Something went wrong."
/api /products /{language} /{id}	GET	Fetches a product by its ID and translates the product details to the specified language.	None	None	**Success (200 OK):** Returns ProductEntity object . Example: { "productID": 101, "productName": "Burger", "productDesc": "Delicious burger", "productPrice": 5.99, "category": "Food", "picture": "/images/burger.jpg", "active": true } **Error (404 Not Found):** Product not found.
/api /products /delete /{id}	DELETE	Deletes a product by its ID if the authenticated user is the owner.	"Authorization": "Bearer <token>"	None	**Success (200 OK):** "Product deleted." **Error (401 Unauthorized):** "Not an owner." **Error (404 Not Found):** Product not found.

Endpoint	Method	Description	Headers	Request Body	Response
/api /products /delete /{productId} /restaurant /{restaurantId}	DELETE	Deletes a product from a specific restaurant if the authenticated user is the owner.	"Authorization": "Bearer <token>"	None	**Success (200 OK):** "Product successfully deleted from the restaurant." **Error (401 Unauthorized):** "You are not authorized to delete products from this restaurant." **Error (404 Not Found):** "Product not found in this restaurant."
/api /products /create /{restaurantId}	POST	Creates a new product for a specified restaurant if the authenticated user is the owner.	"Authorization": "Bearer <token>"	{ "productName": "Burger", "productDesc": "Delicious burger", "productPrice": 5.99, "category": "Burger", "picture": "/images/burger.jpg", "active": true, "translations": [{ "languageCode": "en", "name": "Burger", "description": "Delicious burger" }] }	**Success (201 Created):** "Product created and associated with the restaurant successfully." **Error (401 Unauthorized):** "You are not authorized to create products for this restaurant." **Error (404 Not Found):** Restaurant not found.
/api /products /update /{rid}	PUT	Updates an existing product for the specified restaurant if the authenticated user is the owner.	"Authorization": "Bearer <token>"	{ "productID": 101, "productName": "Updated Burger", "productDesc": "Even more delicious burger", "productPrice": 6.99, "category": "Food", "picture": "/images/updated_burger.jpg", "active": true, "lang": "en", "translations": [{ "languageCode": "en", "name": "Burger" }] }	**Success (200 OK):** "Product updated." **Error (401 Unauthorized):** "You are not the owner of this restaurant." **Error (404 Not Found):** "The specified product does not belong to this restaurant."

Endpoint	Method	Description	Headers	Request Body	Response
/api /products /restaurant /{language} /{restaurant}	GET	Fetches all products for a specific restaurant and translates the product details to the specified language.	None	None	**Success (200 OK):** List of ProductDTO objects. Example: [{ "productID": 101, "productName": "Burger", "productDesc": "Delicious burger", "productPrice": 5.99, "category": "Burger", "picture": "/images/burger.jpg", "active": true }] **Error (404 Not Found):** No products found.
/restaurants /{restaurantId}	PUT	Updates the details of a specific restaurant using the provided data.	None	{ "restaurantName": "New Name", "restaurantPhone": "123456789", "picture": "/images/new.png", "businessHours": "9:00-18:00", "address": "123 Main St" }	**Success (200 OK):** Returns the updated restaurant details. Example: { "id": 1, "restaurantName": "New Name", "restaurantPhone": "123456789", "picture": "/images/new.png", "businessHours": "9:00-18:00", "address": "123 Main St" } **Error (404 Not Found):** Restaurant not found.
/restaurants	GET	Retrieves all restaurants.	None	None	**Success (200 OK):** List of RestaurantDTO objects. Example: [{ "id": 1, "restaurantName": "McDonalds", "restaurantPhone": "123456789", "picture": "/images/logo.png", "ownerUsername": "owner1", "address": "123 Main St", "businessHours": "9:00-21:00" }]

Endpoint	Method	Description	Headers	Request Body	Response
/restaurants /owner /{userId}	GET	Retrieves all restaurants owned by the specified user.	None	None	**Success (200 OK):** List of RestaurantDTO objects. Example: [{ "id": 1, "restaurantName": "McDonalds", "restaurantPhone": "123456789", "picture": "/images/logo.png", "ownerUsername": "owner1", "address": "123 Main St", "businessHours": "9:00-21:00" }]
/restaurants /category /{categoryName}	GET	Retrieves all restaurants belonging to a specific category.	None	None	**Success (200 OK):** List of RestaurantDTO objects. Example: [{ "id": 1, "restaurantName": "Burger King", "restaurantPhone": "123456789", "picture": "/images/logo.png", "address": "123 Main St", "businessHours": "9:00-21:00" }]
/restaurants /changeowner	PUT	Changes the owner of a specific restaurant to a new owner.	None	{ "newOwnerId": 2, "restaurantId": 1 }	**Success (200 OK):** "Owner updated successfully." **Error (404 Not Found):** User or restaurant not found. **Error (500 Internal Server Error):** "An error occurred."
/api/users	GET	Retrieves a list of all users.	None	None	**Success (200 OK):** List of UserDTO objects. Example: [{ "userId": 1, "username": "johndoe", "role": "USER", "enabled": true }]

Endpoint	Method	Description	Headers	Request Body	Response
/api /users /username /{username}	GET	Retrieves user details by username.	None	None	**Success (200 OK):** UserDTO object. Example: { "userId": 1, "username": "johndoe", "role": "USER", "enabled": true } **Error (404 Not Found):** "User not found."
/api /users /id /{id}	GET	Retrieves user details by user ID.	None	None	**Success (200 OK):** UserEntity object. Example: { "userId": 1, "username": "johndoe", "role": "USER", "enabled": true } **Error (404 Not Found):** User not found.
/api /users /create	POST	Creates a new user with the provided username and password.	None	{ "username": "johndoe", "password": "P@ssword123" }	**Success (201 Created):** { "user_id": 1, "username": "johndoe", "message": "User created successfully", "success": true } **Error (400 Bad Request):** { "user_id": null, "username": null, "message": "User creation failed: [error]", "success": false }
/api /users /updatePassword	PUT	Updates the authenticated user's password.	"Authorization": "Bearer <token>"	{ "oldPassword": "OldP@ssword123", "newPassword": "NewP@ssword123" }	**Success (200 OK):** "Password updated." **Error (401 Unauthorized):** "Old password doesn't match." **Error (404 Not Found):** "User not found."
/api /users /delete	DELETE	Deletes the authenticated user.	"Authorization": "Bearer <token>"	None	**Success (200 OK):** "User deleted." **Error (418 I'm a teapot):** "Owner accounts cannot be deleted." **Error (401 Unauthorized):** "Something went wrong."

Endpoint	Method	Description	Headers	Request Body	Response
/api /users /ban /{userId}	POST	Toggles the ban status of the user with the specified ID.	None	None	**Success (200 OK):** "User's ban status updated." **Error (400 Bad Request):** "Something went wrong."
/api /users /changerole	PUT	Changes the role of a user to the specified role.	None	{ "userId": 1, "role": "ADMIN" }	**Success (200 OK):** "User's role updated." **Error (404 Not Found):** "User not found."

7. Testing Strategy

Unit Testing

Done with JUnit and Mockito to ensure components function correctly in isolation

Integration Testing

Backend: Automated test were implemented with JUnit. Postman was used for thorough manual testing of endpoints.

Frontend: Integration testing was performed manually with carefully planned test cases to ensure components interact correctly with each other.

Acceptance Testing

All team members participated in executing acceptance tests based on a comprehensive test case plan to ensure the system meets functional and nonfunctional requirements.

Performance Testing

The Intellij Profiler was used to identify performance bottlenecks and also that no resources were left hanging.

8. Deployment Process

Step 1 : Push code to github
Step 2 : Deploy backend to virtual machine or host locally
Step 3 : Setup MariaDB for database hosting

Step 4 : Run sql script in mariadb

9. User Documentation

9.1. Installation

- Clone the frontend repository.
- Compile the project by running '*mvn clean package*' in CMD
- Run the application by running *'java -jar target/OFOS-Frontend-1.0.jar*' in CMD

9.2. Usage

- Upon opening the application the user can login or register to the service if they don't have an account.
- After logging in the user is on the main page of the application where they can filter restaurants by categories (e.g. Burgers, Pizza) or choose from a restaurant from all the restaurants.
- The user can open the burgermenu on the frontpage to browse their order history, edit user settings and logout.
- Settings that can be edited in the user settings tab:
    - Contact information
    - New delivery address can be added
    - Password can be changed
    - Account can be deleted
- After choosing a restaurant the chosen restaurant's products are presented and the user can add them to their shopping cart. The shopping cart can be accessed from the top right where all the added products can be found.
- By clicking the checkout button the user is redirected to the checkout page where the summary of the order can be seen, the user can select a delivery address, the user can select a payment method and confirm the order.
- If the user in an owner account the user's owned restaurant will be displayed upon login.
- The user can then modify the information of the restaurant or edit the restaurant's menu.
- In the edit menu tab the user can add a product, edit an existing product or delete an existing product.

10. Maintenance Plan

10.1. Overview

The purpose of this maintenance plan is to ensure the continued functionality, performance, and security of the food ordering application. This document

outlines the procedures and schedules for routine maintenance, emergency support, and system updates.

10.2. Maintenance Objectives

Ensure system uptime and availability at 99.9%.
Improve the performance of the application to reduce latency during peak times.
Maintain security protocols to protect user data, including sensitive information like payment details.
Regularly update and patch the software to keep it up to date with the latest industry standards.
Address and resolve customer support tickets in a timely manner.

10.3. Types of Maintenance

Routine Maintenance Tasks:

- **Database optimization:** Weekly database indexing and cleanup to maintain query speed and performance.
- **Server performance checks:** Monthly checks to ensure server resources (CPU, memory, bandwidth) are not overloaded.
- **Backup and restore procedures:** Daily automated backups of the database and application data with monthly testing of restore processes.
- **Security patching:** Monthly review and installation of security patches for both server-side and client-side components.
- **Performance testing:** Quarterly stress tests to ensure the system can handle high traffic, especially during events or peak hours (lunch/dinner times).
- **User interface (UI) testing:** Every 6 months, a review of the UI/UX to check for improvements or fixes for user-reported issues.

Maintenance Windows:

- **Scheduled downtime:** Routine maintenance will occur every Wednesday between 2:00 AM and 4:00 AM local time, when system usage is at its lowest.
- **Notifications:** Users will be notified 48 hours in advance via in-app alerts and email notifications.

Emergency Procedures:

- **Immediate response:** In case of critical failure or security breach, an incident response team will be activated.
- **Communication:** Users will be informed of the emergency maintenance via app alerts, emails, and social media updates.
- **Resolution time:** The goal is to resolve emergency incidents within 4 hours of identification. In extreme cases, an extended period may be required, but constant communication with users will be maintained.

Contingency Plan:

- A backup server will be available for emergency data recovery and load balancing.
- In case of a severe database crash, the latest backup (up to 24 hours old) will be restored.
- A temporary static web page will be displayed during extended downtimes, informing users of ongoing maintenance.

10.4. Software updates

Feature Updates

Release cycle: New features or major updates will be scheduled for every 3 months.

Testing: All feature updates will undergo rigorous testing in a staging environment to avoid introducing new bugs into the live system.

User feedback: Beta testing with select users will be implemented to gather feedback before full roll-out.

Bug Fixes

Minor bug fixes: Patches for non-critical bugs will be deployed on an ongoing basis, with a goal of releasing fixes every 2 weeks.

Critical bugs: Critical bugs that affect core functionality will be resolved within 48 hours of identification.

10.5. Monitoring and Performance

Monitoring Tools

Application performance monitoring (APM): Tools like New Relic or Datadog will be used to monitor real-time application performance and identify bottlenecks.

Error logs: Error logging services will continuously capture exceptions, server errors, and downtime incidents for quick diagnosis.

User activity tracking: Tools such as Google Analytics will be used to track user behavior, load times, and app usage patterns to optimize performance.

Key Performance Indicators (KPIs):

System uptime: Target uptime is 99.9%, with immediate corrective actions if it falls below 99.5%.

Page load time: Average page load time should be under 3 seconds. Any delay beyond this will be investigated.

Error rate: Critical errors (crashes, payment failures) should remain below 1% of transactions.

Customer support resolution time: All support tickets should be addressed within 24 hours, with a resolution time of 72 hours for non-critical issues.

10.6. Escalation Procedures

If an issue cannot be resolved by the first-level support team, it will be escalated as follows:

Level 1: Helpdesk support (customer service team)

Level 2: System administrator or development team (for technical issues)

Level 3: Senior management or third-party vendor involvement (for severe incidents)

10.7. Roles and Responsibilities

Maintenance Team

System Administrator: Manages server health, database, and security patches.

Development Team: Handles bug fixes, feature rollouts, and performance improvements.

Helpdesk Support: Handles user queries and issues, escalating technical problems when necessary.

Quality Assurance (QA): Conducts routine testing, identifies bugs, and verifies the performance of new updates.

10.8. Security

Regular security audits will be conducted every six months.
Compliance with data protection regulations (GDPR, CCPA, etc.) will be ensured by implementing encryption and regular data purges for unused data.
A penetration test will be conducted annually to detect and resolve vulnerabilities.

11. Conclusion

Overall we got to a good place in our project. Almost all of the planned features were implemented and during the project additional features were planned and executed. Most of the challenges in this project came from working with JavaFX and its janky usability. As for future improvements we'd like to implement data collection for restaurant owners to see how their businesses are doing and we'd like to add server sent events so that the user can track their order's status.