FINAL PROJECT DEFINITION AND DELIVERY
SOFTWARE ENGINEERING II

Presented by:

Michael Stiven Betancourt Gelves
Cristhian Yamith Cely Oliveros
Camilo Andres Herrera Gutierrez
Juan Manuel Cristancho Alvarez

Professor:
Carlos Andres Sierra Virguez

November 30th 2025

Universidad Nacional de Colombia
Engineering Faculty

2025

Final Project Definition and Delivery

Overview

This document outlines the requirements and deliverables for the course's **Final Project delivery**. Its objective is to deliver and showcase a complete, well-documented and functional web application that comprehensively integrates concepts and practices of the course within the project idea and its utilized technologies, developed from the beginning of the course.

1. Project Definition

The Click & Munch application is a modern, scalable microservices-based application designed to support a restaurant reservation platform. The system architecture separates concerns into distinct microservices, each responsible for specific business domains. Below we list the architectural decisions, design patterns employed, adherence to SOLID principles, and the integration of geospatial database technologies that enable location-based restaurant discovery features.

1.1 Business Definition

The system aims to allow restaurant customers to make reservations and place orders in advance, aiming to reduce wait times and improve service efficiency. Restaurants will be able to manage table availability, menus and orders, while monitoring customer satisfaction.

1.2 User Stories

The following are the defined User Stories for the first few projected sprints of the development cycle:
US-01: As a Customer, I want to register and log in so I can use the platform.

- Acceptance Criteria:
  Given the registration screen, when I enter valid details, then my account is created.
  Given the login screen, when I enter valid credentials, then I am authenticated.
  Given invalid credentials, when I try to log in, then I receive an error message.

US-02: As a Restaurant Manager, I want to create and manage my restaurant account.

- Acceptance Criteria:
  Given the registration form, when I submit valid business data, then the account is created and pending approval.

Given an approved restaurant account, when I update information, then it is saved.

US-03: As a System Administrator, I want to approve new restaurant accounts.

- Acceptance Criteria:
  Given a pending registration, when the admin approves it, then the restaurant is activated.
  Given a pending registration, when rejected, then the manager is notified with the reason.

US-04: As a User, I want to reset my password.

- Acceptance Criteria:
  Given the forgot password screen, when I submit my email, then a reset link is sent.
  Given a valid reset token, when I set a new password, then it updates successfully.

US-05: As a Customer, I want to view nearby restaurants using geolocation.

- Acceptance Criteria:
  Given location permission, when I open the discovery screen, then nearby restaurants appear.
  Given denied permission, when I try to view nearby restaurants, then I can enter a location manually.

US-06: As a Customer, I want to filter restaurants by rating, type, or distance.

- Acceptance Criteria:
  Given the restaurant list, when I apply filters, then the results update.
  Given multiple filters, when I clear them, then the list resets.

US-07: As a Customer, I want to see restaurant details (menu, hours, rating).

- Acceptance Criteria:
  Given I open restaurant details, when the data loads, then I see menu, hours, address, and rating.

US-08: As a Customer, I want top-rated restaurants (3 km radius) to appear first.

- Acceptance Criteria:
  Given location enabled, when I view the list, then restaurants within 3 km are sorted by rating and distance.

US-09: As a Customer, I want to select a restaurant and date/time to reserve a table.

- Acceptance Criteria:
  Given I choose a restaurant and time, when I submit, then a reservation is created if available.

US-10: As a Customer, I want to see available tables and capacities.

- Acceptance Criteria:
  Given I select date/time, when I check availability, then I see available table sizes.

US-11: As a Customer, I want to confirm my reservation.

- Acceptance Criteria:
  Given a provisional reservation, when I confirm it, then I receive a confirmation notification.

US-12: As a Restaurant Manager, I want to see all reservations in real-time.

- Acceptance Criteria:
  Given the dashboard, when new reservations occur, then they appear in real-time.

US-13: As a Customer, I want to cancel or modify my reservation.

- Acceptance Criteria:
  Given more than one hour before the reservation, when I modify/cancel, then it updates.
  Given less than one hour, when I try to cancel, then I see a policy restriction message.

US-14: As a Customer, I want to browse the restaurant's menu.

- Acceptance Criteria:
  Given I open the menu screen, when data loads, then I see categories, dishes, and prices.

US-15: As a Customer, I want to select dishes for each person.

- Acceptance Criteria:
  Given I build an order, when I add or remove items, then the order summary updates.

US-16: As a Customer, I want to place my order during reservation.

- Acceptance Criteria:
  Given I have a confirmed reservation, when I submit an order, then it's linked and visible in my profile.

US-17: As a Chef, I want to see the order list in real-time.

- Acceptance Criteria:
  Given a new pre-order, when it's submitted, then it appears instantly in the kitchen dashboard.

US-18: As a Waiter, I want to see which orders are ready.

- Acceptance Criteria:
  Given dishes are marked ready, when that occurs, then the waiter is notified.

US-19: As a Chef, I want to update the status of each dish.

- Acceptance Criteria:
  Given an order, when I update dish status, then it's reflected to the customer and waiter.

US-20: As a Customer, I want to see my order status.

- Acceptance Criteria:
  Given my order exists, when its status changes, then I see real-time updates in my profile.

US-21: As a Customer, I want to receive a notification before my reservation.

- Acceptance Criteria:
  Given a reservation, when 30 minutes remain, then I receive a push notification.

US-22: As a Waiter, I want to receive notifications when an order is ready.

- Acceptance Criteria:
  Given a dish is ready, when it's marked ready, then I receive a notification with order info.

US-23: As a Customer, I want to rate the restaurant and food.

- Acceptance Criteria:
  Given a completed reservation, when I rate, then my feedback updates the restaurant's average rating.

US-24: As a Restaurant Manager, I want to review ratings and comments.

- Acceptance Criteria:
  Given ratings exist, when I open the feedback dashboard, then I see comments and stats.

US-25: As a Restaurant Manager, I want to add/edit/remove menu items.

- Acceptance Criteria:
  Given manager access, when I add or modify menu items, then updates are saved and may need approval.

US-26: As a Restaurant Manager, I want to set an ETA for dishes.

- Acceptance Criteria:
  Given menu items, when I set ETA, then it's displayed to customers.

US-27: As a System Administrator, I want to review and approve menus.

- Acceptance Criteria:
  Given a pending menu, when I approve it, then it becomes visible to customers.

US-28: As a System Administrator, I want to manage users and restaurants.

- Acceptance Criteria:
  Given the admin panel, when I search and filter, then I can deactivate or reactivate accounts.

US-29: As a System Administrator, I want to generate reports.

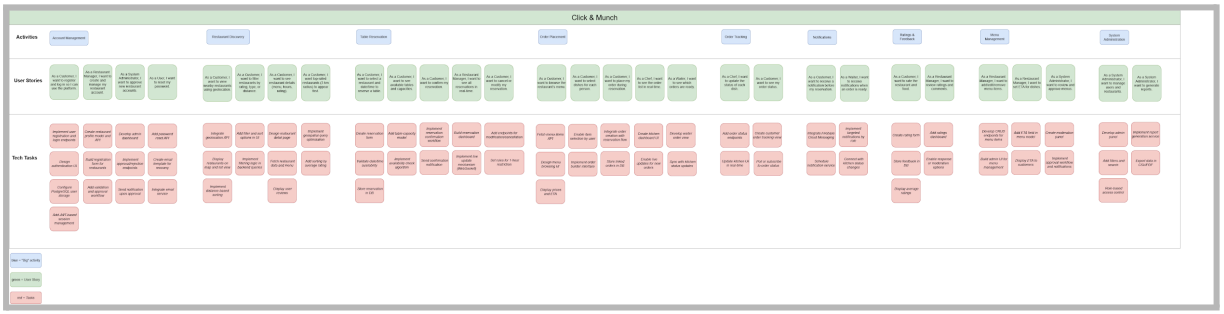
- Acceptance Criteria:
  Given the reporting screen, when I select parameters, then I can export reports in CSV or PDF.

1.3 Business Processes

TO-DO

1.4 User Story Mapping (USM)

An User Story Mapping exercise was conducted in the earlier stages of development.



*The attached PDF file includes the USM diagram in full.

2. Project Requirements

2.1 Functional Requirements

The following Requirements are considered Functional Requirements for the application. They include requirements beyond the MVP delivery.

FR-01. User Authentication and Management

- FR-01.1 The system shall allow customers to register with name, email, and password.
- FR-01.2 The system shall enable login/logout functionality.
- FR-01.3 Users shall be able to update their profile information.
- FR-01.4 Restaurants shall register their business data and contact information.
- FR-01.5 The system administrator shall approve or reject restaurant registrations.

FR-02. Restaurant and Menu Management

- FR-02.1 Restaurants shall create, edit, and delete menu items.
- FR-02.2 Menu items shall include name, category, description, price, and ETA.
- FR-02.3 Menus shall be subject to administrative approval before publication.

FR-03. Geolocation and Restaurant Listing

- FR-03.1 The system shall automatically obtain the user's geolocation.
- FR-03.2 Restaurants shall be listed within a 3 km radius, sorted by rating and distance.
- FR-03.3 The system shall display restaurants in both list and map view.

FR-04. Reservations

- FR-04.1 Customers shall select restaurant, date, time, and number of people.
- FR-04.2 The system shall confirm reservations automatically based on availability.
- FR-04.3 Restaurants may temporarily disable reservations when full.
- FR-04.4 Customers shall modify or cancel reservations at least one hour in advance.
- FR-04.5 The system shall send push notifications to remind and confirm reservations.

FR-05. Orders

- FR-05.1 Customers shall place their order alongside the reservation.
- FR-05.2 Orders can be modified or canceled before the cutoff time.
- FR-05.3 Restaurants shall receive the order automatically after confirmation.
- FR-05.4 Chefs shall update dish status: ‘Preparing’, ‘Ready’, ‘Served’, ‘Delivered’ ‘Cancelled’.
- FR-05.5 Waiters shall view ready orders for serving.

FR-06. Feedback and Ratings

- FR-06.1 Customers shall rate restaurants and dishes after their visit.

- FR-06.2 Customers may optionally rate the waiter.
- FR-06.3 The system shall compute average restaurant ratings.

FR-07. Notifications

- FR-07.1 The system shall send push notifications for upcoming reservations.
- FR-07.2 Restaurants shall be notified of new or canceled reservations.
- FR-07.3 Waiters shall be notified when dishes are ready to serve.

FR-08. Roles and Permissions

- FR-08.1 Each role (Customer, Manager, Waiter, Chef, Administrator) shall have distinct privileges.
- FR-08.2 Only managers may modify menus and restaurant data.
- FR-08.3 Chefs may only update dish preparation statuses.
- FR-08.4 Waiters may only view ready orders and mark as served.

FR-09. History

- FR-09.1 Customers shall access their order and reservation history.
- FR-09.2 Restaurants shall review service performance and customer feedback history.

2.2 Non-functional Requirements

The following Requirements are considered Nopn-Functional Requirements for the application. They also include requirements beyond the MVP delivery.

NFR-01. Availability

- The system should be available 99.5% of the time.

NFR-02. Performance

- Average response time for any request should not exceed 3 seconds.

NFR-03. Scalability

- The application must be scalable to support multiple restaurants and users concurrently.

NFR-04. Security

- All communications must be encrypted using HTTPS and SSL.

NFR-05. Usability

- The mobile and web interfaces should be intuitive and user-friendly.

NFR-06. Architecture

- NFR-06.1: The backend should be designed with RESTful APIs.
- NFR-06.2: Database should be normalized to maintain data integrity.

NFR-07. Compliance

- The system must comply with local data protection and privacy regulations.

NFR-08. Localization

- The solution should support localization (e.g., English and Spanish).

NFR-09. Real-Time Updates

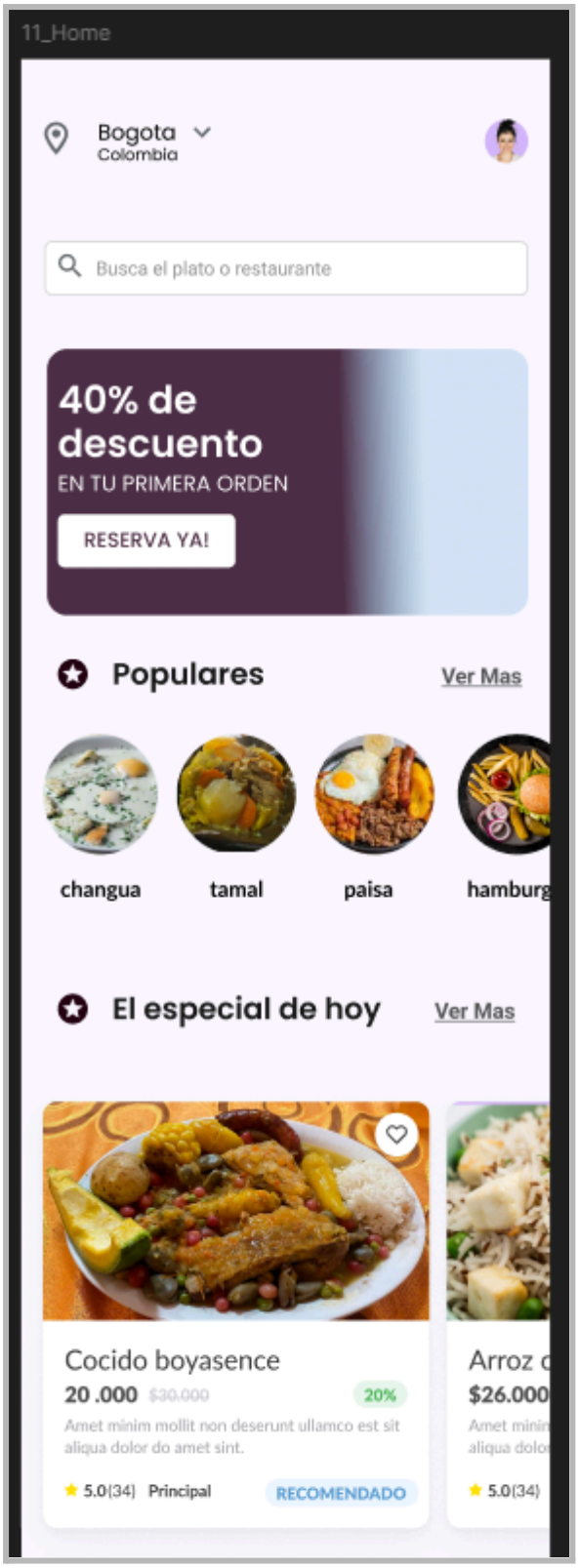
- Restaurant staff interface must support real-time updates for orders and reservations.

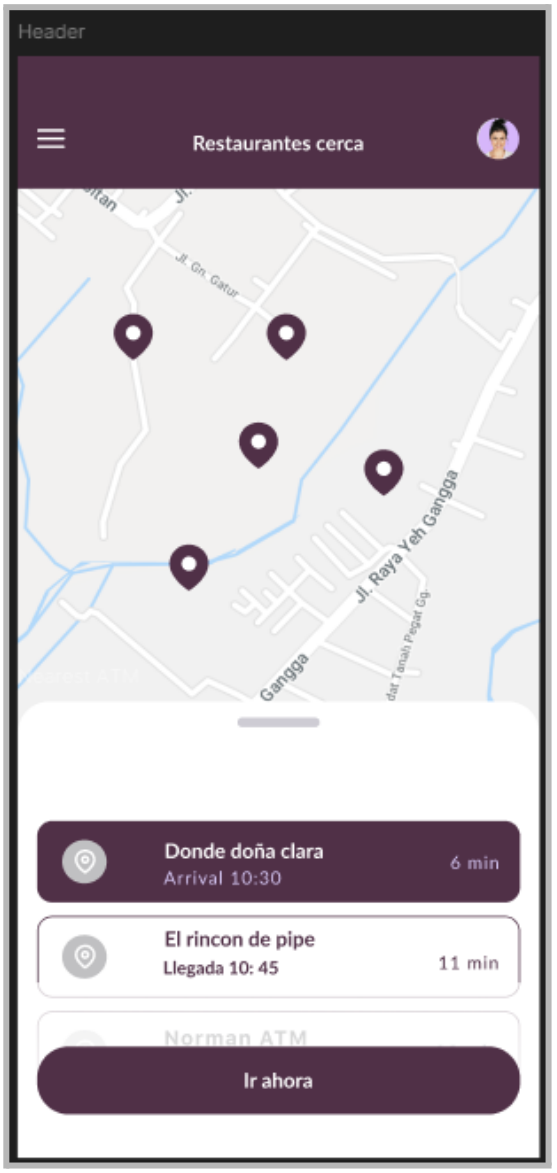
2.3 Mockups

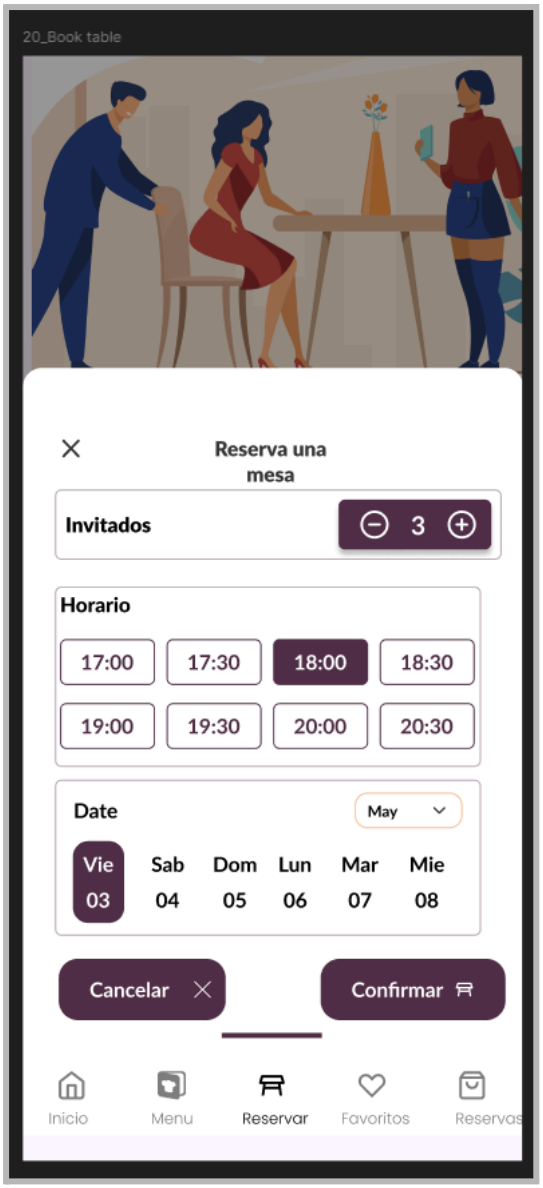
Frontend application mockups made for the Workshop 2 stage are available at https://www.figma.com/design/OsUJLRGwAVkChvzjCQdMFN/MockupsClick-Munch?node-id=0-1&p=f


01_Splash Screen
Reservas inteligentes.
CLICK
MUNCH
Mejores experiencias


Login
Ingresa
Usuario, correo, numero celular
Contrasña
Olvidaste tu contraseña?
Ingresar
O registrate con


11_Home
Bogota
Colombia
Busca el plato o restaurante
40% de descuento
EN TU PRIMERA ORDEN
RESERVA YA!
Populares
Ver Mas
changua
tamal
paisa
hamburg
El especial de hoy
Ver Mas
Cocido boyasence
20 .000
$30.000
20%
Amet minim mollit non deserunt ullamco est sit aliqua dolor do amet sint.
5.0(34) Principal
RECOMENDADO
Arroz c
$26.000
5.0(34)


Header
Restaurantes cerca
Jl. Raya Yeh Gangga
Donde doña clara
Arrival 10:30
6 min
El rincon de pipe
Llegada 10: 45
11 min
Norman ATM
Ir ahora

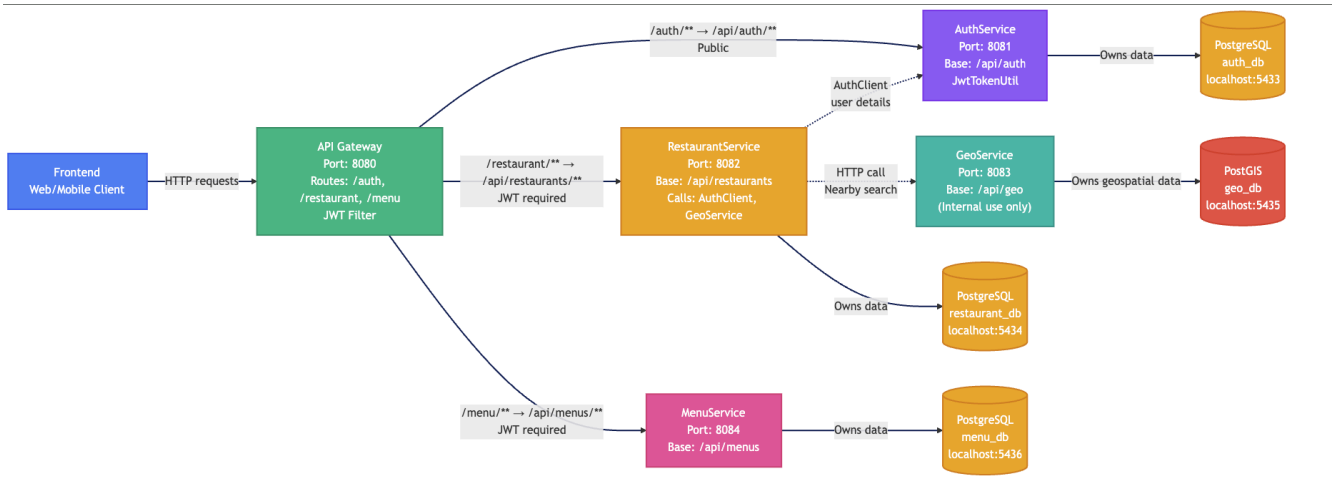


3. Design

The Click & Munch application was designed from the beginning with an approach oriented in a microservice architecture. Each microservice contains its own database, repository, service, entities and distinct configurations, and as such, their responsibilities are limited to the functions delegated within them. Inter-service communication is managed using API controller calls.

3.1 Architecture Diagram

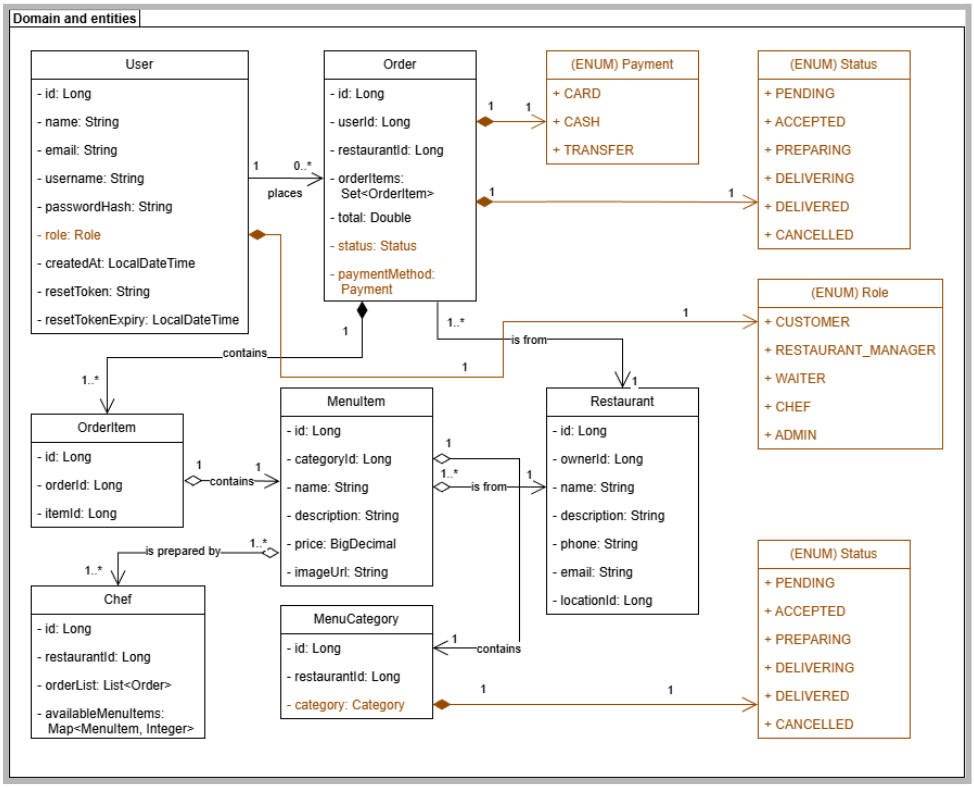
The following high-level interaction diagram describes the interaction of the reservation system.



3.2 Class Diagram

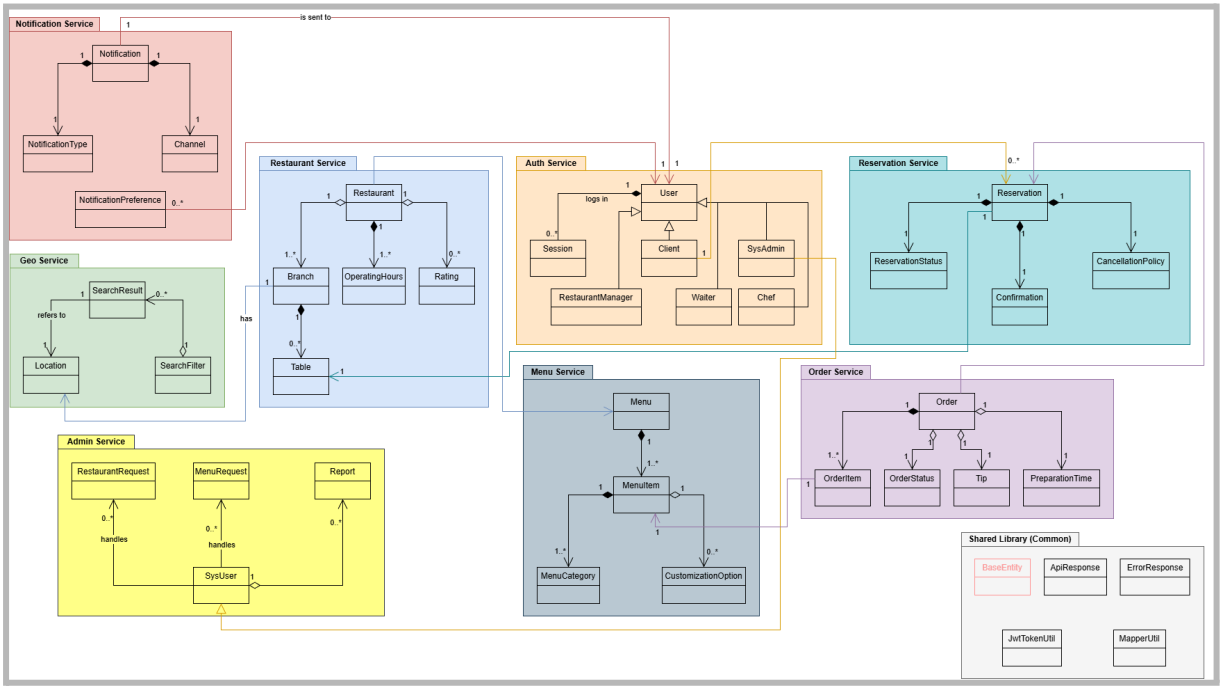
3.2.1 Domain/Entities

The following is the UML Class Diagram for domain/entities within the Backend application component of Click & Munch. This diagram references processes beyond the MVP delivery of the application:

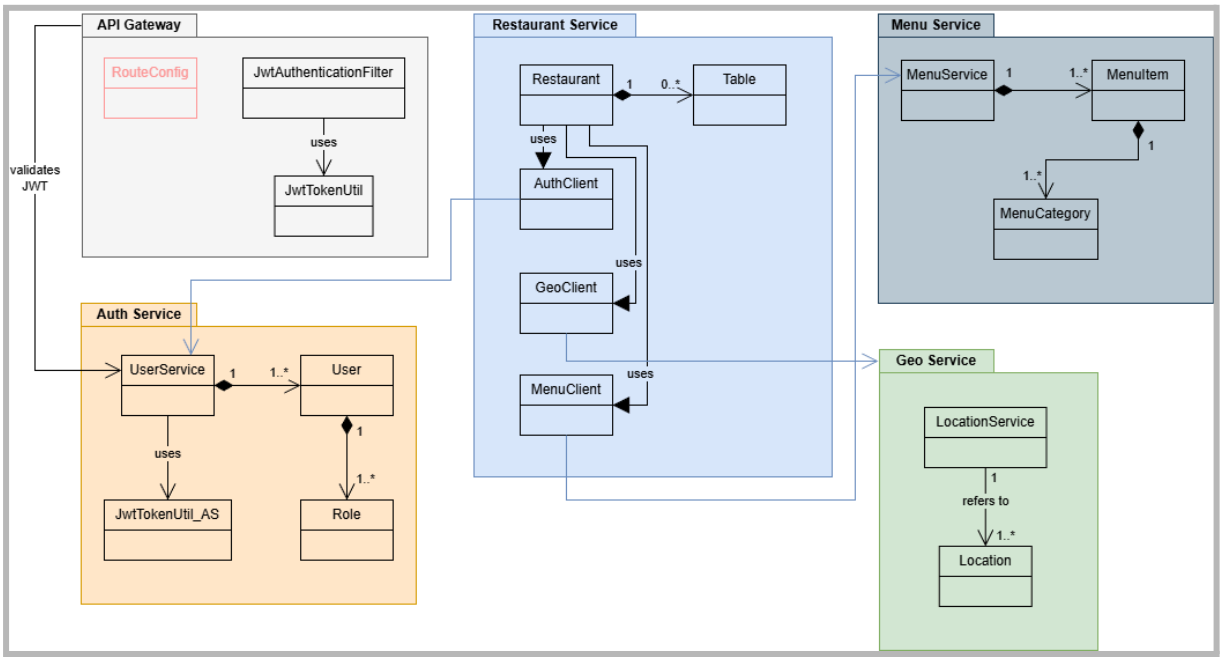


3.2.2 Microservices

The following is the initial UML Class Diagram for the Click & Munch application microservices. This initial estimation includes projected services beyond the scope of the MVP delivery and considering the initial estimations for the development:



For this MVP delivery, the Click & Munch application presents the following Class Diagram for its microservices:



3.3 SOLID Implementation

3.3.1 Single Responsibility Principle (SRP)

The application adheres to SRP by assigning single, well-defined responsibilities to each component:

- **AuthService**: Solely responsible for authentication and authorization
- **RestaurantService**: Focuses exclusively on restaurant management
- **MenuService**: Dedicated to menu and menu item management
- **GeoService**: Specializes in geospatial operations and location queries

Within each service, classes maintain single responsibilities. Controllers handle HTTP concerns, services encapsulate business logic, and repositories manage data persistence.

3.3.2 Open/Closed Principle (OCP)

The architecture supports the Open/Closed Principle through several mechanisms:

- **Extension Without Modification**: New authentication methods can be added to AuthService without modifying existing code
- **Plugin Architecture**: Additional filters and security strategies can be registered in the API Gateway
- **Repository Abstraction**: New persistence mechanisms can replace existing repositories without affecting business logic
- **Service Expansion**: New microservices can be added and registered with the API Gateway without modifying core gateway logic

3.3.3 Liskov Substitution Principle (LSP)

The application respects LSP through proper interface contracts:

- **Spring Data Repositories**: All repositories implement consistent interfaces (CrudRepository, JpaRepository)
- **Service Interfaces**: Services expose contracts that allow substitution of implementations
- **HTTP Clients**: Client implementations can be substituted without affecting consumer code

3.3.4 Interface Segregation Principle (ISP)

The design isolates interfaces to prevent clients from depending on methods they do not use:

- **Focused Controllers**: Each controller exposes only relevant endpoints for its domain

- **Specialized Services**: Services implement focused interfaces for specific operations
- **DTOs**: Request and response DTOs contain only necessary fields, avoiding bloated payload structures
- **Repository Contracts**: Repositories provide minimal required methods rather than comprehensive data access interfaces

3.3.5 Dependency Inversion Principle (DIP)

The application inverts dependencies through Spring's dependency injection framework:

- **Abstraction Over Concretion**: Components depend on abstractions (interfaces) rather than concrete implementations
- **Constructor Injection**: Services receive dependencies through constructors, making dependencies explicit and testable
- **Service Interfaces**: Business logic depends on service abstractions, not implementations
- **Repository Abstraction**: Business logic depends on repository interfaces, not database-specific code

3.4 Design Patterns

Within the Backend component of the Click & Munch application, the following design patterns were implemented in the different services.

3.4.1 API Gateway Pattern

The API Gateway pattern serves as the primary architectural design pattern in this system. The gateway provides several critical benefits:

- **Single Entry Point**: Eliminates the need for clients to know about individual service locations
- **Path Rewriting**: Abstracts internal API structure from external consumers
- **Cross-Cutting Concerns**: Centralizes CORS handling, request validation, and security
- **Request Routing**: Implements intelligent routing logic based on request paths

3.4.2 Layered Architecture Pattern

Each microservice implements a classical layered architecture consisting of four distinct layers:

1. **Controller Layer**: Handles HTTP requests and responses, provides REST endpoints
2. **Service Layer**: Contains business logic and orchestrates operations
3. **Repository Layer**: Manages data access and persistence operations
4. **Data Transfer Objects (DTOs)**: Define request and response data structures

This layering promotes separation of concerns and facilitates testing at each layer independently.

3.4.3 Repository Pattern

The application uses Spring Data repositories to abstract data access logic. Repositories like `UserRepository` provide a clean interface for database operations without exposing underlying persistence details. This pattern enables:

- Abstraction of database technology
- Simplified unit testing through mocking
- Consistency in data access methods across the application
- Encapsulation of query logic

3.4.4 Builder Pattern

Entity classes, such as `User`, utilize Lombok's `@Builder` annotation to implement the Builder pattern. This pattern provides:

- Cleaner object construction with optional parameters
- Enhanced code readability
- Reduced constructor overloading
- Immutability support

3.4.5 Dependency Injection Pattern

Spring Framework's dependency injection container manages component lifecycle and dependencies. Components are annotated with `@Service`, `@RestController`, `@Repository`, and `@Bean` stereotypes, enabling automatic wiring and loose coupling between components.

3.4.6 Client/Integration Pattern

Services communicate with each other through HTTP-based integration. The RestaurantService, for instance, uses an `AuthClient` to query user details from the AuthService. This approach enables:

- Asynchronous service-to-service communication
- Service independence and loose coupling
- Potential for circuit breaker patterns and resilience

3.4.7 Security Filter Pattern

The `JwtAuthenticationFilter` in the API Gateway implements the filter pattern for cross-cutting security concerns. This filter:

- Validates JWT tokens on protected routes
- Allows public access to authentication endpoints
- Enforces authentication without modifying service code
- Provides centralized security policy management

4. Implementation

4.1 Project Repository and File Structure

The project is available on GitHub at:
https://github.com/msbetancourtge/IngesoftII

Global Structure: The full repository for the Software Engineering II class is divided into two main sections: the root, for the main `README.md` file, and the class Workshops made through the course's duration. Each Workshop folder contains the required files and diagrams for their correspondent activities.
In general, the project repository is structured as follows:

```
│
├── ClickAndMunchApp/
│    ├── frontend/
│    ├── backend/
│    ├── .gitignore
├── Workshop-1/
│    ├── README.md
```

```
│     ├── USM_Final.pdf
│     ├── Workshop N°1.pdf
├── Workshop-2/
│     ├── Architecture C&M.png
│     ├── Class Diagram, app.png
│     ├── Class Diagram, entities.png
│     ├── Order.jpeg
│     ├── Readme.md
│     ├── Relational Diagram_tmp.png
│     ├── Restaurant Reg.jpeg
│     ├── Workshop N°2.pdf
├── Workshop-3/
│     ├── Workshop N°3.pdf
├── Workshop-4/
│     ├── Architecture Diagram.png
│     ├── Workshop N°4.pdf
├── .gitignore
├── README.md
```

Source Code: Source code for the Frontend and Backend are located within the `./ClickAndMunchApp/` directory. The structure for the source code folder is detailed in the section **4.5 Backend Services**.

4.2 Good Practices

4.3 Tech Stack

4.3.1 Backend

For the Backend component and its microservices, Java with SpringBoot have been used as the language and framework of choice. The database for all services is PostgreSQL.

4.3.2 Frontend

The Frontend applications use TypeScript, React and React Native with Expo for its main components.

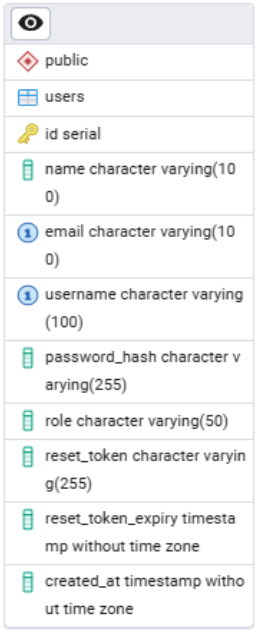
4.4 Database Implementation

Each microservice within the Click & Munch application has been designed to have its own database with one or more tables inside it, to facilitate management inside each microservice.

PostgreSQL has been used as the primary relational database management system for all microservices.

The following are the ERD's and concrete `schema.sql` scripts for the databases and tables of each microservice.

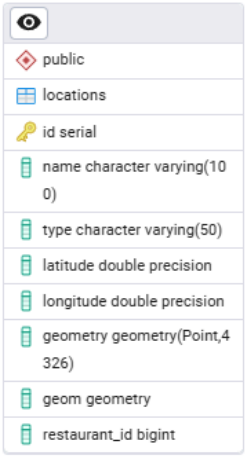
4.4.1 AuthService



SQL

```
CREATE TABLE IF NOT EXISTS users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
       username VARCHAR(100) UNIQUE NOT
NULL,
    password_hash VARCHAR(255) NOT NULL,
    role VARCHAR(50) NOT NULL,
    reset_token VARCHAR(255),
    reset_token_expiry TIMESTAMP,
          created_at TIMESTAMP DEFAULT
CURRENT_TIMESTAMP
);
```

4.4.2 GeoService



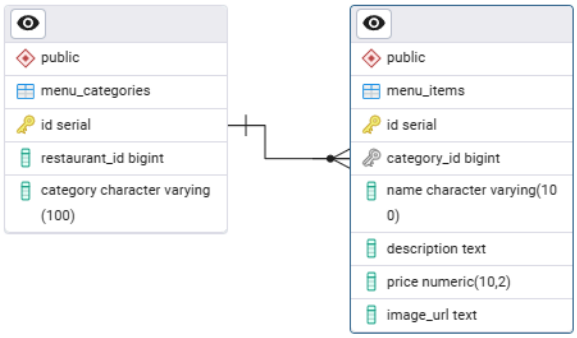
SQL

```
CREATE EXTENSION IF NOT EXISTS postgis;

CREATE TABLE IF NOT EXISTS locations (
     id SERIAL PRIMARY KEY,
     restaurant_id BIGINT,
     name VARCHAR(100) NOT NULL,
     type VARCHAR(50) NOT NULL,
     latitude DOUBLE PRECISION NOT NULL,
         longitude DOUBLE PRECISION NOT
NULL,
            geom geometry(Point, 4326)
GENERATED ALWAYS AS (

ST_SetSRID(ST_MakePoint(longitude,
latitude), 4326)
         ) STORED
);

CREATE INDEX IF NOT EXISTS
idx_locations_geom ON locations USING
GIST (geom);
```

```
CREATE INDEX IF NOT EXISTS
idx_locations_restaurant_id ON locations
(restaurant_id);
```

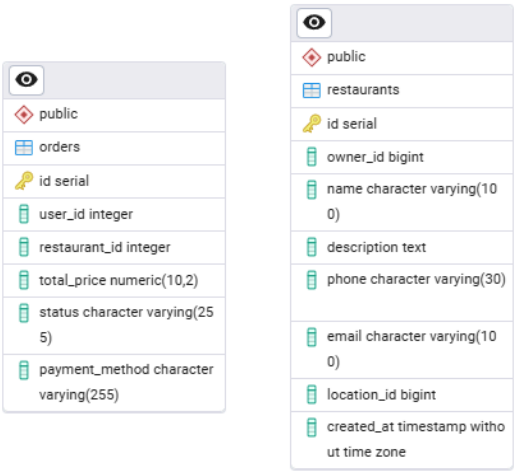
```
);
```

4.4.3 MenuService




```
SQL
CREATE TABLE IF NOT EXISTS
menu_categories (
          id SERIAL PRIMARY KEY,
               restaurant_id BIGINT NOT
NULL,
              category VARCHAR(100) NOT
NULL
);

CREATE TABLE IF NOT EXISTS menu_items(
          id SERIAL PRIMARY KEY,
          category_id BIGINT NOT NULL,
          name VARCHAR(100) NOT NULL,
          description TEXT,
          price NUMERIC(10,2) NOT NULL,
          image_url TEXT,
               FOREIGN KEY (category_id)
REFERENCES menu_categories(id) ON DELETE
CASCADE
```

4.4.4 RestaurantService




```
SQL
CREATE TABLE IF NOT EXISTS
restaurants (
       id SERIAL PRIMARY KEY,
       owner_id BIGINT NOT NULL,
       name VARCHAR(100) NOT NULL,
       description TEXT,
       phone VARCHAR(30),
       email VARCHAR(100),
       location_id BIGINT NOT NULL,
        created_at TIMESTAMP DEFAULT
NOW()
);
```

The simultaneous creation of the databases for each microservice is orchestrated by a dedicated dockerfile. It is detailed in the following sections (**4.5. Backend Services**; **6.1. Deployment Configuration**).

4.5 Backend Services

4.5.1 Source Code Structure

Source Code: Source code for the frontend and backend are both located within the `~/ClickAndMunchApp/` directory.

Global Structure: The application repository is divided into two main sections for backend and frontend.

```
│
├── backend/
│     ├── APIGateway/
│     ├── AuthService/
│     ├── GeoService/
│     ├── MenuService/
│     ├── RestaurantService/
│     ├── docker-compose.yml
├── frontend/
│     ├── dashboard/
│     ├── mobile/
├── .gitignore
```

Per-Service Structure: In general, the backend services are structured as follows:

```
│
├── BackendService/
│     ├── gradle/
│     ├── src/
│     │     ├── main/
│     │     │     ├── java.com.clickmunch.BackendService/
│     │     │     │     ├── config/
│     │     │     │     ├── controller/
│     │     │     │     ├── dto/
│     │     │     │     ├── entity/
│     │     │     │     ├── respository/
│     │     │     │     ├── service/
│     │     │     │     ├── BackendServiceApplication.java
│     │     │     ├── resources/
│     │     │     │     ├── application.yaml
│     │     │     │     ├── schema.sql
│     │     ├── test/
│     │     │     ├── java.com.clickmunch.BackendService/
│     ├── build.gradle
│     ├── settings.gradle
```

4.5.2 Configuration

As mentioned in section **4.4. Database Implementation**, YAML located in the root of the backend directory is provided for database creation and connection. The full file is described in section **6.1. Deployment Configuration**.

4.5.3 REST API endpoints

4.5.3.1 API Docs setup and deployment

Each Java-Spring Boot microservice implements an automatic API documentation using an OpenAPI/Swager UI integration with the Spring Doc Java library (`sprindoc-openapi`). This library examines an application at runtime to infer API semantics based on Spring configurations, project structure and various annotations, and then, automatically generates documentation in JSON, YAML and HTML format for the application API. The `swagger-api` annotations allow for completing the documentation.

The Spring Doc library supports:

- OpenAPI 3
- Spring-boot v3 (Java 17 & Jakarta EE 9)
- JSR-303 Bean validation.
- Swagger UI
- Scalar
- OAuth2

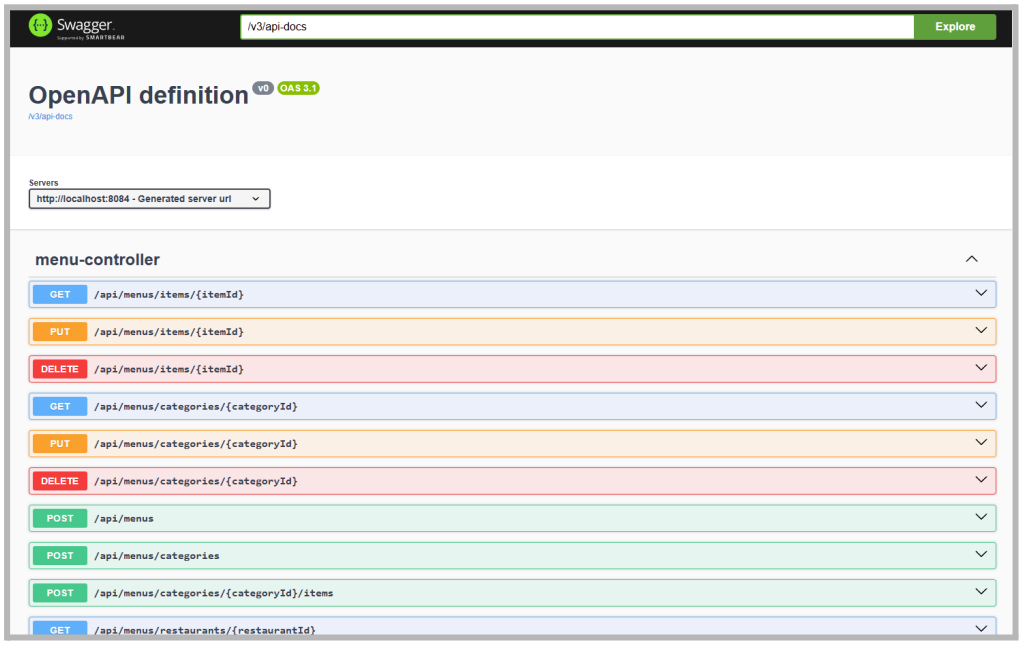
To get started with the usage of the automatic API docs library, it is necessary to include it as a dependency for the services. This project uses Gradle for the Java-Spring Boot services, and as such, it needs to be added as a dependency in the `build.gradle` files for each service:

```JSON
dependencies {
(...existing dependencies...)
// Docs
implementation 'org.springdoc:springdoc-openapi-starter-webmvc-ui:2.8.14'
}
```

The version of Spring Doc to be used depends on the Spring Boot version of the application itself. For this project, the version `4.0.0` of the Spring Framework is used, and as such, using a `2.8.x` or higher version of the Spring Doc OpenAPI library will suffice. Doing this automatically deploys Swagger UI to a Spring Boot application. Documentation will be available in HTML using the official `swagger-ui` interface.

The Swagger UI OpenAPI docs page will be available at `http://server:port/context-path/swagger-ui.html`, or more generally, a localhost address such as `http://localhost:8080/swagger-ui/index.html`.

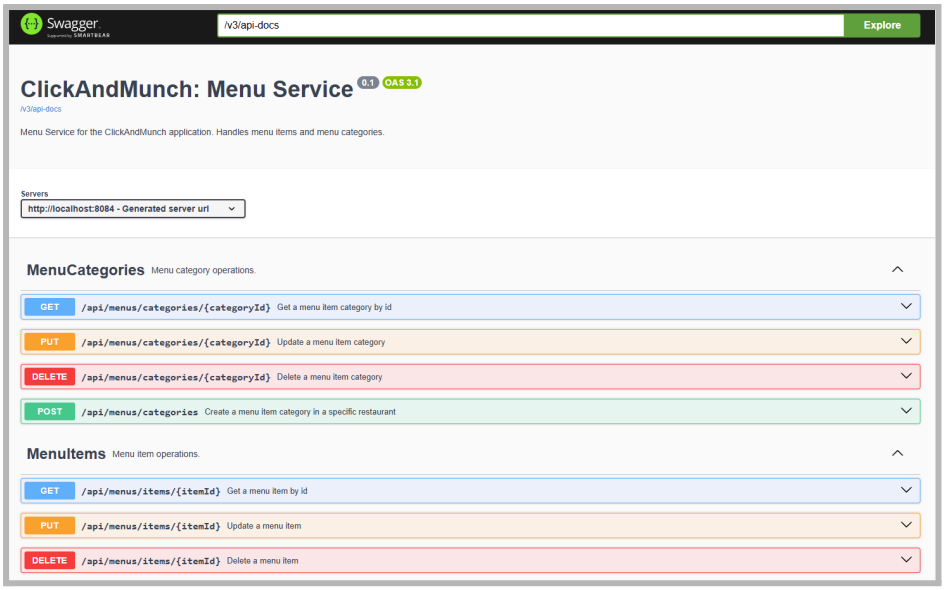
Documentation will be available in YAML format as well, on the following path: `/v3/api-docs.yaml`. A freshly-setup Swagger UI will look like this:



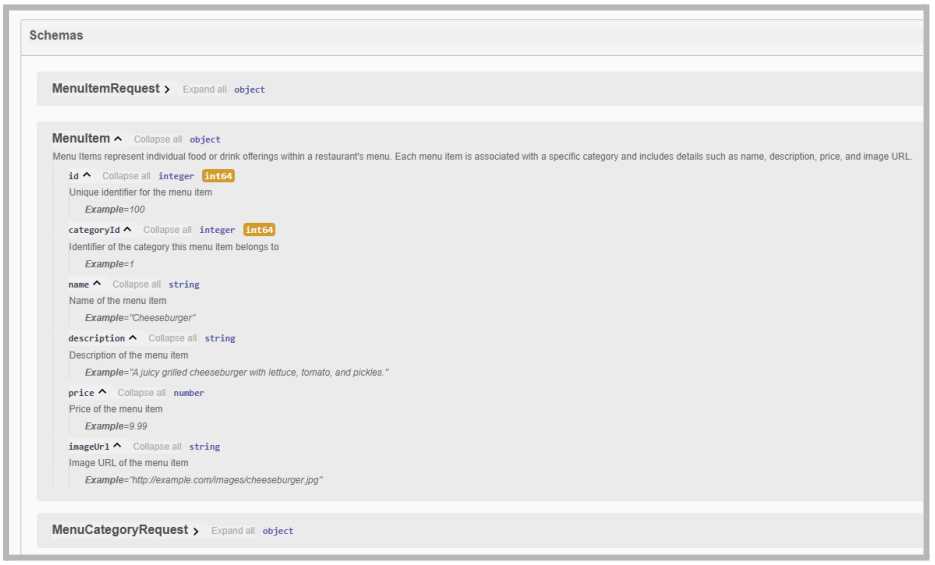
OpenAPI UI docs. Default configuration and view of the Menu Service.

The library provides different Java annotations to further document the endpoint/path operations and provide clear context and explanations. These annotations can be added to the controllers and entities or models to provide context on how the API functions. This is done using the `io.swagger.v3.oas.annotations` package. Some of the useful annotations include:

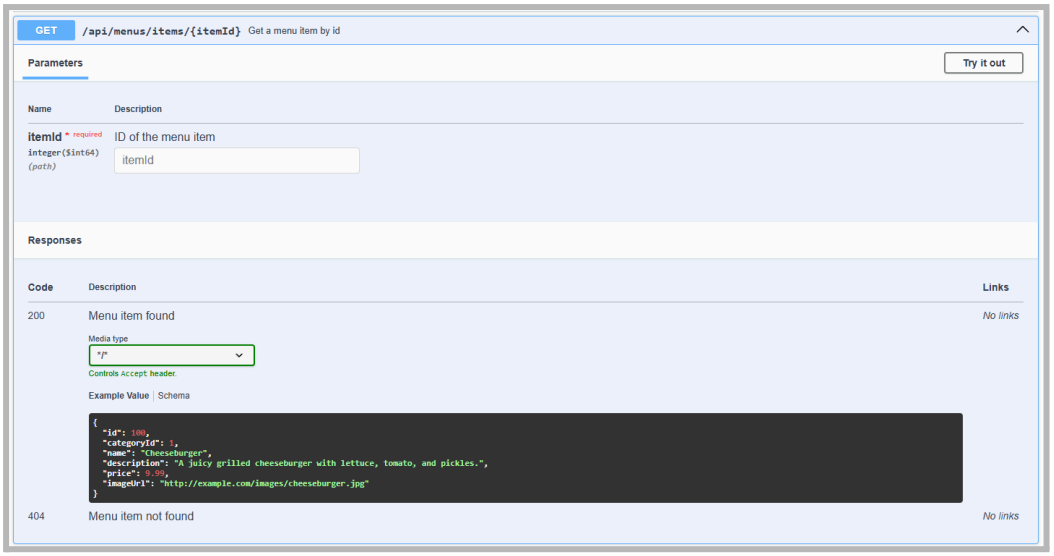
- `@Operation`: Helps adding a summary and description, as well as other information to a given operation.
- `@Tag`: Tags a controller. Adds a description to it.
- `@Parameter`: Helps describe path, query and request body parameters.
- `@ApiResponse` and `@ApiResponses`: Customize the response body descriptions, overriding the default response schema provided by OpenAPI.
- `@Schema`: Helps annotate the models and/or entities. The class and parameters can be annotated to describe its properties.



UI docs for the Menu Service. Custom titles, descriptions and tags for the controller can be seen in the main view.



Customized schema documentation for the MenuItem entity. Includes a general description of the schema itself, as well as descriptions and examples for the parameters.



Documentation for the MenuItem GET operation within the Swagger UI.

4.5.3.2 API Documentation

The following is the documentation for the available POST endpoints within the microservices. GET, PUT and DELETE operations follow a similar structure unless specified otherwise.

AuthService: `localhost:8081/`

- **POST** - Register
  `/api/auth/register`
  Request Headers
  Content-Type: `application/json`
  Request Body

```
{
    "name": "Pepito Perez",
    "email": "pepo@example.com",
    "username": "pepito",
    "password": "789456",
    "role": "RESTAURANT_MANAGER"
}
```

- **POST** - Login
  `/api/auth/login`
  Request Headers
  Content-Type: `application/json`
  Request Body

```
{
    "username": "juancho",
    "password": "789456"
}
```

- **POST** - Reset

`/auth/password-reset/request`

Request Headers

Content-Type: `application/json`

Request Body

```
{
    "email": "cristhian@example.com"
}
```

- **POST** - Confirm
  `/auth/password-reset/confirm`

  Request Headers

  Content-Type: `application/json`

  Request Body

```
{
    "resetToken":
    "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJjcmlzdGFuY2hvIiwiaWF0IjoxNzYzNj
g2NzEzLCJleHAiOjE3NjM2OTAzMTN9.aYlz2bMuFWPlR25Ydts0TBPNg
bHAZ41W-NZEwVpuR9M",
    "newPassword": "jamaica"
}
```

GeoService: `localhost:8083/`

- **POST** - Add Location
  `/api/geo/locations`

  Request Headers

  Content-Type: `application/json`

  Request Body

```
{
    "name": "McDonalds Salitre",
    "type": "RESTAURANT",
    "latitude": 52.74757,
    "longitude": -72.952145
}
```

- **POST** - Find Nearby
  `/api/geo/nearbyRequest`

  Request Headers

  Content-Type: `application/json`

  Request Body

```
{
    "latitude": 52.75214,
    "longitude": -72.886547,
    "radiusInKm": 5
}
```

MenuService: `localhost:8084/`

- **POST** - Create Menu Category
  `/api/menus/categories`
  Request Headers
  Content-Type: `application/json`
  Request Body
  ```
  {
    "restaurantId": 2,
    "category": "ENTRADA"
  }
  ```
- **POST** - Create Menu Item
  `/api/menus/categories/{categoryId}/items`
  Request Headers
  Content-Type: `application/json`
  Request Body
  ```
  {
    "name": "Cheeseburger",
    "description": "A juicy grilled cheeseburger with lettuce, tom…",
    "price": 9.99,
    "imageUrl": "http://example.com/images/cheeseburger.jpg"
  }
  ```
- **POST** - Create full menu (for a Restaurant)
  `/api/menus`
  Request Headers
  Content-Type: `application/json`
  Request Body
  ```
  {
    "restaurantId": 2,
    "categories": [
      {
        "category": "PLATO",
        "items": [
          {
            "name": "Cheeseburger",
            "description": "A juicy grilled cheeseburger with lett…",
            "price": 9.99,
            "imageUrl": "http://example.com/images/cheeseburger.jpg"
          },
          {
            "name": "Hot Dog",
            "description": "Sausage and bun with your toppings!",
            "price": 7.99,
            "imageUrl": "http://example.com/images/hotdog.jpg"
          }
  ```

```
        ]
      }
    ]
  }
```

RestaurantService: `localhost:8082/`

- **POST** - Create Restaurant
  `/api/restaurants`
  Request Headers
  Content-Type: `application/json`
  Request Body

```
{
  "ownerId": 67,
  "name": "McDonald's Calle 45",
  "description": "Sucursal McDonald's de la Calle 45 en Bogotá.",
  "phone": "3999999999",
  "email": "mcdonalds.cl45@example.com",
  "latitude": 5.65846560,
  "longitude": -74.21365465
}
```

4.6 Frontend Applications

The Frontend component for Click & Munch includes two main applications: a mobile application for end users (clients) and a web dashboard for stores (restaurants).

The mobile application is structured as follows:

```
mobile
├── .expo
├── app
│   ├── (products-app)
│   ├── auth
│   └── _layout.tsx
├── assets
├── constants
├── core
│   ├── api
│   ├── auth
│   └── restaurants
├── helpers
│   └── adapters
│       └── secure-storage-adapter.ts
├── node_modules
```

```
├── presentation
│   ├── auth
│   ├── restaurants
│   └── theme
├── .env
├── .env.template
├── .gitignore
├── app.json
├── bun.lock
├── eslint.config.js
├── expo-env.d.ts
├── package.json
├── README.md
└── tsconfig.json
```

The dashboard (web application) is structured as follows:

```
dashboard
├── node_modules
├── public
├── src
│   ├── admin
│   │   ├── components
│   │   ├── layouts
│   │   └── pages
│   ├── assets
│   ├── auth
│   │   ├── layouts
│   │   └── pages
│   ├── components
│   │   └── ui
│   ├── lib
│   ├── app.router.tsx
│   ├── index.css
│   ├── main.tsx
│   └── RestaurantApp.tsx
├── .gitignore
├── bun.lock
├── components.json
├── eslint.config.js
├── index.html
├── package.json
├── README.md
```

```
├── tsconfig.app.json
├── tsconfig.json
├── tsconfig.node.json
└── vite.config.ts
```

Authentication

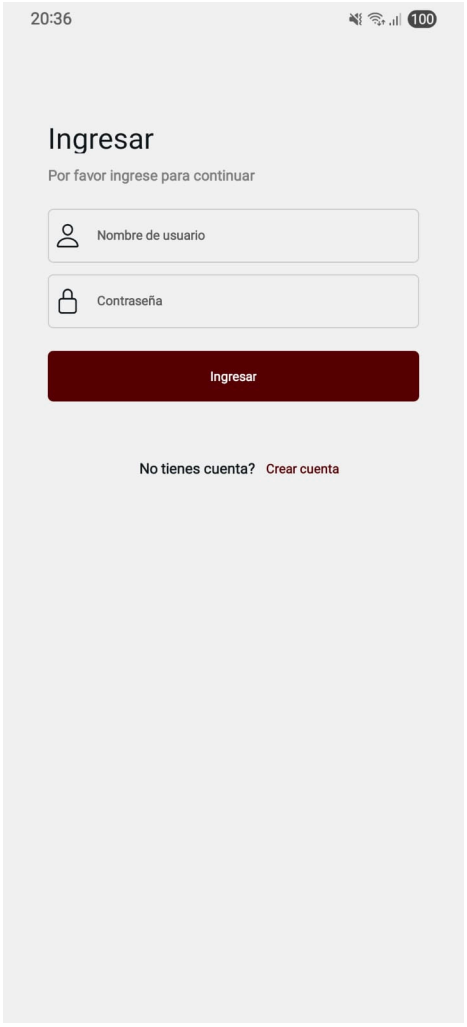
We use a core api in order to make only one connection with the backend services. To successfully create a connection we have to create a .env file to organize environmental variables.

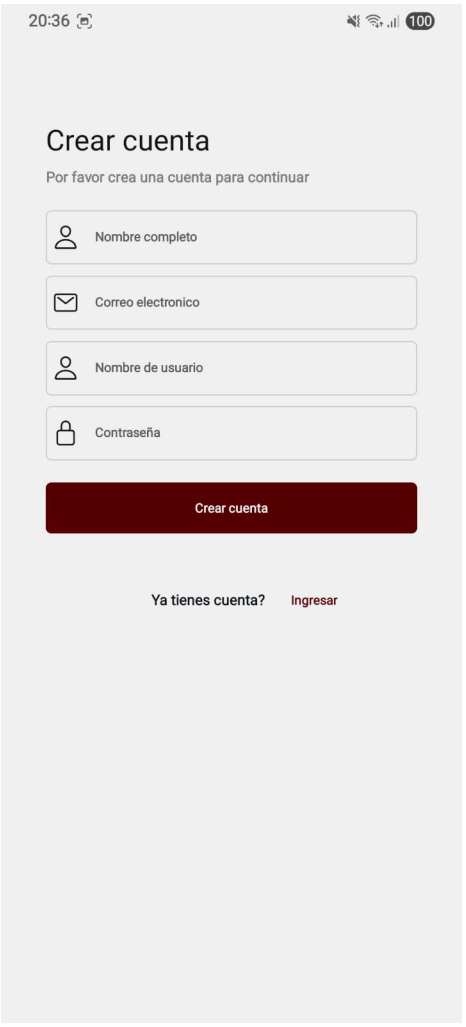
```
EXPO_PUBLIC_STAGE=dev

EXPO_PUBLIC_API_URL=http://192.168.1.14:8080
EXPO_PUBLIC_API_URL_IOS=http://localhost:8080
EXPO_PUBLIC_API_URL_ANDROID=http://192.168.1.14:8080
```

Also for keep the sessions we have to create a storage to manage it. We use the Adapter pattern in order to use external libraries and adapt to our purposes.

```
export class SecureStorageAdapater {

    static async setItem(key: string, value: string) {
        try {
            await SecureStore.setItemAsync(key, value);
        } catch (error) {
            Alert.alert("Error", "Failed to save data");
        }

    }

    static async getItem(key: string ) {
        try {
            return await SecureStore.getItemAsync(key);
        } catch (error) {
            Alert.alert("Error", "Failed to get data");
            return null;
        }
    }

    static async deleteItem(key: string) {
        try {
            await SecureStore.deleteItemAsync(key);
        } catch (error) {
            console.log(error);
            Alert.alert("Error", "Failed to delete data")
```

```
            }
        }
}
```





Web Application

Bienvenido

Ingresa a Click&Munch

Email

m@ejemplo.com

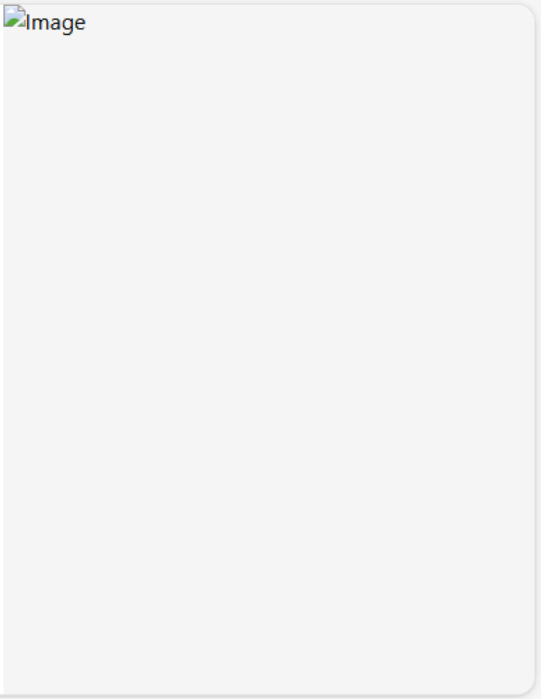
Contraseña Olvidaste tu contraseña?

Contraseña

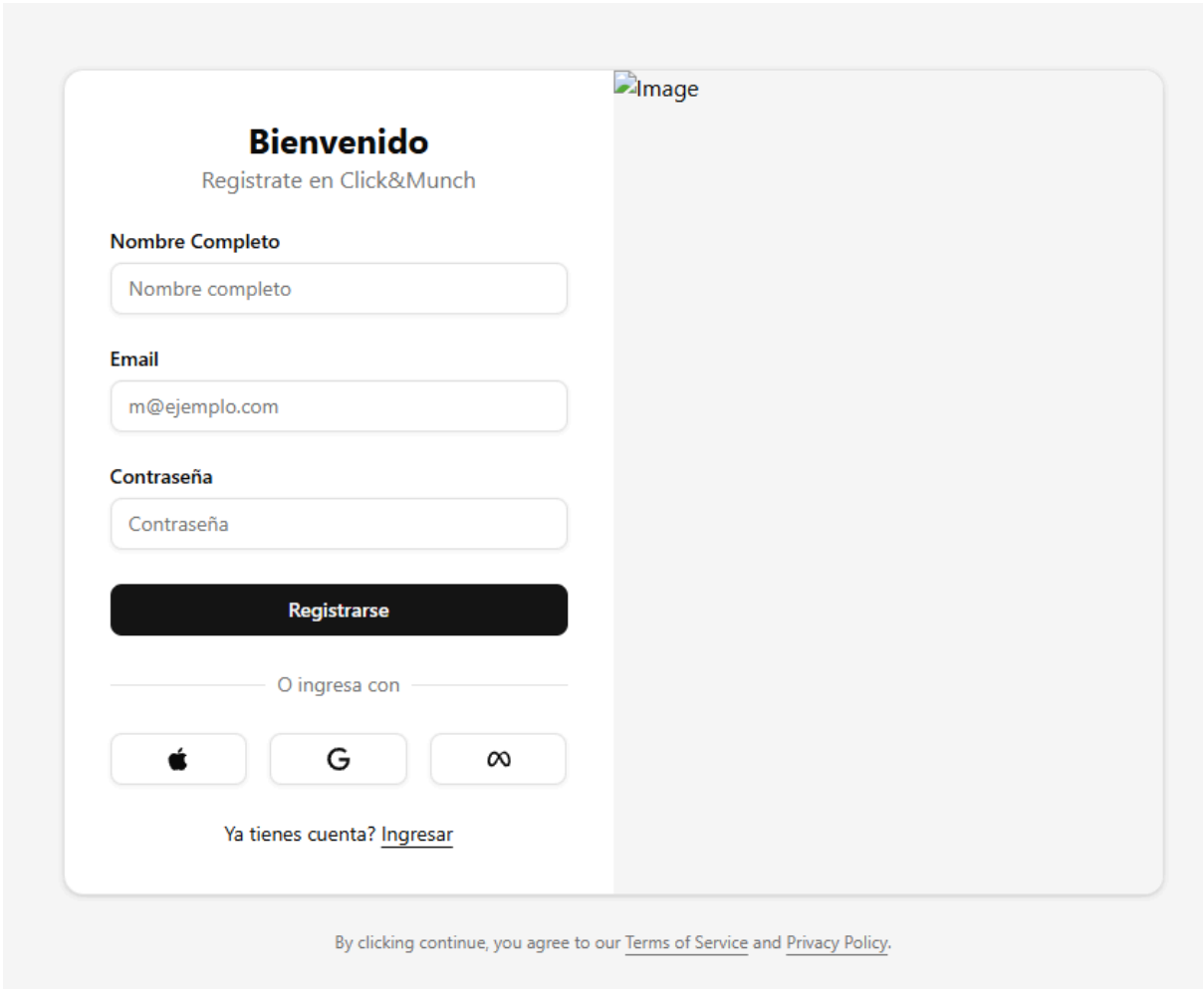
Ingresar

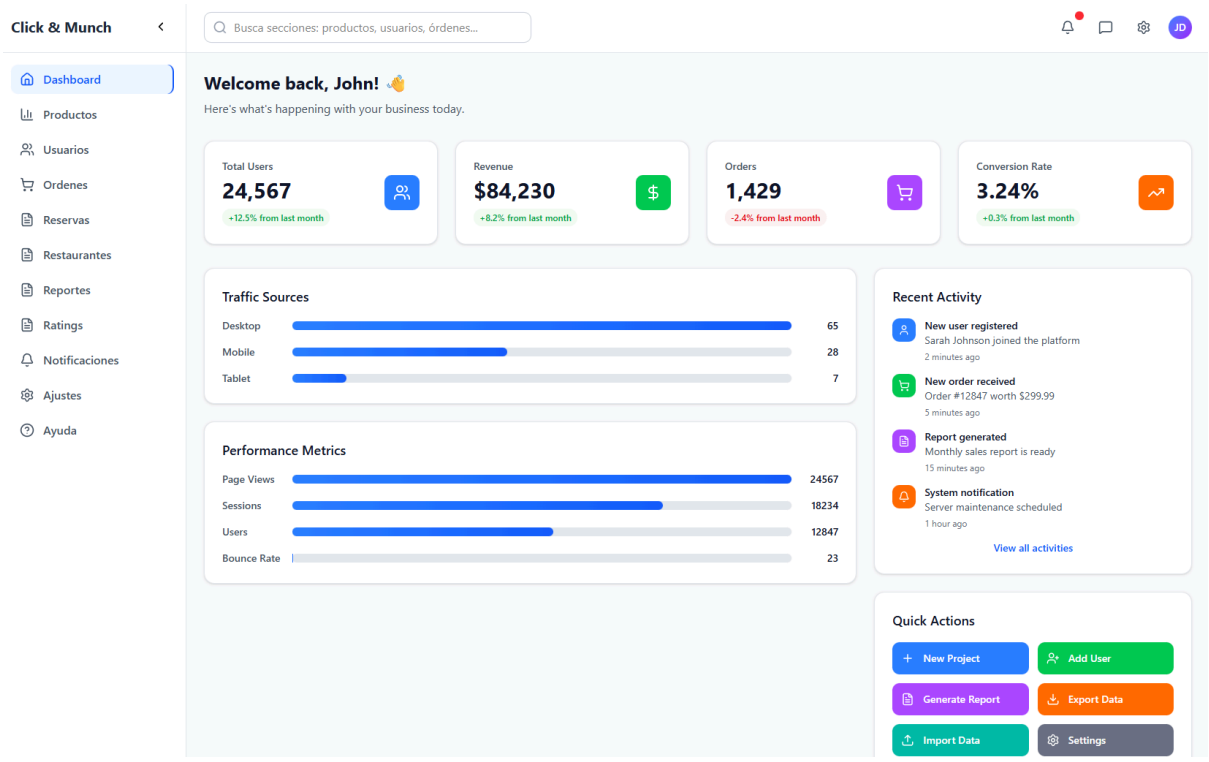
O ingresa con

No tienes cuenta? Crear cuenta



By clicking continue, you agree to our Terms of Service and Privacy Policy.


Bienvenido
Registrate en Click&Munch
Nombre Completo
Nombre completo
Email
m@ejemplo.com
Contraseña
Contraseña
Registrarse
O ingresa con
Ya tienes cuenta? Ingresar
Image
By clicking continue, you agree to our Terms of Service and Privacy Policy.


Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
Welcome back, John!
Here's what's happening with your business today.
Total Users
24,567
+12.5% from last month
Revenue
$84,230
+8.2% from last month
Orders
1,429
-2.4% from last month
Conversion Rate
3.24%
+0.3% from last month
Traffic Sources
Desktop 65
Mobile 28
Tablet 7
Performance Metrics
Page Views 24567
Sessions 18234
Users 12847
Bounce Rate 23
Recent Activity
New user registered
Sarah Johnson joined the platform
2 minutes ago
New order received
Order #12847 worth $299.99
5 minutes ago
Report generated
Monthly sales report is ready
15 minutes ago
System notification
Server maintenance scheduled
1 hour ago
View all activities
Quick Actions
New Project
Add User
Generate Report
Export Data
Import Data
Settings

Click & Munch

Busca secciones: productos, usuarios, órdenes...

- Dashboard
- Productos
- Usuarios
- Ordenes
- Reservas
- Restaurantes
- Reportes
- Ratings
- Notificaciones
- Ajustes
- Ayuda

John Doe
john@company.com

Productos

Catálogo con estado de publicación, inventario y categoría.

ID	Imagen	Nombre	Precio	Inventario	Categoría	Estado	Acciones
PRD-001		Latte Vainilla	$4.50	120	Bebidas	Publicado	Editar
PRD-002		Ensalada Mediterránea	$9.80	45	Ensaladas	Pendiente	Editar
PRD-003		Burger Doble	$12.00	30	Platos fuertes	Publicado	Editar
PRD-004		Cheesecake Frutos Rojos	$6.40	25	Postres	Borrador	Editar

Gestión de productos listos para aprobación y publicación.

Click & Munch

Busca secciones: productos, usuarios, órdenes...

- Dashboard
- Productos
- Usuarios
- Ordenes
- Reservas
- Restaurantes
- Reportes
- Ratings
- Notificaciones
- Ajustes
- Ayuda

John Doe
john@company.com

Usuarios

Gestión de cuentas y roles (clientes, administradores de restaurante, meseros, chefs y administradores del sistema).

Total	Activos	Pendientes	Suspendidos
6	4	1	1

1 solicitudes de administradores de restaurante requieren aprobación.

Buscar por nombre, correo o restaurante... | Todos los roles | Todos los estados

ID	Nombre	Correo	Rol	Restaurante	Estado	Creado	Acciones
U-1001	Sarah Johnson Hace 2 días	sarah.johnson@bistro.com	Administrador restaurante	Urban Bistro	Pendiente Requiere aprobación	2025-10-18	Aprobar Suspender
U-1002	Carlos Mendez Hace 3 horas	carlos.mendez@cafeandino.com	Administrador restaurante	Café Andino	Activo	2025-09-12	Suspender Marcar pendiente
U-2001	Ana Ríos Hace 10 minutos	ana.rios@client.com	Cliente	—	Activo	2025-09-25	Suspender Marcar pendiente
U-3001	Luis Ortega Hace 30 minutos	luis.ortega@chefhouse.com	Chef	Urban Bistro	Activo	2025-08-30	Suspender Marcar pendiente
U-4001	Marta Díaz Hace 1 mes	marta.diaz@service.com	Mesero	Café Andino	Suspendido	2025-07-02	Reactivar
U-9001	Admin Principal Hace 5 minutos	admin@clickmunch.com	Administrador del sistema	—	Activo	2025-05-01	Suspender Marcar pendiente

Gestión de usuarios y aprobaciones.

Click & Munch

Busca secciones: productos, usuarios, órdenes...

- Dashboard
- Productos
- Usuarios
- Ordenes
- Reservas
- Restaurantes
- Reportes
- Ratings
- Notificaciones
- Ajustes
- Ayuda

John Doe
john@company.com

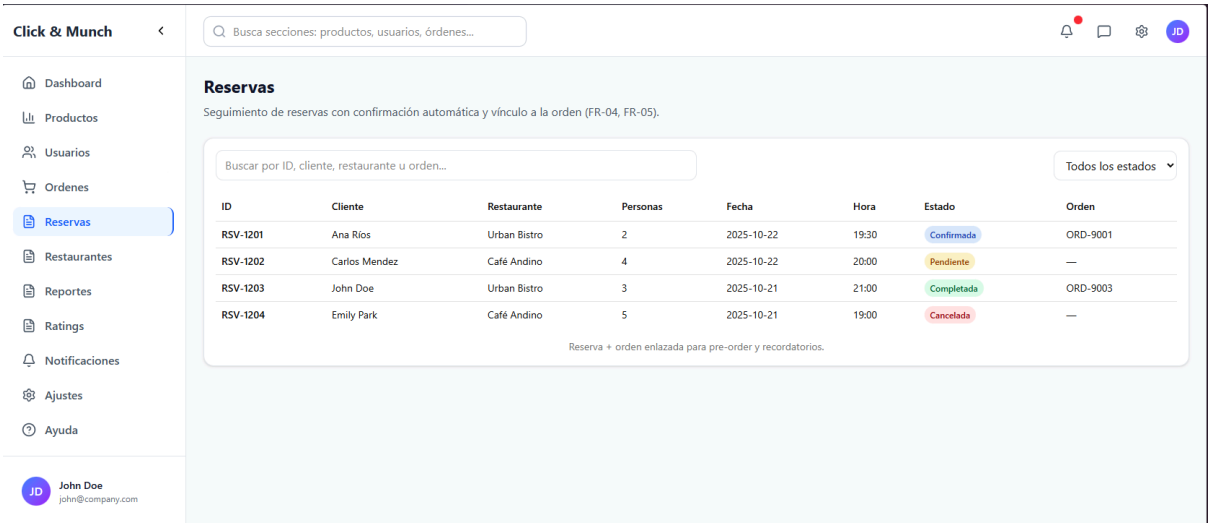
Órdenes

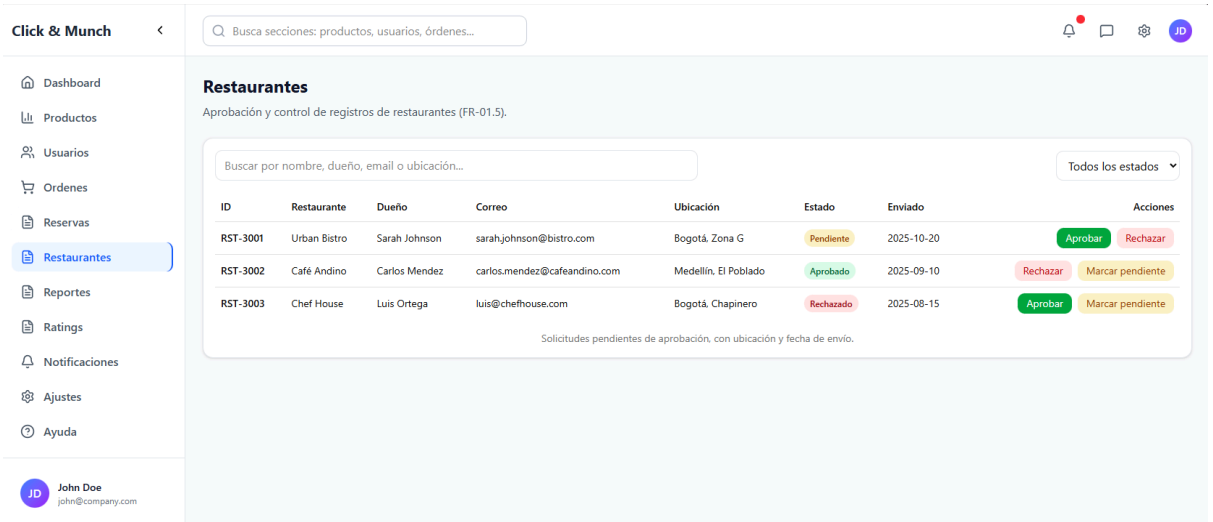
Control de órdenes ligadas a reservas (requerimiento: vincular pedido a la reserva y permitir seguimiento de estado).

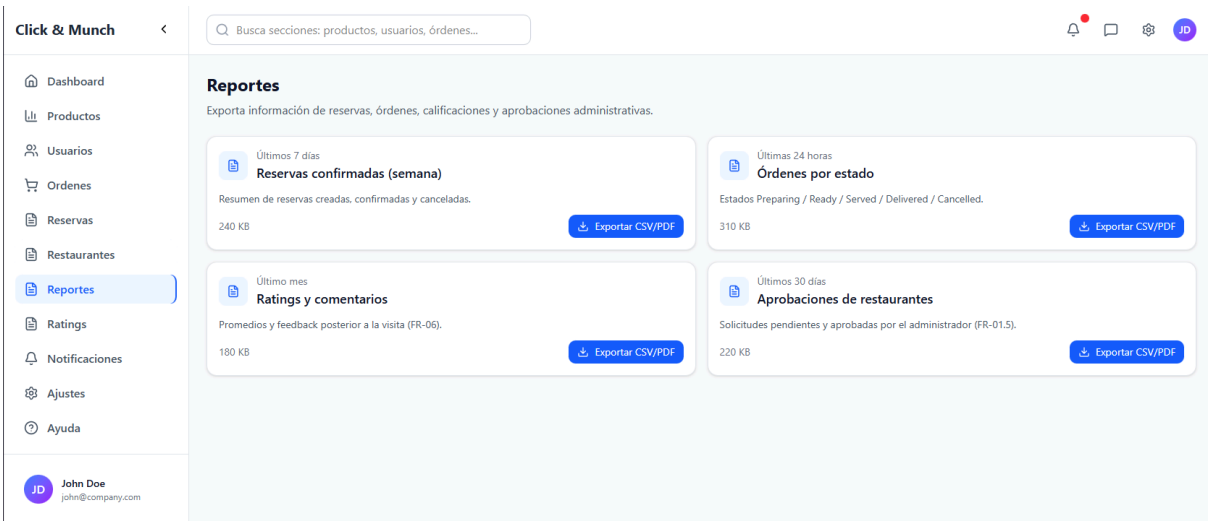
Buscar por ID, cliente o restaurante... | Todos los estados | Todos los canales

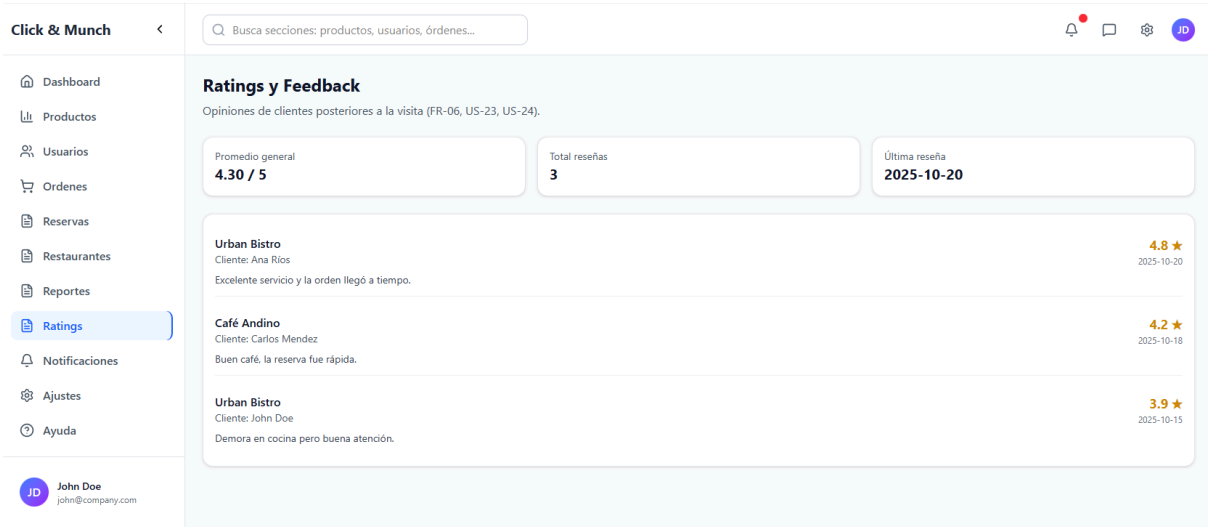
ID	Cliente	Restaurante	Canal	ETA	Estado	Total	Creado
ORD-9001	Ana Ríos	Urban Bistro	Reservation	12 min	Preparing	$48.20	2025-10-21 12:05
ORD-9002	Carlos Mendez	Café Andino	Reservation	Listo	Ready	$28.50	2025-10-21 12:10
ORD-9003	John Doe	Urban Bistro	In-person	Entregado	Delivered	$16.90	2025-10-21 11:40
ORD-9004	Emily Park	Café Andino	Reservation	Servido	Served	$54.00	2025-10-21 11:55
ORD-9005	Miguel Soto	Urban Bistro	Reservation	Cancelado	Cancelled	$0.00	2025-10-21 11:20

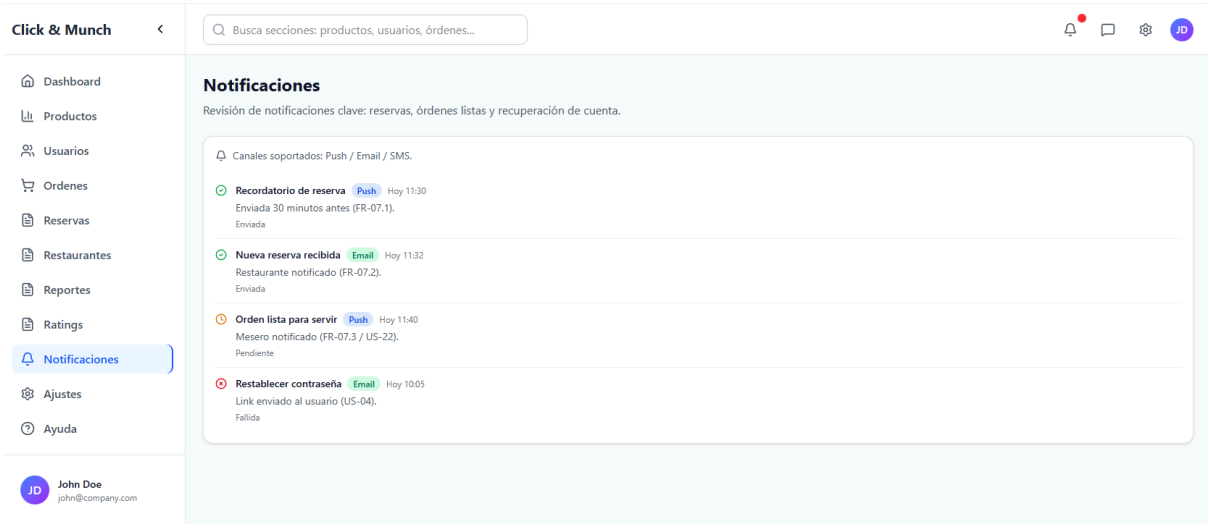
Seguimiento de órdenes y estados de cocina.

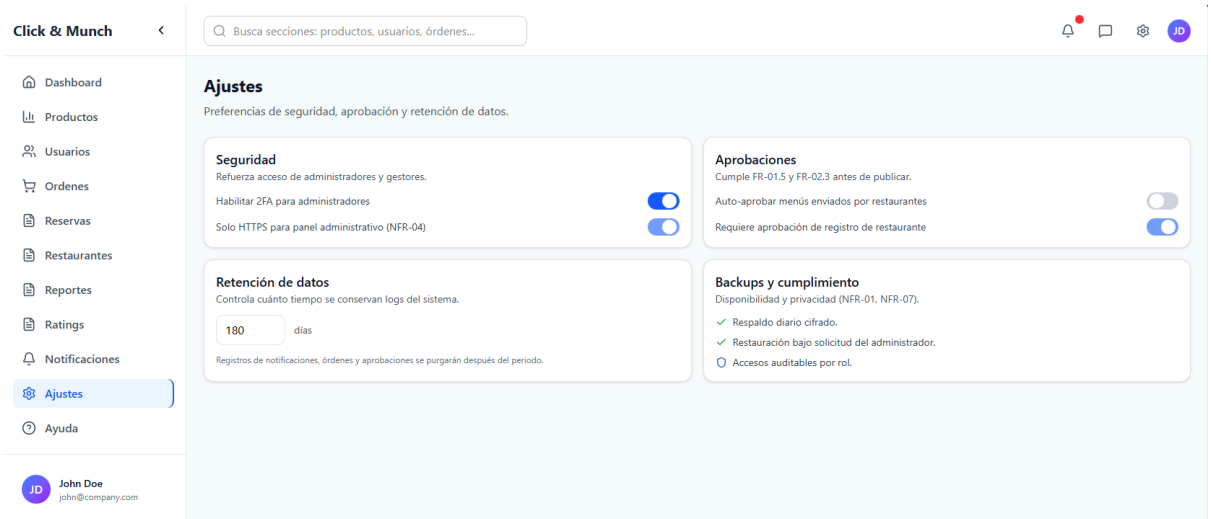

Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Reservas
Seguimiento de reservas con confirmación automática y vínculo a la orden (FR-04, FR-05).
Buscar por ID, cliente, restaurante u orden...
Todos los estados
ID
Cliente
Restaurante
Personas
Fecha
Hora
Estado
Orden
RSV-1201
Ana Ríos
Urban Bistro
2
2025-10-22
19:30
Confirmada
ORD-9001
RSV-1202
Carlos Mendez
Café Andino
4
2025-10-22
20:00
Pendiente
—
RSV-1203
John Doe
Urban Bistro
3
2025-10-21
21:00
Completada
ORD-9003
RSV-1204
Emily Park
Café Andino
5
2025-10-21
19:00
Cancelada
—
Reserva + orden enlazada para pre-order y recordatorios.

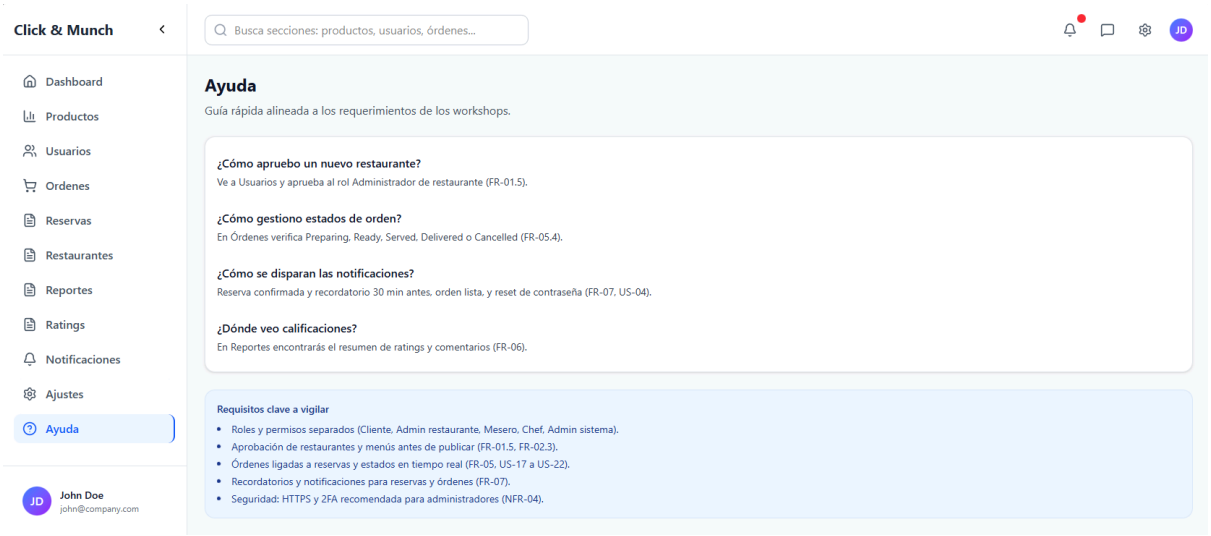

Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Restaurantes
Aprobación y control de registros de restaurantes (FR-01.5).
Buscar por nombre, dueño, email o ubicación...
Todos los estados
ID
Restaurante
Dueño
Correo
Ubicación
Estado
Enviado
Acciones
RST-3001
Urban Bistro
Sarah Johnson
sarah.johnson@bistro.com
Bogotá, Zona G
Pendiente
2025-10-20
Aprobar
Rechazar
RST-3002
Café Andino
Carlos Mendez
carlos.mendez@cafeandino.com
Medellín, El Poblado
Aprobado
2025-09-10
Rechazar
Marcar pendiente
RST-3003
Chef House
Luis Ortega
luis@chefhouse.com
Bogotá, Chapinero
Rechazado
2025-08-15
Aprobar
Marcar pendiente
Solicitudes pendientes de aprobación, con ubicación y fecha de envío.


Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Reportes
Exporta información de reservas, órdenes, calificaciones y aprobaciones administrativas.
Últimos 7 días
Reservas confirmadas (semana)
Resumen de reservas creadas, confirmadas y canceladas.
240 KB
Exportar CSV/PDF
Últimas 24 horas
Órdenes por estado
Estados Preparing / Ready / Served / Delivered / Cancelled.
310 KB
Exportar CSV/PDF
Último mes
Ratings y comentarios
Promedios y feedback posterior a la visita (FR-06).
180 KB
Exportar CSV/PDF
Últimos 30 días
Aprobaciones de restaurantes
Solicitudes pendientes y aprobadas por el administrador (FR-01.5).
220 KB
Exportar CSV/PDF


Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Ratings y Feedback
Opiniones de clientes posteriores a la visita (FR-06, US-23, US-24).
Promedio general
4.30 / 5
Total reseñas
3
Última reseña
2025-10-20
Urban Bistro
Cliente: Ana Ríos
4.8 ★
2025-10-20
Excelente servicio y la orden llegó a tiempo.
Café Andino
Cliente: Carlos Mendez
4.2 ★
2025-10-18
Buen café, la reserva fue rápida.
Urban Bistro
Cliente: John Doe
3.9 ★
2025-10-15
Demora en cocina pero buena atención.


Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Notificaciones
Revisión de notificaciones clave: reservas, órdenes listas y recuperación de cuenta.
Canales soportados: Push / Email / SMS.
Recordatorio de reserva
Push
Hoy 11:30
Enviada 30 minutos antes (FR-07.1).
Enviada
Nueva reserva recibida
Email
Hoy 11:32
Restaurante notificado (FR-07.2).
Enviada
Orden lista para servir
Push
Hoy 11:40
Mesero notificado (FR-07.3 / US-22).
Pendiente
Restablecer contraseña
Email
Hoy 10:05
Link enviado al usuario (US-04).
Fallida

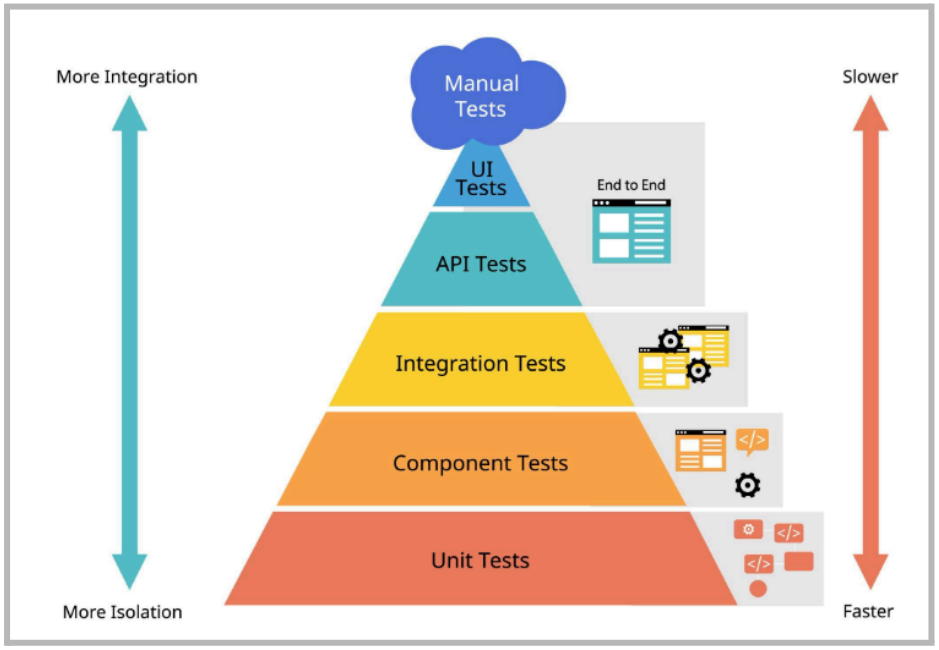

Click & Munch
Busca secciones: productos, usuarios, órdenes...
JD
Dashboard
Productos
Usuarios
Ordenes
Reservas
Restaurantes
Reportes
Ratings
Notificaciones
Ajustes
Ayuda
John Doe
john@company.com
Ajustes
Preferencias de seguridad, aprobación y retención de datos.
Seguridad
Refuerza acceso de administradores y gestores.
Habilitar 2FA para administradores
Solo HTTPS para panel administrativo (NFR-04)
Aprobaciones
Cumple FR-01.5 y FR-02.3 antes de publicar.
Auto-aprobar menús enviados por restaurantes
Requiere aprobación de registro de restaurante
Retención de datos
Controla cuánto tiempo se conservan logs del sistema.
180
días
Registros de notificaciones, órdenes y aprobaciones se purgarán después del periodo.
Backups y cumplimiento
Disponibilidad y privacidad (NFR-01, NFR-07).
Respaldo diario cifrado.
Restauración bajo solicitud del administrador.
Accesos auditables por rol.



5. Testing

During the development of the MVP, the testing strategies for this application include from more isolated yet faster strategies like Unit Testing, to more integrated yet slower strategies such as Integration Testing and Acceptance Testing.

Other strategies like Black Box testing, White Box Testing or API Tests have been conducted during the development cycles when they have been necessary, such as at the moment of implementing new services, modules, features and endpoints.



Test Automation Pyramid model.

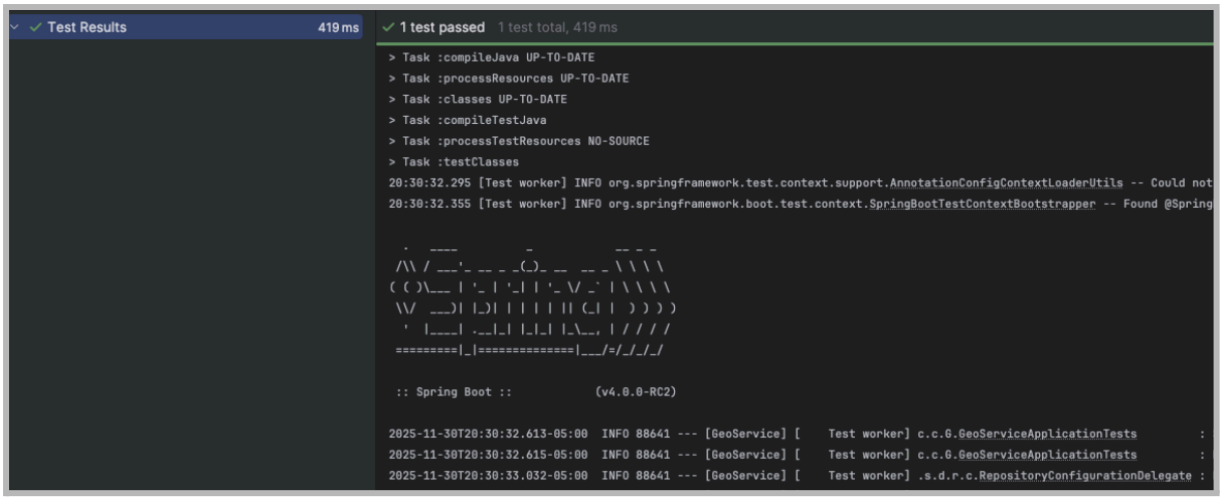
5.1 Unit Testing

To conduct unit testing in the Backend component of the Click & Munch application, the Mockito library has been used, as it is a proven-to-work library for such tests. Unit tests have been conducted on each endpoint as per requirements given by the project leader to guarantee the functionality of every single endpoint.
The available microservices have been developed in Java using SpringBoot. As such, unit tests are built in the project using framework decorators for testing, including the Mockito library. Tests were created with the goal of verifying special conditions and REST API calls.

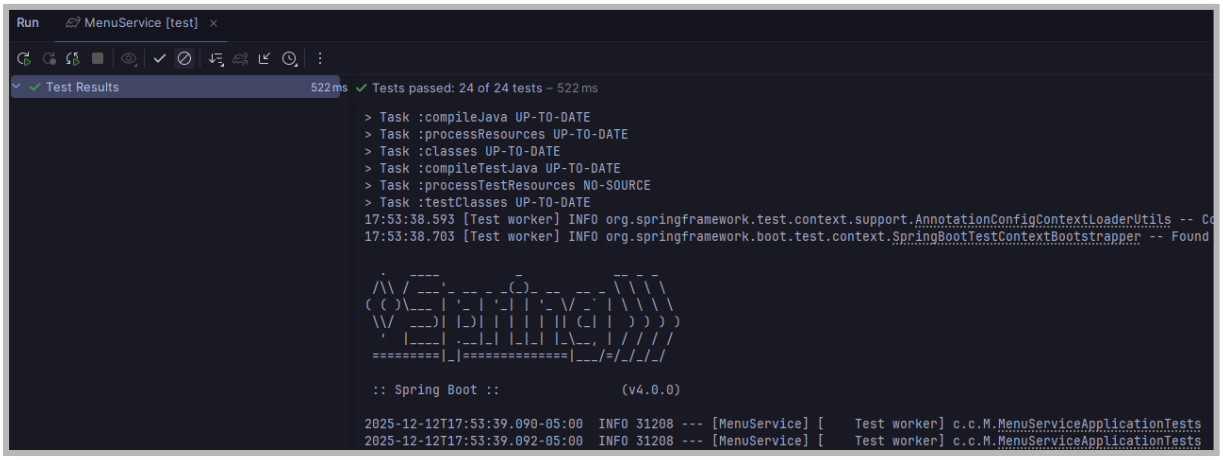
For both applications in the Frontend component, Jest has been used as the framework for the testing, alongside `@testing-library/react-native` to validate component rendering and behavior, as well as user interaction. For the Expo-based project, `jest-expo` was also integrated to ensure full compatibility with Expo modules and native features. This setup allowed for early issue detection, to maintain UI consistency and ensure reliable component behavior across both Frontend applications.

5.1.1 Backend Component

AuthService



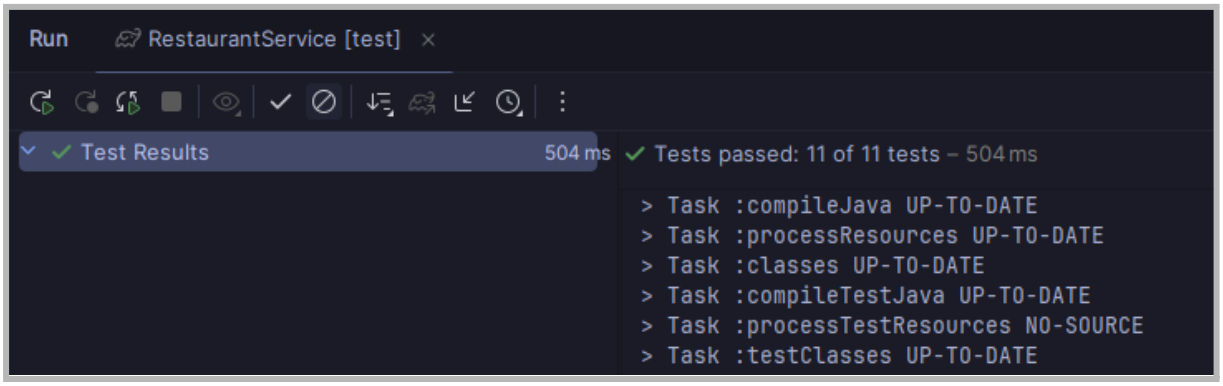
MenuService



The MenuService Unit Tests cover the following Test Cases by method:

Method	**Test Cases**
`getMenuByRestaurantId`	With items, empty categories, null categories.
`deleteMenuByRestaurantId`	With categories, without categories.
`createMenuCategory`	Basic creation.
`createMenuItem`	Basic creation.
`createFullMenu`	Full menu, null categories, empty categories, null restaurantId.
`findMenuCategoryById`	Found, not found.
`findMenuItemById`	Found, not found.
`findMenuItemsByRestaurantId`	With items, no categories.
`updateMenuCategory`	Success, not found.
`updateMenuItem`	All fields, partial update with nulls.
`deleteMenuCategory`	Basic deletion.
`deleteMenuItem`	Basic deletion.

RestaurantService



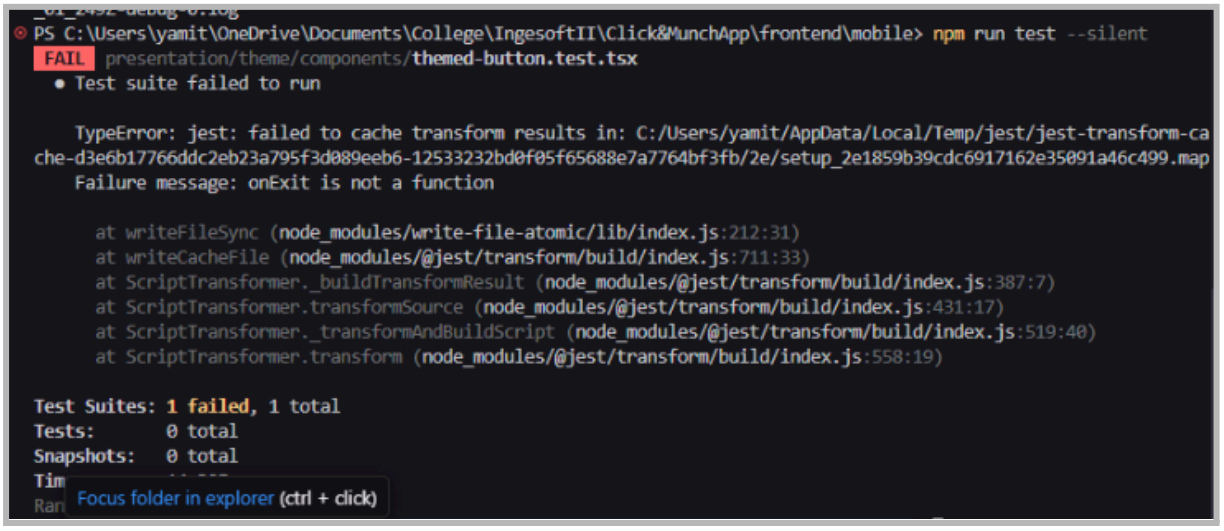
The RestaurantService Unit Tests cover the following Test Cases by method:

Method	**Test Cases**
`createRestaurant(CreateRestaurantRequest request)`	Valid owner with RESTAURANT_MANAGER role creates restaurant successfully.
	Null owner from AuthClient throws RuntimeException.
	User with CUSTOMER role throws HttpClientErrorException.
`getRestaurant(Long id)`	Valid ID returns RestaurantResponse.
	Non-existent ID throws RuntimeException.
`listByOwnerId(Long ownerId)`	Owner with restaurants returns populated list.
	Owner with no restaurants returns empty list.
`findNearby(Double latitude, Double longitude, Double radiusInKm)`	Nearby locations found returns matching restaurants.
	No nearby locations returns empty list without querying repository.
`getRestaurantDetails(Long id)`	Valid ID returns complete details with menu and location.
	Non-existent ID throws RuntimeException.

5.1.2 Frontend Component

`themed-button.test.tsx:`

```tsx
import React from 'react';
import { render } from '@testing-library/react-native';
import { test, expect } from '@jest/globals';
import ThemedButton from './themed-button';

test('renders label', () => {
  const { getByText } = render(<ThemedButton>Tap</ThemedButton>);
  expect(getByText('Tap')).toBeTruthy();
});
```



5.2 Integration Testing

Once the validation of the unit tests was completed, the integration testing was conducted. In this phase, the backend microservices and frontend controllers were tested to work together and ensure that all the requests made were valid.

5.2.1 Integration Strategy

Prior to continuing with the deployment, the Frontend and Backend components were executed in single machines. Tests were conducted and passed as per team metrics.

5.2.2 Key Testing Snippets

Arguably the main and most important tests to validate integration are the tests of communication between services. In this case, several test snippets were developed to make them work only with the Backend services, however, Frontend against Backend cases are more interesting to see.

When integrating Backend and Frontend, the “Find Nearby Restaurants” request was sent using a GET operation:
`GET localhost:8082/api/restaurants/nearby`:
The body must have the following components:

```JSON
{
	"latitude": 52.75214,
	"longitude": -72.886547,
	"radiusInKm": 5
 }
```

And the response in the integration tests was:

```JSON
[
	{
		"id": 1,
		"name": "Burger Station"
		"description": "Best burgers in town"
		"phone": "300123456"
		"email": "contact@burger.com"
		"locationId": 2
	},
	{
		"id": 2,
		"name": "Burger Station"
		"description": "Best burgers in town"
		"phone": "300123456"
		"email": "contact@burger.com"
		"locationId": 3
	},
	{
		"id": 3,
		"name": "Burger Station"
		"description": "Best burgers in town"
		"phone": "300123456"
		"email": "contact@burger.com"
		"locationId": 4
	},
	{
		"id": 4,
		"name": "Burger Station"
		"description": "Best burgers in town"
		"phone": "300123456"
		"email": "contact@burger.com"
		"locationId": 5
	},
	{
```

```
                "id": 5,
                "name": "Burger Station"
                "description": "Best burgers in town"
                "phone": "300123456"
                "email": "contact@burger.com"
                "locationId": 6
        },
        {
                "id": 6,
                "name": "Burger Station"
                "description": "Best burgers in town"
                "phone": "300123456"
                "email": "contact@burger.com"
                "locationId": 7
        },
        {
                "id": 7,
                "name": "Burger Station"
                "description": "Best burgers in town"
                "phone": "300123456"
                "email": "contact@burger.com"
                "locationId": 8
        },
        {
                "id": 8,
                "name": "Burger Station"
                "description": "Best burgers in town"
                "phone": "300123456"
                "email": "contact@burger.com"
                "locationId": 9
        }
]
```


5.3 Acceptance Testing

The following are some of the key user stories that were tested.

User Story 1:
US-01: As a Customer, I want to register and log in so I can use the platform.
The acceptance criteria here was based on the following:

- Given the registration screen, When I enter valid details, Then my account is created.
- Given the login screen, When I enter valid credentials, Then I am authenticated.
- Given invalid credentials, When I try to log in, Then I receive an error message.

User Story 2:
US-02: As a Restaurant Manager, I want to create and manage my restaurant

account.
The acceptance criteria here was based on the following:

- Given the registration form, When I submit valid business data, Then the account is created and pending approval.
- Given an approved restaurant account, When I update information, Then it is saved.

6. Deployment and CI/CD

6.1 Deployment Configuration

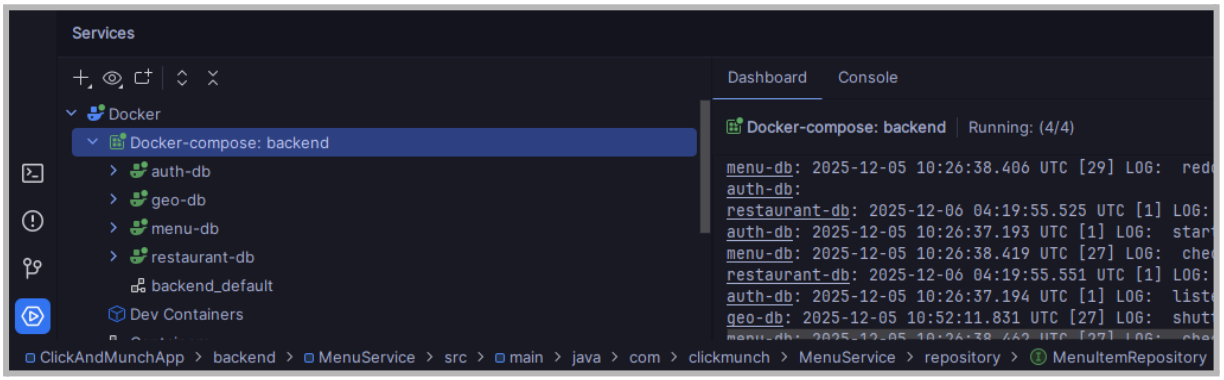
6.1.1 Dockerfiles

As mentioned in a previous section, **4.4. Database Implementation**, the databases for each microservice within the Click & Munch application are created simultaneously. This execution is orchestrated by a dedicated dockerfile:

```JSON
version: '3.8'

services:
  auth-db:
    image: postgres:16
    container_name: auth-db
    restart: always
    environment:
      POSTGRES_DB: auth_db
      POSTGRES_USER: mike
      POSTGRES_PASSWORD: secret
    ports:
      - "5433:5432"
    volumes:
      - auth_data:/var/lib/postgresql/data

  restaurant-db:
    image: postgres:16
    container_name: restaurant-db
    restart: always
    environment:
      POSTGRES_DB: restaurant_db
      POSTGRES_USER: mike
      POSTGRES_PASSWORD: secret
    ports:
      - "5434:5432"
    volumes:
      - restaurant_data:/var/lib/postgresql/data

  geo-db:
```

```
    image: postgis/postgis:16-3.4
    container_name: geo-db
    restart: always
    environment:
      POSTGRES_DB: geo_db
      POSTGRES_USER: mike
      POSTGRES_PASSWORD: secret
    ports:
      - "5435:5432"
    volumes:
      - geo_data:/var/lib/postgresql/data

  menu-db:
    image: postgres:16
    container_name: menu-db
    restart: always
    environment:
      POSTGRES_DB: menu_db
      POSTGRES_USER: mike
      POSTGRES_PASSWORD: secret
    ports:
      - "5436:5432"
    volumes:
      - menu_data:/var/lib/postgresql/data

volumes:
  auth_data:
  restaurant_data:
  geo_data:
  menu_data:
```

The `docker-compose.yml` dockerfile composes the entire database on a per-microservice basis, by using a single host but several microservices.



*Docker Compose Stack "**`backend`**" running the containers for the different microservices. Screenshot from the Services tab inside the IntelliJ IDEA IDE.*

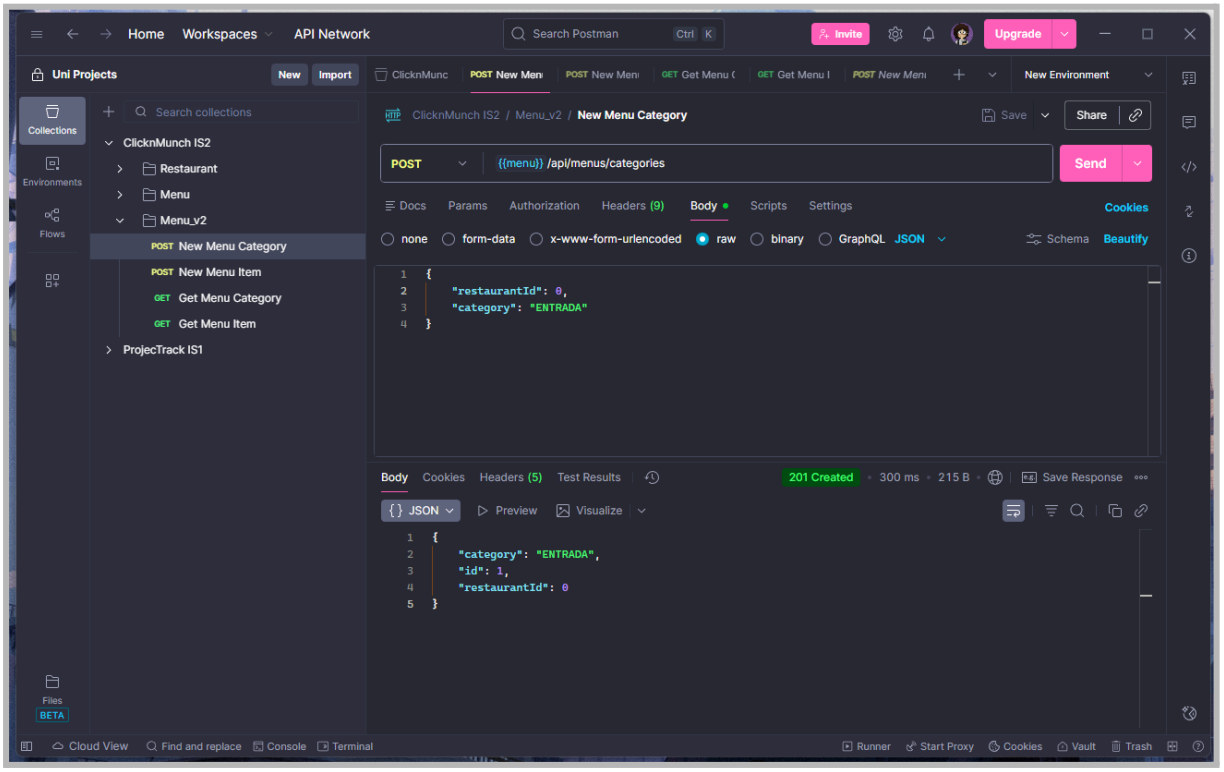
For the cloud deployment, the following `docker-compose.services.yml` dockerfile is provided:

JSON

```
version: '3.8'
services:
  apigateway:
    build: ./APIGateway
    container_name: apigateway
    ports:
      - "8080:8080"
    environment:
      - JAVA_OPTS=
    depends_on:
      authservice:
        condition: service_healthy
      restaurantservice:
        condition: service_healthy
      menuservice:
        condition: service_healthy
    networks:
      - appnet

  authservice:
    build: ./AuthService
    container_name: authservice
    ports:
      - "8081:8081"
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://auth-db:5432/auth_db
      SPRING_DATASOURCE_USERNAME: mike
      SPRING_DATASOURCE_PASSWORD: secret
    depends_on:
      auth-db:
        condition: service_healthy
    networks:
      - appnet

  restaurantservice:
    build: ./RestaurantService
    container_name: restaurantservice
    ports:
      - "8082:8082"
    environment:
                                                    SPRING_DATASOURCE_URL:
jdbc:postgresql://restaurant-db:5432/restaurant_db
      SPRING_DATASOURCE_USERNAME: mike
      SPRING_DATASOURCE_PASSWORD: secret
      AUTH_SERVICE_URL: http://authservice:8081
      GEO_SERVICE_URL: http://geoservice:8083
    depends_on:
      restaurant-db:
```

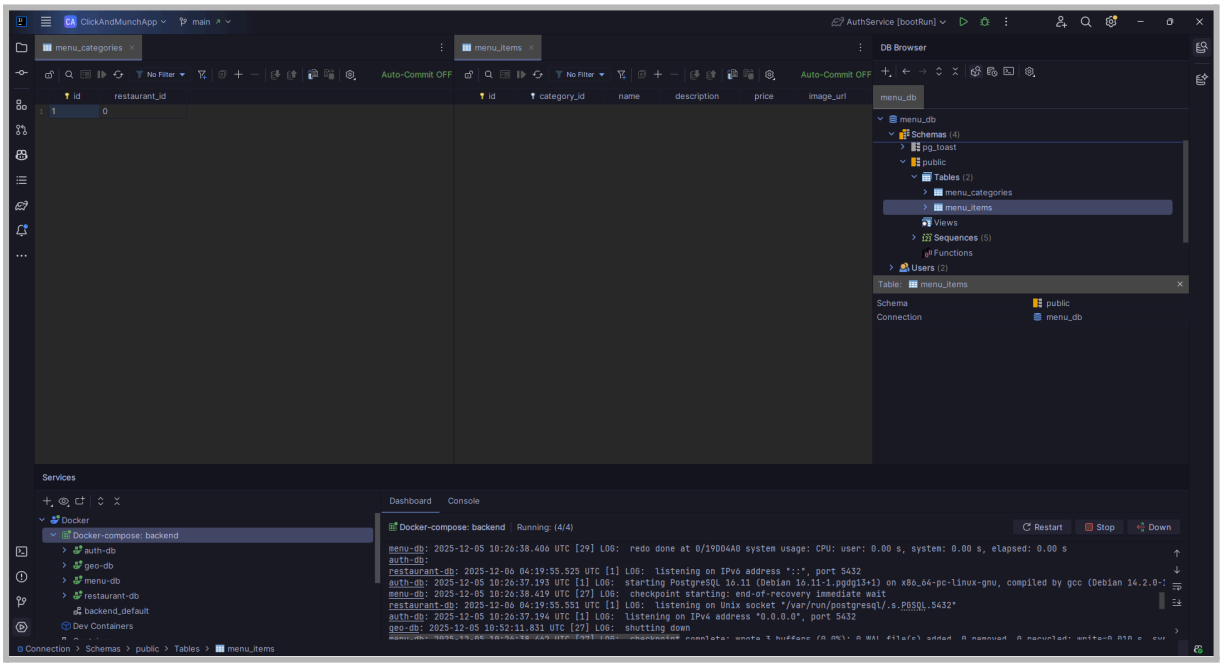
```
        condition: service_healthy
      authservice:
        condition: service_healthy
      geoservice:
        condition: service_healthy
    networks:
      - appnet

  geoservice:
    build: ./GeoService
    container_name: geoservice
    ports:
      - "8083:8083"
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://geo-db:5432/geo_db
      SPRING_DATASOURCE_USERNAME: mike
      SPRING_DATASOURCE_PASSWORD: secret
    depends_on:
      geo-db:
        condition: service_healthy
    networks:
      - appnet

  menuservice:
    build: ./MenuService
    container_name: menuservice
    ports:
      - "8084:8084"
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://menu-db:5432/menu_db
      SPRING_DATASOURCE_USERNAME: mike
      SPRING_DATASOURCE_PASSWORD: secret
      RESTAURANT_SERVICE_URL: http://restaurantservice:8082
    depends_on:
      menu-db:
        condition: service_healthy
      restaurantservice:
        condition: service_healthy
    networks:
      - appnet

networks:
  appnet:
    driver: bridge
```

The correct functioning and deployment of the databases for the different microservices were tested via the microservices controllers using API platform software such as Postman, to try the different path operations (GET, POST, PUT, DELETE) that result in database queries.



POST query to the MenuService database. Screenshot from Postman.

The database changes, then could be checked using database management software such as pgAdmin 4 or any IDE extension that manages database connections (e.g. Database Navigator for IntelliJ IDEA).



Database Navigator extension for IntelliJ IDEA showing the connection view and the data view for the MenuService tables. At that given moment, both tables were nearly empty.

6.1.2 Azure Environment Setup

The following is the configuration used for the Azure VM used:

VM Specification:

- Azure B1s
- Ubuntu 20.04 LTS

Installed Prerequisites:

- Docker
- Docker Compose
- Git

Port Configuration:

Ports were opened in Azure Network Security Group: 80, 443, 8081, 8082, 8083, 8084.

6.2 CI/CD Pipeline implementation

DevOps lifecycle.

6.2.1 Pipeline Workflow

The tools that were leveraged for a CI/CD pipeline implementation were GitHub Actions, using an Azure Linux VM as a host. The workflow is mentioned within the README.md files located inside the repository, with the instructions to deploy the application and manage microservices.

The following is the basic workflow:

1. **Build:** GitHub Actions builds Docker images for all the services.
2. **Test:** Runs unit tests inside the containers.

3. **Push:** Pushes images to Docker Hub.
4. **Deploy:** SSH into Azure VM, pull new images, restart containers.

The CI/CD workflow is defined in the following YML files, within the `.github/workflows/` directory:

```
JSON
name: Build and Push Backend Images to ECR

on:
 push:
   branches: [ main ]
   paths:
     - 'ClickAndMunchApp/backend/**'
 workflow_dispatch:

jobs:
 build-and-push:
   runs-on: ubuntu-latest
   env:
     AWS_REGION: ${{ secrets.AWS_REGION }}
     ECR_REGISTRY: ${{ secrets.ECR_REGISTRY }} # e.g.,
123456789012.dkr.ecr.us-east-1.amazonaws.com
   strategy:
     matrix:
       include:
         - service: APIGateway
           repo: clickandmunchapp/api-gateway
         - service: AuthService
           repo: clickandmunchapp/auth
         - service: RestaurantService
           repo: clickandmunchapp/restaurant
         - service: GeoService
           repo: clickandmunchapp/geo
         - service: MenuService
           repo: clickandmunchapp/menu
   steps:
     - name: Checkout
       uses: actions/checkout@v4

     - name: Configure AWS credentials
       uses: aws-actions/configure-aws-credentials@v4
       with:
         aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
         aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
         aws-region: ${{ env.AWS_REGION }}

     - name: Verify AWS identity and ECR access
       run: |
         aws sts get-caller-identity
         aws ecr describe-registry --region $AWS_REGION
         aws ecr describe-repositories --region $AWS_REGION || true
```

```
      - name: Login to Amazon ECR (CLI)
        id: login-ecr
        env:
          AWS_REGION: ${{ env.AWS_REGION }}
          ECR_REGISTRY: ${{ env.ECR_REGISTRY }}
        run: |
          aws ecr get-login-password --region "$AWS_REGION" | docker login --username
AWS --password-stdin "$ECR_REGISTRY"

      - name: Set image name
        id: vars
        run: |
          IMAGE_NAME=${{ env.ECR_REGISTRY }}/${{ matrix.repo }}:$(echo "${GITHUB_SHA}"
| cut -c1-7)
          echo "IMAGE_NAME=$IMAGE_NAME" >> $GITHUB_OUTPUT

      - name: Build Docker image
        working-directory: ClickAndMunchApp/backend/${{ matrix.service }}
        run: |
          docker build -t ${{ steps.vars.outputs.IMAGE_NAME }} .

      - name: Push Docker image
        run: |
          docker push ${{ steps.vars.outputs.IMAGE_NAME }}

      - name: Output image tag
        run: echo "Pushed ${{ steps.vars.outputs.IMAGE_NAME }}"
```

JSON

```
name: Build Dashboard

on:
  push:
    branches: [ main ]
    paths:
      - 'ClickAndMunchApp/frontend/dashboard/**'
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: '20'
```

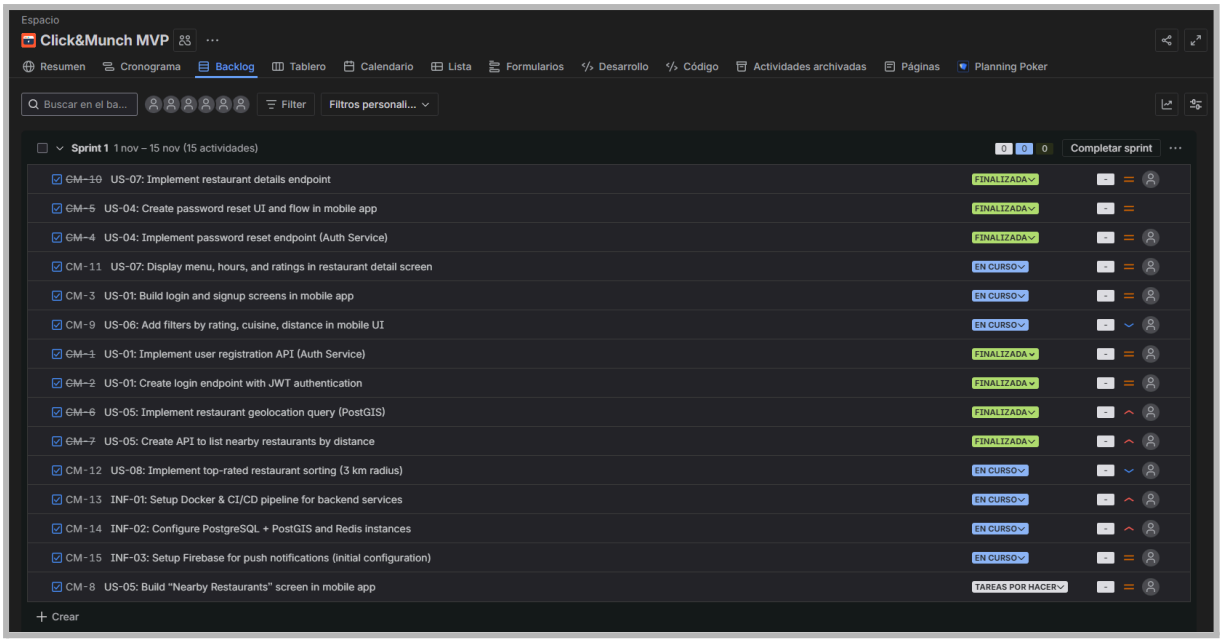
```
        cache: 'npm'
        cache-dependency-path: ClickAndMunchApp/frontend/dashboard/package.json

    - name: Install deps
      working-directory: ClickAndMunchApp/frontend/dashboard
      run: npm ci

    - name: Build
      working-directory: ClickAndMunchApp/frontend/dashboard
      run: npm run build

    - name: Upload artifact
      uses: actions/upload-artifact@v4
      with:
        name: dashboard-dist
        path: ClickAndMunchApp/frontend/dashboard/dist
```

7. Scrum

7.1 Product Backlog + Sprint Backlogs

The product backlog for a number of sprints was initially defined with the elaboration of the User Story Map (USM) described in **Section 1.4 User Story Mapping (USM)** in the earlier stages of development. The initial estimation made using the USM covers functionalities expected to be implemented beyond a MVP delivery for the scope of the course.

Once the stories were defined, they were registered into a backlog and distributed into five projected Sprints Backlogs using Jira. This MVP delivery for this course's project consists, for the most part, of the tasks defined in the first two sprints of the Jira space.



Jira screenshot, showing backlog for the Sprint 1.

7.2 Scrum Ceremonies

7.2.1 Sprint

As mentioned before, the Sprint Backlog, and as such, the Sprints themselves were defined at the beginning of the development, when building the UMA. According to the needs, and possible issues of the team– mainly related to other university and/or work duties, the secondary goals and backlog were refined.

7.2.2 Sprint Planning

For Sprint 1 and Sprint 2– the MVP sprints, meetings between the developer team were conducted, with the objective of estimating the number of story points for each task, and assigning them to a developer. Posterior meetings were conducted for adjustments, and completion of the ceremonies, given unexpected logistic issues to coordinate the first meetings.

7.2.3 Daily Sprint

No Daily Sprint ceremonies were conducted, as the developers' university and/or work responsibilities made the coordination of this Scrum artifact difficult.

7.2.4 Spring Review and Spring Retrospective

No live meetings for these Scrum artifacts were conducted, given the increasing difficulties to coordinate them, as the semester went by. Further reviews and adjustments for the development cycle were discussed within the development team via

text. Features were implemented, tested, pushed and merged by them in an incremental manner, regardless of the coordination issues.

7.3 Scrum Team – Roles

The main three roles of a Scrum Team were fulfilled by the development team. The main distinction between the roles were within the **Developers**, as they were divided into Frontend and Backend groups.
Product Owner and **Scrum Master** roles were fulfilled "on the go" by the team members.

7.4 Difficulties

As mentioned above, various logistical, personal and coordination issues made it difficult to fulfill all the different Scrum artifacts and ceremonies. The work was made on an incremental basis as much as possible, despite these problems.

References

- EngAndres. (n.d.). *unal_public* [Folder: Software Engineering 2_Morning (G3)/slides]. GitHub. https://github.com/EngAndres/unal_public/tree/main/Software%20Engineering%202_Morning%20(G3)/slides
- OpenAPI 3 Library for spring-boot By Badr NASS LAHSEN & Library for OpenAPI 3 with spring-boot By Badr NASS LAHSEN. (n.d.). OpenAPI 3 Library for spring-boot. OpenAPI 3 Library for Spring-boot. https://springdoc.org/
- Generating OpenAPI docs for Java with Spring Boot · Bump.sh. (n.d.). Bump.sh. https://bump.sh/blog/generating-openapi-docs-for-java-with-spring-boot/
- Mockito - mockito-core 5.20.0 javadoc. (n.d.). https://javadoc.io/doc/org.mockito/mockito-core/latest/org.mockito/org/mockito/Mockito.html
- Schwaber, K., & Sutherland, J. (2013). La guía de Scrum. Scrumguides. Org, 1, 21.