UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE AUTOMATICĂ, CALCULATOARE ȘI ELECTRONICĂ

DEPARTAMENTUL DE CALCULATOARE ȘI TEHNOLOGIA INFORMAȚIEI

PROIECT DE DIPLOMĂ

Mircea-Denis Brașoveanu

COORDONATOR ȘTIINȚIFIC

Prof. univ. dr. ing. Marian Cristian Mihăescu

Iulie 2025

CRAIOVA

UNIVERSITATEA DIN CRAIOVA
FACULTATEA DE AUTOMATICĂ, CALCULATOARE ȘI ELECTRONICĂ

DEPARTAMENTUL DE CALCULATOARE ŞI TEHNOLOGIA INFORMAȚIEI

Online Grocery Store with AI-Based Recommendations

Mircea-Denis Brașoveanu

COORDONATOR ŞTIINȚIFIC

Prof. univ. dr. ing. Marian Cristian Mihăescu

Iulie 2025

CRAIOVA

„The important thing is not to stop questioning. Curiosity has its own reason for existence.”

Albert Einstein

DECLARAȚIE DE ORIGINALITATE

Subsemnatul Mircea-Denis Brașoveanu student la specializarea Calculatoare (în limba engleză) din cadrul Facultății de Automatică, Calculatoare și Electronică a Universității din Craiova, certific prin prezenta că am luat la cunoştinţă de cele prezentate mai jos şi că îmi asum, în acest context, originalitatea proiectului meu de licenţă:

- cu titlul “Online Grocery Store with AI-Based Recommendations”,
- coordonată de Prof. univ. dr. ing. Marian Cristian Mihăescu,
- prezentată în sesiunea Iulie 2025.

La elaborarea proiectului de licenţă, se consideră plagiat una dintre următoarele acţiuni:

- reproducerea exactă a cuvintelor unui alt autor, dintr-o altă lucrare, în limba română sau prin traducere dintr-o altă limbă, dacă se omit ghilimele şi referinţa precisă,
- redarea cu alte cuvinte, reformularea prin cuvinte proprii sau rezumarea ideilor din alte lucrări, dacă nu se indică sursa bibliografică,
- prezentarea unor date experimentale obţinute sau a unor aplicaţii realizate de alţi autori fără menţionarea corectă a acestor surse,
- însuşirea totală sau parţială a unei lucrări în care regulile de mai sus sunt respectate, dar care are alt autor.

Pentru evitarea acestor situaţii neplăcute se recomandă:

- plasarea între ghilimele a citatelor directe şi indicarea referinţei într-o listă corespunzătoare la sfărşitul lucrării,
- indicarea în text a reformulării unei idei, opinii sau teorii şi corespunzător în lista de referinţe a sursei originale de la care s-a făcut preluarea,
- precizarea sursei de la care s-au preluat date experimentale, descrieri tehnice, figuri, imagini, statistici, tabele et caetera,
- precizarea referinţelor poate fi omisă dacă se folosesc informaţii sau teorii arhicunoscute, a căror paternitate este unanim cunoscută și acceptată.

Data, Semnătura candidatului,

17.06.2025

[illegible]

UNIVERSITATEA DIN CRAIOVA
Facultatea de Automatică, Calculatoare şi Electronică

Departamentul de Calculatoare şi Tehnologia Informației

Aprobat la data de
…………………
Şef de departament,
Prof. dr. ing.
Nicolae Enescu

PROIECTUL DE DIPLOMĂ

Numele și prenumele studentului:	Brașoveanu Mircea-Denis
Enunțul temei:	Online Grocery Store with AI-Based Recommendations
Datele de pornire:	React Native, ASP.NET, TypeScript, EF Core, SQL Server 2019
Conținutul proiectului:	Introduction: Application overview Theoretical context and technologies: Used technologies and concepts Software arhitecture: Arhitecture, Context, Structure Software implementation: Security, Performance, Modern Components Testing: Testing strategy, devices, approach Conclusions: Possible features/improvements Bibliography: Books and websites citations
Material grafic obligatoriu:	Use case diagrams, Database structure, PPT Presentation
Consultații:	Săptămânale
Conducătorul științific (titlul, nume și prenume, semnătura):	Prof. univ. dr. ing. Marian Cristian Mihăescu
Data eliberării temei:	19.07.2024
Termenul estimat de predare a proiectului:	15.06.2025
Data predării proiectului de către student și semnătura acestuia:	15.06.2025 [illegible]

UNIVERSITATEA DIN CRAIOVA
Facultatea de Automatică, Calculatoare şi Electronică

Departamentul de Calculatoare și Tehnologia Informației

REFERATUL CONDUCĂTORULUI ŞTIINȚIFIC

Numele şi prenumele candidatului:	Braşoveanu Mircea-Denis
Specializarea:	Calculatoare cu predare în limba engleză (CE)
Titlul proiectului:	Online Grocery Store with AI-Based Recommendations
Locația în care s-a realizat practica de documentare (se bifează una sau mai multe din opțiunile din dreapta):	În facultate ☐ În producție ☐ În cercetare ☐ Altă locație:

În urma analizei lucrării candidatului au fost constatate următoarele:

Nivelul documentării		Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
Tipul proiectului		Cercetare ☐	Proiectare ☐	Realizare practică ☐	Altul
Aparatul matematic utilizat		Simplu ☐	Mediu ☐	Complex ☐	Absent ☐
Utilitate		Contract de cercetare ☐	Cercetare internă ☐	Utilare ☐	Altul
Redactarea lucrării		Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
Partea grafică, desene		Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
Realizarea practică	Contribuția autorului	Insuficientă ☐	Satisfăcătoare ☐	Mare ☐	Foarte mare ☐
	Complexitatea temei	Simplă ☐	Medie ☐	Mare ☐	Complexă ☐
	Analiza cerințelor	Insuficient ☐	Satisfăcător ☐	Bine ☐	Foarte bine ☐
	Arhitectura	Simplă ☐	Medie ☐	Mare ☐	Complexă ☐

	Întocmirea specificațiilor funcționale	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
	Implementarea	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
	Testarea	Insuficientă ☐	Satisfăcătoare ☐	Bună ☐	Foarte bună ☐
	Funcționarea	Da ☐	Parțială ☐	Nu ☐	
Rezultate experimentale		Experiment propriu ☐		Preluare din bibliografie ☐	
Bibliografie		Cărți	Reviste	Articole	Referințe web
Comentarii și observații					

În concluzie, se propune:

ADMITEREA PROIECTULUI	RESPINGEREA PROIECTULUI
☐	☐

Data,   Semnătura conducătorului științific,

PROJECT SUMMARY

This project focuses on the development of a mobile delivery application built with React Native, offering separate interfaces for all types of users (customers, delivery users and administrators). A key feature of the app is the personalized recommendation system, based on a custom algorithm I created. Its purpose is to suggest relevant options to users in a smart and adaptive way, depending on their preferences and behavior within the app.

The idea for this project came from my passion for programming, algorithms, and learning new technologies, as well as my growing interest in how digital tools support the business world. I wanted to build something practical and up-to-date, using technologies that are widely adopted in the industry today.

Throughout the process, I dealt with several challenges, such as ensuring smooth cross-platform performance, managing data efficiently, and integrating the recommendation system without compromising user experience. These were addressed through a mix of research, trial and error, and gradual improvements.

My personal contribution includes the design and implementation of the recommendation algorithm, building users interfaces, and choosing the right tools and libraries to make the app scalable and efficient. Besides the technical experience, this project taught me a lot about mobile development, structuring real-world apps, and the balance between functionality and usability.

Keywords: React Native, mobile development, smart recommendations, delivery application, algorithm design, user interface, cross-platform, business application, modern technologies.

ACKNOWLEDGEMENTS

I would like to thank my project coordinator, Prof. univ. dr. ing. Marian Cristian Mihăescu, for his support and guidance during this final year. His advice helped me stay on track and complete this project. Beyond this, I also deeply appreciate his continuous encouragement and support for my involvement in competitive programming during the past four years. I'm especially thankful for the knowledge and skills he equipped me with, which directly contributed to my achievements in this challenging field.

I am also grateful to all my teachers from the past four years. Thanks to them, I have learned many important things that I will use in my future career.

CONTENT

FIGURES LIST

TABLES LIST

1 INTRODUCTION

1.1 Scope

This thesis focuses on designing and developing a mobile delivery application with React Native. There are three primary sections to the app: administrator view, delivery user view and customer view. The app's unique feature is its SMART suggestion system (AI based), which was developed using a custom-made algorithm to provide recommendations for users based on the system orders in a specific timeframe.

This project covers every step of creating the application, from organizing the structure, selecting the appropriate technologies, creating the user interfaces, and putting the features into use and testing them. The goal is to design an user-friendly software that follows current mobile development standards.

The application was designed to offer a modality for smaller stores to sell their products using their own dedicated app.

The study also looks at how delivery apps' user experience might be enhanced by clever algorithms. The finished item is a working prototype that can be expanded upon.

1.2 Motivation

I have always been interested in algorithms, programming, and learning new technologies, which is why I picked this topic. I liked working on real-world projects during my education; therefore, I wanted my final project to be practical and applicable.

At the same time, I have become more curious about how technology is used in the business environment, especially in areas like delivery, where smart systems can improve services and make processes more efficient. I wanted to create something that combined this business viewpoint with my technical expertise.

With this project, I was able to use cutting-edge tools like React Native and create a clever recommendation system from the ground up using my own reasoning. It aided in my development and gave me a better understanding of how software can solve real-life problems.

2 THEORETICAL CONTEXT AND TECHNOLOGIES

2.1 Application Logo

The application's logo was carefully designed by me, incorporating the principles of the golden ratio to achieve aesthetic balance and harmony. This was specifically accomplished through the use of golden ratio circles, which guided the proportional relationships and overall composition of the design.

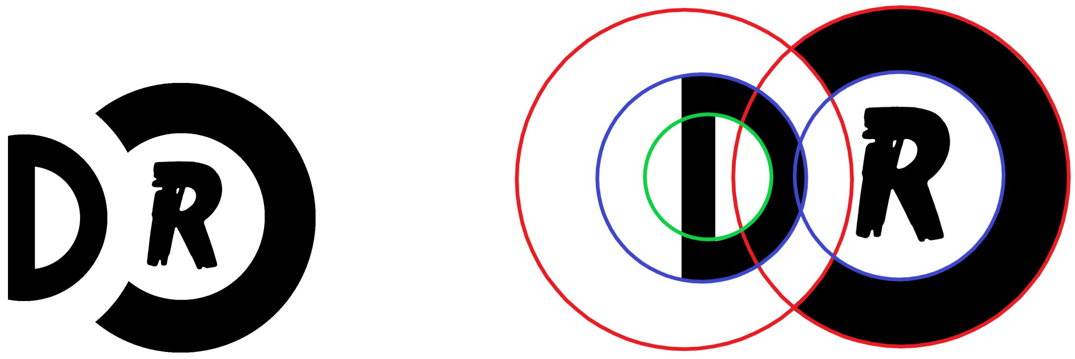


Figure 1: Logo

The circles I used in the logo design maintain a harmonious relationship, precisely adhering to the golden ratio (ϕ≈1.618). The colors were added using the inclusion/exclusion of the circles.

Through its application, the logo gains an intrinsic sense of order and visual appeal, where every element resonates with this celebrated proportion. This principle is widely adopted by numerous companies in the design of their logos.

2.2 React Native

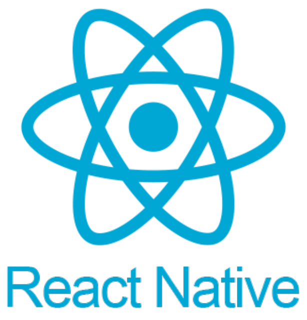


Figure 2: React Native

React and React Native are two different JavaScript libraries used to build user interfaces. React is mainly used for building web applications, while React Native is used for creating mobile apps. Even though React Native is based on React, they are designed for different platforms and have different purposes.

React Native is a popular mobile app framework based on JavaScript. It allows developers to create mobile apps for both iOS and Android using a single codebase. This means users can build apps for different platforms without writing separate code for each one.

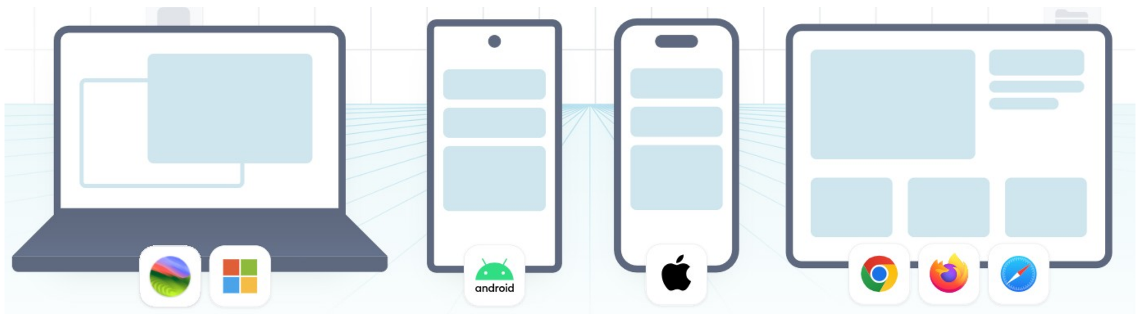


Figure 3: Cross-platform apps with one codebase [2]

“React Native was released as an open-source project by Facebook in 2015. Since then, it has become one of the most popular tools for mobile app development. It is used in many well-known apps, such as Instagram, Facebook, and Skype.” [1]

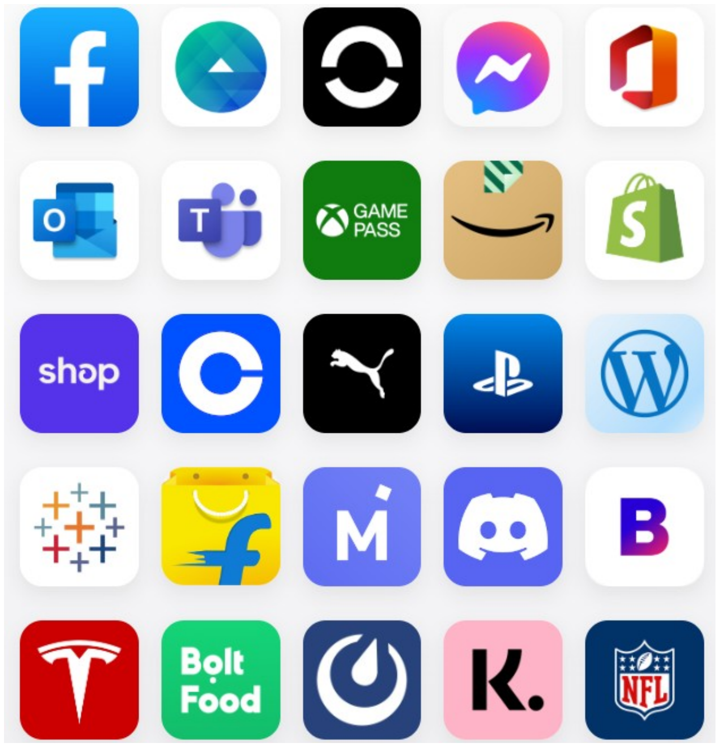


Figure 4: React Native is being used in thousands of apps [2]

"React Native can be used on its own or within a framework that provides a set of tools and APIs for building production-ready applications. Using a framework such as Expo offers features such as file-based routing, access to universal libraries, and support for creating plugins that interact with native code, without the need to directly handle native files." [3]

2.3 Expo

Figure 5: Expo

“Expo is a framework and platform built around React Native that helps with the development, building, and deployment of apps. It allows developers to create iOS, Android, and web applications using the same JavaScript or TypeScript codebase, with tools that support fast development and testing.” [4]

Expo simplifies the development process with pre-configured tools, libraries, and services.

Expo is a good choose because it assures:

1. Fast Setup: Expo simplifies the initial setup process by removing the need to configure native development environments, allowing developers to focus on core app features.
2. Cross-Platform Support: Applications can be developed using a single codebase with JavaScript and React Native and run on multiple platforms.
3. Managed Workflow: Expo handles many technical details behind the scenes, which is especially helpful for developers with limited mobile development experience.
4. Active Community and Resources: Expo benefits from a strong community and a wide range of tools, libraries, and documentation. [5]

2.4 .NET

Figure 6: .NET

“.NET is a free, open-source, and cross-platform framework used to build modern applications and cloud-based services. It supports multiple programming languages, editors, and libraries, allowing developers to build for web, mobile, desktop, games, IoT, and more.” [6]

In this application, .NET was used mainly for the backend, where it helped manage the business logic, user data, and communication with the database.

By using .NET, the backend is able to expose RESTful APIs that allow the React Native mobile app to interact with the server, for example, to register users, process orders, or fetch personalized recommendations.

The main reasons for choosing .NET are:

- Strong support for building secure and efficient web services
- Good integration with databases
- Clean and structured development using C#
- Excellent support and documentation from Microsoft and the developer community

.NET benefits from a vast and mature ecosystem of libraries and tools accessible via NuGet, the official package manager. This enables rapid development by leveraging existing, well-tested solutions for various functionalities, from logging and authentication to data processing and external API integrations, reducing the need to 'reinvent the wheel'.

2.5 Entity Framework

Figure 7: Entity Framework

Entity Framework (EF) is a modern object-relational mapper (ORM) used in .NET applications to interact with databases in a clean and efficient way. It allows developers to work with data using .NET objects, without writing raw SQL queries. [7]

In this project, Entity Framework was used to create the data access layer, making it easier to manage and query the database. It supports features like LINQ queries, change tracking, data updates, and schema migrations. EF works with different types of databases, such as SQL Server, SQLite, MySQL, and PostgreSQL.

Using EF helped speed up development and ensured that the database logic remained organized, readable, and easy to maintain.

Key Features:

- Cross-Platform: Works on Windows, macOS, and Linux.
- LINQ Support: Allows querying the database using Language Integrated Query (LINQ), making code easier to read and maintain.
- Change Tracking: Keeps track of changes made to objects so that updates to the database can be done automatically.
- Migrations: Makes it easy to apply and manage database schema changes over time. [8]

How EF Core Works:

EF Core uses a special class called DbContext, which acts as a connection between the application and the database. The flow includes three main parts:

1. Model: The C# classes (called entities) represent database tables, and their properties represent the table columns.

2. Context: The DbContext class handles tasks like querying data, adding new entries, updating records, or deleting them.

3. Database Provider: EF Core can work with different types of databases (e.g., SQL Server, MySQL, PostgreSQL, SQLite) using specific database providers. [8]

A key aspect of EF Core is its ability to translate LINQ queries into efficient SQL statements. When a LINQ query is executed against a DbSet, EF Core's query provider analyzes the expression tree and generates the appropriate SQL query, which is then sent to the database. This abstraction allows developers to focus on data manipulation using object-oriented constructs rather than writing database-specific SQL.

EF Core supports different development approaches, including Code-First, where the database schema is generated from the C# model; Database-First, where the model is scaffolded from an existing database; and conceptually, Model-First (less common in modern EF Core, often a variant of Code-First with a designer). This flexibility allows developers to choose the approach that best fits their project's needs.

2.6 SQL Server

Figure 8: SQL Server

“Microsoft SQL Server is a popular relational database system. It's designed to handle a wide range of tasks, from managing daily transactions to powering business intelligence and data analytics for companies.

At its core, SQL Server uses Structured Query Language (SQL), the standard language for managing and querying databases. Microsoft's own version, Transact-SQL (T-SQL), allows applications and tools to communicate directly with SQL Server databases.” [9]

Security is a core focus of SQL Server. It includes advanced features like Transparent Data Encryption (TDE) to encrypt data at rest, Row-Level Security to control access to specific rows of data, and robust authentication and authorization mechanisms to protect sensitive information.

SQL Server comes with comprehensive management and development tools, most notably SQL Server Management Studio (SSMS) and Azure Data Studio, which provide graphical interfaces and powerful capabilities for administering, developing, and optimizing databases.

SQL Server offers deep integration with other Microsoft technologies, including Azure cloud services and .NET applications.

2.7 Visual Studio

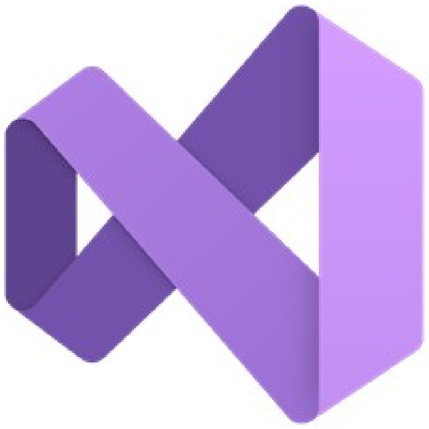


Figure 9: Visual Studio

“Visual Studio IDE is a comprehensive tool for software development. It lets you write, debug, and build code, and then easily publish your applications. Beyond basic editing and debugging, Visual Studio offers compilers, smart code completion, visual designers, and many other features to streamline the entire development process.” [10]

Visual Studio is highly versatile, supporting a broad range of programming languages beyond C# and enabling development for various platforms including Windows desktop, web, mobile (with Xamarin/MAUI), cloud (Azure), and games.

A significant strength of Visual Studio is its extensive marketplace of extensions. Developers can customize their IDE and add new functionalities, tools, and integrations for almost any need. It also integrates with Git, developers being able to manage their code, track changes, collaborate with teams, and perform operations like committing, branching, and merging directly within the IDE.

Features like GitHub Copilot provide AI-powered code suggestions and completions, directly within the IDE. This means the developer remains in control, guiding the creative process, while Copilot acts as an intelligent assistant, speeding up coding, suggesting boilerplate, and helping explore solutions, thereby embodying the principle: 'You control. AI assists.' [10]

2.8 Visual Studio Code

Figure 10: Visual Studio Code

“Visual Studio Code is a free, stand-alone code editor that runs on Windows, macOS, and Linux. It's a top choice for web and JavaScript developers, offering extensive extensions to support virtually any programming language.” [10]

VS Code has deep, built-in integration with Git, making source code management incredibly intuitive. Developers can perform common Git operations (like committing, branching, and merging) directly from the editor, fostering efficient collaboration and change tracking.

Further enhancing developer productivity, VS Code seamlessly integrates GitHub Copilot. This AI assistant provides real-time code suggestions, completes lines and functions, and can even generate entire code blocks based on natural language comments. It transforms the coding experience by acting as an intelligent pair programmer, embodying the principle that 'You control. AI assists.' by accelerating development while keeping the developer in charge of the final output.

Unlike larger Integrated Development Environments (IDEs), VS Code is renowned for its lightweight nature and fast startup times.

2.9 GitHub

Figure 11: GitHub

“GitHub is a popular platform where developers can create, store, manage, and share their code. It uses Git for version control, and also offers tools for access control, bug tracking, feature requests, task management, continuous integration, and wikis for every project.” [11]

GitHub is widely recognized as the world's largest hub for open-source projects. It fosters a massive global community of developers, enabling them to discover, contribute to, and learn from millions of public repositories. This collaborative environment is fundamental to the advancement of open-source software.

Further enhancing developer productivity, GitHub integrates GitHub Copilot, an AI-powered coding assistant. This tool provides real-time code suggestions and completions, leveraging the vast amount of code available on GitHub to help developers write code faster and more efficiently.

2.10 ngrok

Figure 12: ngrok

“ngrok is a globally distributed reverse proxy that secures, protects and accelerates your applications and network services, no matter where you run them. You can think of ngrok as the front door to your applications.

ngrok is environment independent because it can deliver traffic to services running anywhere with no changes to your environment's networking. Run your app on AWS, Azure, Heroku, an on-premise Kubernetes cluster, a Raspberry Pi, and even your laptop. With ngrok, it all works the same.

ngrok is a unified ingress platform because it combines all the components to deliver traffic from your services to the internet into one. ngrok consolidates together your reverse proxy, load balancer, API gateway, firewall, delivery network, DDoS protection and more.” [15]

ngrok is a powerful tool that creates secure, public tunnels to your local development server. This allows you to expose local services to the internet for testing webhooks, mobile apps, or showcasing work to others, greatly streamlining development workflows.

3 OBJECTIVES AND BENCHMARKS

3.1 Problem Description

Today's customers expect easy home delivery, a trend that puts smaller local food stores, restaurants, and fast-food establishments at a disadvantage. While large chains have invested in advanced delivery systems, these smaller businesses often rely on their delivery personnel but lack the integrated tools to manage the process effectively.

This disconnect leads to significant issues. Manually tracking orders and assigning deliveries are both inefficient and prone to errors, especially during busy periods. Furthermore, they often struggle to plan efficient delivery routes, which results in drivers wasting time and fuel, leading to slower deliveries and potentially compromising food quality. Customers also lack real-time updates on their orders, causing frustration and frequent calls to the store.

Ultimately, fragmented communication and outdated systems hinder a business's ability to scale. These problems translate to higher operational costs, lower customer satisfaction, and a missed opportunity for smaller food businesses to compete in the delivery market. There is a clear need for a specialized solution to help these local establishments streamline their delivery operations and enhance customer service using their existing teams.

3.2 Objectives

This application enables small local food stores, restaurants, and fast-food businesses by centralizing and streamlining their delivery operations. It will provide an intuitive platform for efficient order processing and assignment to in-house delivery personnel.

Crucially, the app seeks to enhance customer satisfaction by offering real-time order tracking and facilitating seamless communication between the store, drivers, and customers. By automating these key aspects, the application intends to significantly minimize administrative tasks, improve overall efficiency, and allow these local businesses to scale their delivery services effectively in the competitive food market.

Furthermore, by empowering businesses to utilize their delivery teams, this application helps them avoid the high commission fees imposed by large-scale third-party delivery platforms. This allows them to retain a larger share of their revenue from each order, directly contributing to their profitability and sustainable growth.

3.3 Constraints

Each participating store will need to handle its technical setup and put its unique store version of the app on platforms like the Apple App Store and Google Play Store. While our main system helps with this, each store is in charge of managing its app presence.

A key operational constraint within the system dictates that a delivery user can have only one order in progress at any given time. This ensures that delivery personnel can focus entirely on completing their current task efficiently and accurately before accepting new assignments.

3.4 Functional Requirements

3.4.1 Customer User Functional Requirements

- **User Authentication & Profile Management:**
    - The system shall allow users to register for a new account.
    - The system shall allow registered users to log in and log out.
    - The system shall allow users to manage their profile information (e.g., name, profile image).
- **Product Discovery & Selection:**
    - The system shall allow users to browse available products.
    - The system shall allow users to view information about a product, including its description, price, and images.
- **Shopping Cart & Ordering:**
    - The system shall allow users to add or remove items from their shopping cart.
    - The system shall display the current contents and total cost of the shopping cart.
    - The system shall allow users to select a delivery using map interaction.
    - The system shall allow users to place an order.
- **Order Tracking & History:**
    - The system shall allow users to track the real-time status of their active orders.

    - The system shall allow users to view their past order history.

3.4.2 Delivery User Functional Requirements

- **User Authentication & Profile Management:**
    - The system shall allow delivery users to log in and log out.
    - The system shall allow delivery users to update their profile information.
- **Order Management:**
    - The system shall display a list of available delivery orders for acceptance.
    - The system shall allow delivery users to accept delivery orders.
    - The system shall display detailed information for an accepted order (e.g., customer address, order items).
    - The system shall allow delivery users to update the status of an order (e.g., "Delivery in progress", "Delivered").
- **Navigation & Communication:**
    - The system shall provide navigation assistance to the customer's location using integrated maps.
    - The system shall allow delivery users to contact the customer (e.g., via call or in-app message).
- **Delivery History:**
    - The system shall allow delivery users to view their completed delivery history.

3.4.3 Admin User Functional Requirements

- **Product Management:**
    - The system shall allow administrators to add, view, edit products.
- **Order & Delivery Management:**
    - The system shall allow administrators to view all active and historical orders analytics.

- **Reporting & Analytics:**
  - The system shall provide reports and analytics on sales, user activity, and delivery performance.

3.5 Non-Functional Requirements

- Performance
  - The application shall respond to user requests (e.g., placing an order, loading a map, searching for products) within 3 seconds under normal load conditions.
  - Transaction Processing: Core transactional operations, such as order placement, shall be processed within 5 seconds.
  - Image Loading: All images shall load within 2 seconds on a stable connection.
- Security
  - Authorization: Access to specific functionalities and data shall be strictly controlled based on the user's assigned role (Customer, Admin, Delivery User).
  - Vulnerability Protection: The application shall be protected against common web and mobile vulnerabilities, including SQL Injection, Cross-Site Scripting (XSS)
  - Authentication: User authentication shall be secured using industry-standard protocols (e.g., JWT) with hashed passwords to prevent brute-force attacks.
- Usability
  - Intuitiveness: The user interface for all user types shall be intuitive and easy to navigate, requiring minimal training.
  - Consistency: The UI/UX shall maintain a consistent look, feel, and navigation pattern across all screens and platforms.
- Maintainability
  - Code Quality: The codebase shall be well-documented, modular to facilitate future enhancements and bug fixes.
  - Deployment: New features and updates shall be deployable with minimal downtime and disruption to service.

4 SOFTWARE ARCHITECTURE

4.1 General

My application's architecture is highly distributed and modular, designed to allow each food business to operate its own entirely independent digital delivery system. Instead of a single shared backend, each participating store will deploy and manage its own separate instance of the application components.

At the core of each store's independent system are distinct client and server components. On the client side, every store will have a single branded React Native application. This unified app is designed to serve all user roles – customers, store management, and delivery personnel – by presenting different views and functionalities based on the user's login and permissions. This individual store app will be published to app stores under each store's identity.

This React Native frontend for a specific store will communicate directly with that same store's dedicated Backend Service, which is built using ASP.NET API. This means each store maintains its own exclusive backend to manage its unique customer data, order processing, product catalogs, and transaction history. This approach ensures complete data isolation and operational independence for each business.

This architectural design prioritizes the autonomy of each store, allowing them full control over their data and operations. It also inherently provides scalability at an individual store level, ensures security through data compartmentalization, and offers the flexibility for each business to manage its specific delivery ecosystem without sharing backend resources with other stores.

4.2 Database Structure

4.2.1 General

The data for each individual store's application is managed using Microsoft SQL Server 2019. This robust relational database management system was chosen for its reliability, security features, and seamless integration with the ASP.NET API backend.

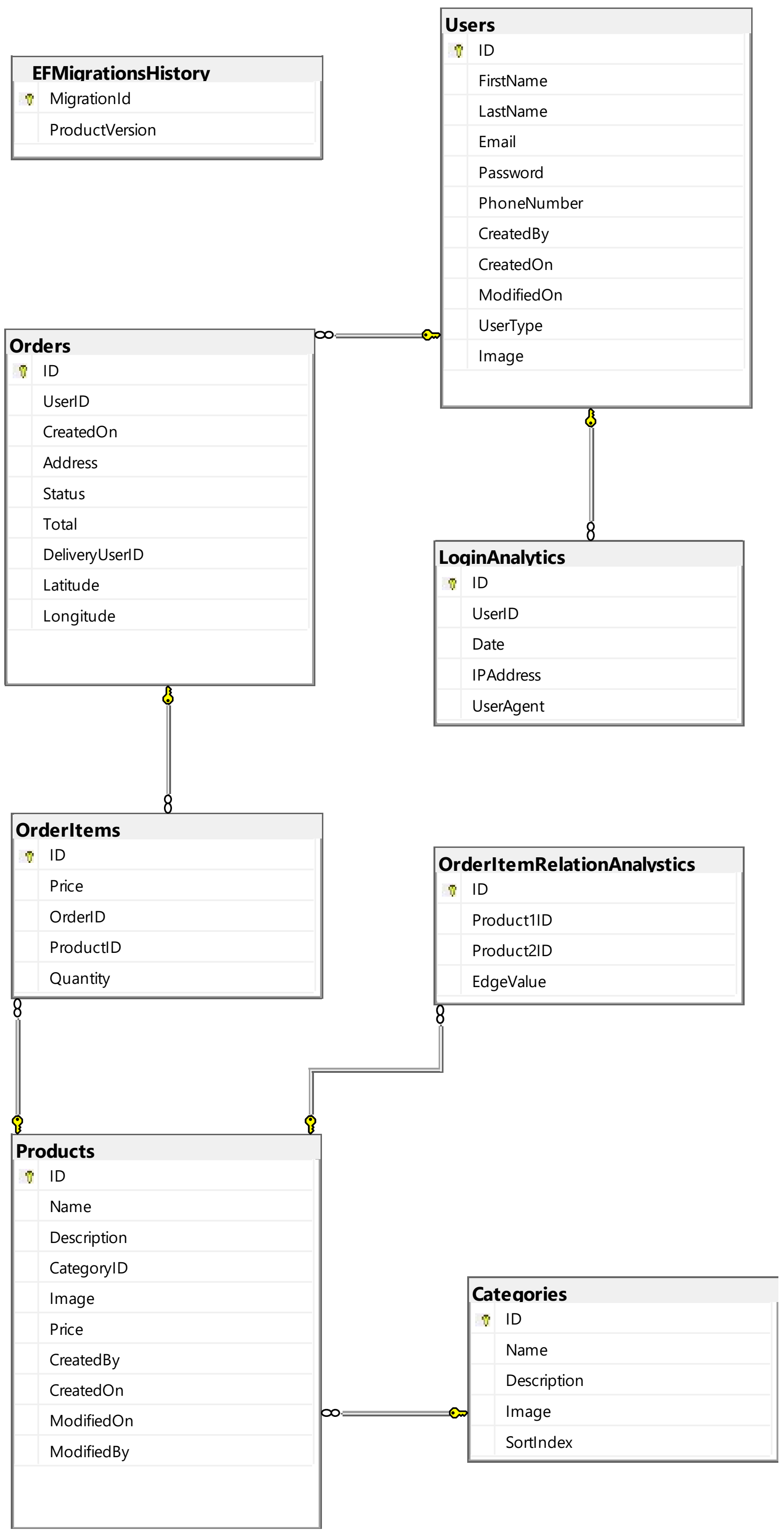


Figure 13: Database ERD (entity-relationship diagram)

4.2.2 Stored Procedures

The stored procedure was specifically created so it could be added as a distinct step within the SQL Server Agent, enabling automated execution of the contained logic.

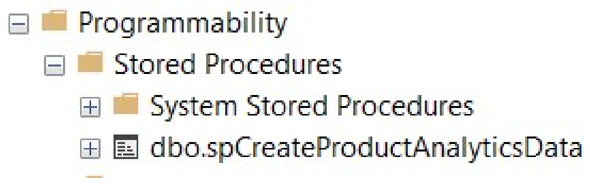


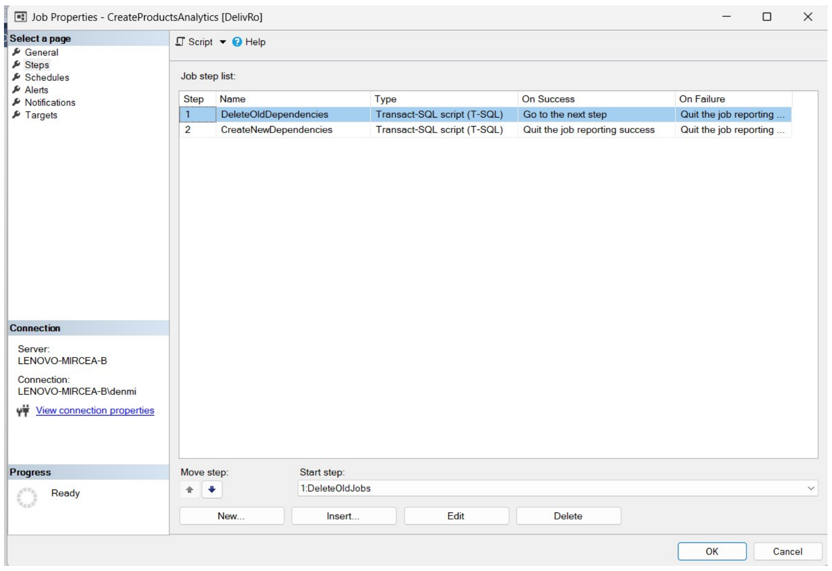
Figure 14: Stored Procedures Structure

4.2.3 SQL Server Agent Jobs

Building on the use of stored procedures, I also leveraged **SQL Server Agent Jobs** to manage time-consuming tasks within the database. These jobs are configured to execute specific operations periodically, offloading heavy processing from the live user experience and thereby ensuring that user performance is not negatively impacted during peak hours.

For example, computing all the recommendation cost edges from my algorithm was computed using a stored procedure scheduled every hour.

To ensure system stability, SQL Server Agent can be configured to send email notifications directly to administrators if any scheduled job fails, allowing for immediate intervention.



Job step list:

Step	Name	Type	On Success	On Failure
1	DeleteOldDependencies	Transact-SQL script (T-SQL)	Go to the next step	Quit the job reporting ...
2	CreateNewDependencies	Transact-SQL script (T-SQL)	Quit the job reporting success	Quit the job reporting ...

Server:
LENOVO-MIRCEA-B

Connection:
LENOVO-MIRCEA-B\denmi

Ready

Move step:

Start step: 1:DeleteOldJobs

Figure 15: SQL Server Agent Job Configuration

4.3 Backend Structure

4.3.1 General

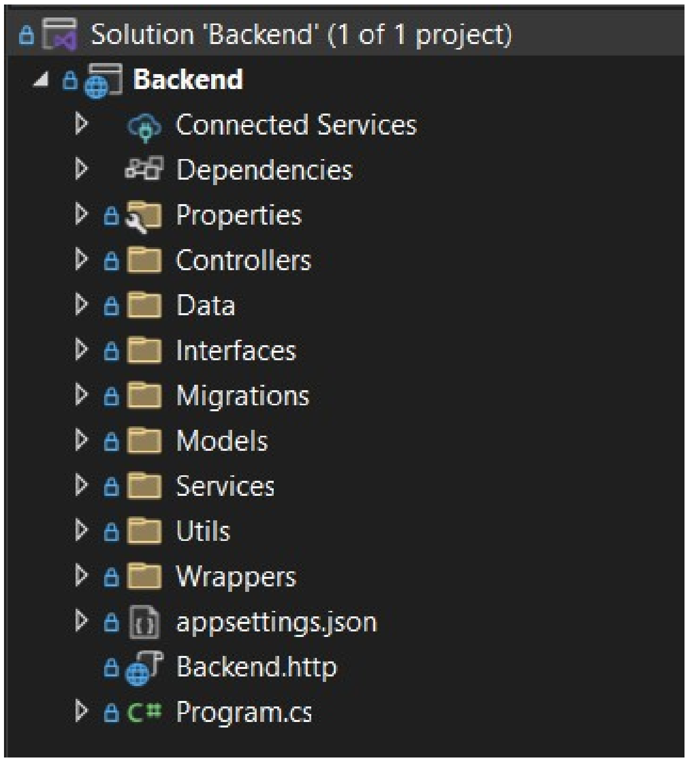


Figure 16: General Backend Structure

The backend of my application is built using ASP.NET 8.0. Its structure, as shown in the provided solution explorer, indicates a well-organized and modular approach to development.

Key components and their purposes within the backend structure include:

- ➢ The Controllers folder contains the classes that handle incoming HTTP requests from the React Native frontend, defining the API endpoints that client applications interact with. They serve as the entry points for my application's logic.
- ➢ The Interfaces folder defines the contracts (interfaces) for the services and other components within my application, promoting loose coupling and testability.
- ➢ The Migrations folder holds the database migration scripts used to manage and evolve my application's database schema over its lifecycle.

- The Models folder contains the data models or entities that represent the structure of the data stored in the SQL Server database and exchanged with the frontend.
- The Services folder encapsulates my application's core business logic and operations, orchestrating interactions between controllers and data access mechanisms.
- The appsettings.json file stores my application's configuration settings, including database connection strings and other environment-specific configurations.

This structure follows common ASP.NET best practices, promoting separation of concerns and making the codebase maintainable and scalable for each independent store's backend.

4.3.2 Controllers

The controllers in my ASP.NET 8.0 backend adhere to a well-defined and consistent structure, inheriting from a custom BaseController. This approach centralizes common functionalities and architectural patterns for all API endpoints.

Each controller in my application is automatically routed with api/[Controller]/[action], meaning that API endpoints are consistently organized by controller name and action method. The [ApiController] attribute is applied, enabling conventions for API behavior such as automatic model validation and HTTP 400 responses. A critical aspect of the design is the [Authorize] attribute, applied at the BaseController level. This ensures that all endpoints by default require authentication, enhancing the security of the API by preventing unauthorized access to protected resources.

Furthermore, the BaseController provides protected methods to simplify access to user information from authenticated requests.

- GetRequestUserID(): This method extracts the authenticated user's unique identifier (ID) from their claims, facilitating operations that require knowledge of the current user.
- GetRequestUserType(): This method retrieves the user's role or type (e.g., customer, store management, delivery personnel) from their claims, enabling role-based authorization logic within the controllers and services.

This standardized structure ensures consistency, promotes reusability of security and user context logic, and simplifies the development of new API endpoints across the application.

4.3.3 Migrations

The Migrations folder within my ASP.NET 8.0 backend is crucial for managing the evolution of my application's database schema. This folder contains a series of Entity Framework Core migration files, each representing a specific change or update made to the Microsoft SQL Server 2019 database structure over time.

Each migration file is uniquely identified by a timestamp followed by a descriptive name. This naming convention ensures that changes are applied in the correct chronological order. These migrations allow for version control of the database schema, enabling controlled updates, rollbacks, and consistent database states across different environments. Specific schema changes managed through these migrations include the introduction of user table types, product and category definitions, various updates to order and order item tables, and the addition of login and order item analytics features. [12]

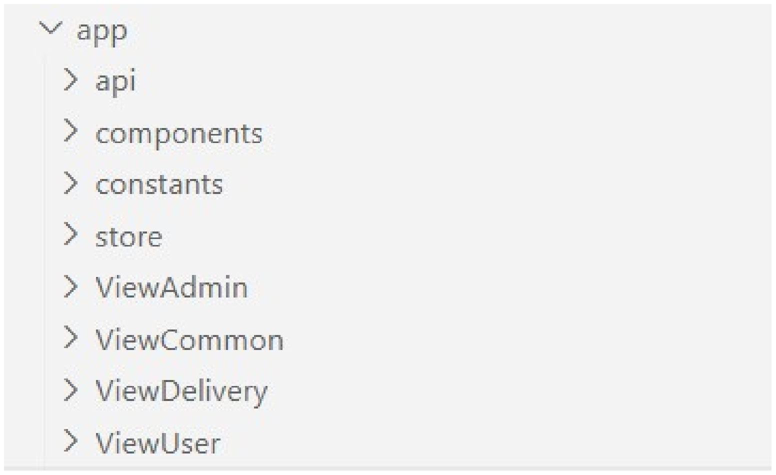
4.3.4 Services

The Services folder encapsulates my application's core business logic and operations, orchestrating interactions between controllers and data access mechanisms. A fundamental aspect of this architecture is that each service is defined by, and implements, a specific interface. This design choice is crucial for enabling robust dependency injection. Controllers, instead of directly instantiating a concrete service class, depend solely on its interface.

This approach promotes loose coupling, making the system more modular, flexible, and significantly easier to maintain. It also greatly enhances testability, as different implementations of a service (e.g., mock services for testing) can be seamlessly swapped in without altering the controller's code. This ensures that the business logic remains distinct and highly manageable while providing controllers with the necessary functionalities in a standardized manner.

4.4 Frontend Structure

4.4.1 General



```
∨ app
  > api
  > components
  > constants
  > store
  > ViewAdmin
  > ViewCommon
  > ViewDelivery
  > ViewUser
```

Figure 17: General Frontend Structure

The app folder serves as the root for this single frontend application. Within this structure, key directories serve specific purposes, contributing to the app's overall functionality and organization:

- api: This folder contains code responsible for making requests to the backend ASP.NET API, handling API service calls, and managing data fetching for the app.
- components: This directory contains reusable UI components (e.g., dropdowns, input fields, cards) that are used across different parts of the application, promoting consistency and reusability throughout the single app.
- constants: This folder stores static values, configurations, or enumerated types that are frequently used throughout the application, such as types endpoints or placeholder images constants.
- store: This folder is designed with Redux for state management. It contains the logic for managing the app's global state, ensuring data consistency across different views and components.
- ViewAdmin: This folder contains the specific UI components and logic related to the administrative views presented within the app, accessible by store management roles.

- ViewCommon: This directory holds common UI components or layouts shared across various user roles within the app, reducing redundancy in design and code.
- ViewDelivery: This contains the UI and logic specific to the delivery personnel's interface within the app, such as order assignment screens and delivery status updates.
- ViewUser: This folder encompasses the UI and logic pertinent to the customer-facing views within the app, including product Browse, order placement, and personal profile management.

This organized folder structure helps in maintaining a clean, scalable, and understandable codebase for the single React Native application that serves multiple user roles.

4.4.2 API utils

In my React Native frontend, all communication with the backend API is centralized through dedicated utility functions found within the *apiUtils.tsx* file in the api folder. This approach ensures that every request is made consistently and robustly. These utility functions, such as *getRequestWithHeaders, getCall,* and *postCall,* automatically attach necessary headers (like authorization tokens) to outgoing requests. Furthermore, they are designed to manage possible errors, handling responses from the backend to identify and process various issues such as network failures or server errors, providing a unified error handling mechanism across the application.

4.4.3 Metro bundler config

The metro.config.js file is essential for configuring Metro, the JavaScript bundler used by React Native. This configuration file allows for customization of how Metro processes and bundles your application's code.

It was used to set up custom module resolution logic. This is particularly valuable for cross-platform development, as it allows you to specify different versions of a module or package to be used depending on the target platform (e.g., mobile or web). This custom resolution is specifically implemented to avoid errors caused by the incompatibility of certain packages or functionalities with different environments.

4.4.4 App Configuration

The app.json file serves as the central configuration manifest for your Expo React Native application. It defines essential metadata and platform-specific settings that dictate how your app behaves and appears across different devices and environments.

This file includes fundamental details such as the application's name, unique identifier (slug), and version number. It also controls orientation settings, the path to the app's icon, and custom URL schemes for deep linking. You can configure the user interface style.

For platform-specific behaviors, app.json provides dedicated sections for iOS, Android, and web. These sections allow you to set platform-specific bundle identifiers, adaptive icon configurations for splash screens, and web-specific bundling options.

Furthermore, app.json is where you declare and configure various Expo plugins that extend your app's capabilities, such as those for routing (like Expo Router) or managing splash screens. It also handles experimental features, custom extra variables for project-specific data, and defines properties related to app ownership, runtime versioning, and over-the-air update mechanisms.

```
{
 "expo": {
  "name": "DeliveRO",
  "slug": "delivero",
  "version": "1.0.1",
  "orientation": "portrait",
  "icon": "./assets/images/delivero.png",
  "scheme": "delivero",
  "userInterfaceStyle": "automatic",
  "newArchEnabled": true,
  "ios": {
   "supportsTablet": true,
   "bundleIdentifier": "com.denmircea.delivero",
   "infoPlist": {
    "ITSAppUsesNonExemptEncryption": false
   },
   "adaptiveIcon": {
    "foregroundImage": "./assets/images/delivero.png",
    "backgroundColor": "#555500"
   }
  },
  "android": {
   "adaptiveIcon": {
    "foregroundImage": "./assets/images/delivero.png",
    "backgroundColor": "#ffffff"
   },
   "edgeToEdgeEnabled": true,
   "package": "com.denmircea.delivero"
  },
  "web": {
   "bundler": "metro",
   "output": "static",
   "favicon": "./assets/images/icon.png",
   "aliases": {
    "react-native-maps": "@teovilla/react-native-web-maps"
   }
```

```
    },
    "plugins": [
      "expo-router",
      [
        "expo-splash-screen",
        {
          "backgroundColor": "#555500",
          "image": "./assets/images/icon.png",
          "imageWidth": 200
        }
      ]
    ],
    "experiments": {
      "typedRoutes": true
    },
    "extra": {
      "router": {},
      "eas": {
        "projectId": "570b973c-749e-4f80-a464-df1ad6c3217b"
      }
    },
    "owner": "denmircea",
    "runtimeVersion": {
      "policy": "appVersion"
    },
    "updates": {
      "url": "https://u.expo.dev/570b973c-749e-4f80-a464-df1ad6c3217b"
    }
  }
}
```

4.4.5 Package Configuration

The *package.json* file is fundamental to every JavaScript project, including my React Native application. It serves as the manifest for the project, detailing crucial metadata and managing dependencies. This file records the project's name, version, and a brief description. More importantly, it lists all the third-party libraries and packages that my application relies on, separating them into dependencies (required for the app to run in production) and devDependencies (needed only during development or testing). It also defines scripts for common tasks like starting the application, running tests, or building the project. When other developers set up the project, **npm install** reads this file to automatically download all necessary packages, ensuring a consistent development environment and enabling collaborative work.

The *package-lock.json* file works alongside package.json as a crucial companion. It's automatically created by npm and acts like a detailed receipt of every single piece of code (package) installed in your project.

While *package.json* tells the system what range of versions you're okay with for a dependency (like "any 1.x version"), *package-lock.json* records the exact version of every package that was actually installed. This "locks down" all versions, ensuring that when anyone else sets up your project, they get precisely the same code as you. This is vital for consistent development and prevents those frustrating "it works on my machine!" issues.

5 SOFTWARE IMPLEMENTATION

5.1 Security

5.1.1 Password encryption

My application implements a secure method for handling user passwords to protect sensitive user data. Passwords are never stored in plain text; instead, **a one-way hashing algorithm** is used.

GetUserPasswordHash method is designed to securely generate a hash of a given password. It begins by initializing a SHA256 hashing algorithm. The input password string is then converted into a byte array, which is subsequently processed by the SHA256 algorithm to compute its cryptographic hash. Finally, this resulting hash (a byte array) is converted into a hexadecimal string representation for storage or comparison. This entire process ensures that the original password is never stored in plain text, enhancing security.

It utilizes the SHA256 (Secure Hash Algorithm 256), a cryptographic hash function, to create a fixed-size string of characters.

When a user provides a password, it's first converted into a sequence of bytes using UTF-8 encoding.

The SHA256 algorithm then computes a hash from these bytes.

This computed hash, a seemingly random string of hexadecimal characters, is what gets stored in the database, not the original password.

This hashing mechanism ensures that even if the database were compromised, the actual user passwords would remain unreadable, significantly enhancing user security. To verify a user's login, the provided password is hashed using the same method and compared against the stored hash.

Plain Password	Encrypted Password
admin	8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918
test	9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08

password1234	b9c950640e1b3740e98acb93e669c65766f6670dd1609ba91ff41052ba48c6f3
12341234	1718c24b10aeb8099e3fc44960ab6949ab76a267352459f203ea1036bec382c2
parola	a80b568a237f50391d2f1f97beaf99564e33d2e1c8a2e5cac21ceda701570312

Table 1: Password hashing

While SHA256 is a robust hashing algorithm, there is a growing concern that websites are increasingly indexing common SHA256 hashes, which can potentially lead to the 'decryption' (more accurately, reverse-lookup) of simple or common passwords. To mitigate this vulnerability and enhance security, a solution can be implemented in the future to incorporate a SALT string. This unique, randomly generated string will be added to each password before hashing, making each resulting hash unique even for identical passwords, thus preventing dictionary attacks and rainbow table lookups.

Plain Password	**SHA256(plainPassword + “secret@1234”)**
admin	406306ad00711756a95929630dde1a39df4c2f2e45cdd47ffc83f1bfd03a4f62
test	9d5999c2ab51cb8210211b96940cab987b872a15e2def0039317507613fb6d60
password1234	f4137c57b78bb3af049db7d5ebaed599238cb40308b1c744da1ee5b8e76aae26
12341234	959f167ace9a5a372cee62f7d8a9c1abb7c26aee6279c91d427c9d2cccd46496
parola	a03aa52157da6c26509e38e6e7bd11078ae4640bd57d9e92f7565b720c978e45

Table 2: Password hashed with salt secret

5.1.2 JWT Token

My application utilizes authorization with JWT (JSON Web Token) tokens for every request. This means that after a user successfully authenticates, a JWT is issued and must be included in the headers of all subsequent requests to access protected resources, ensuring secure communication and verifying user identity for each interaction.

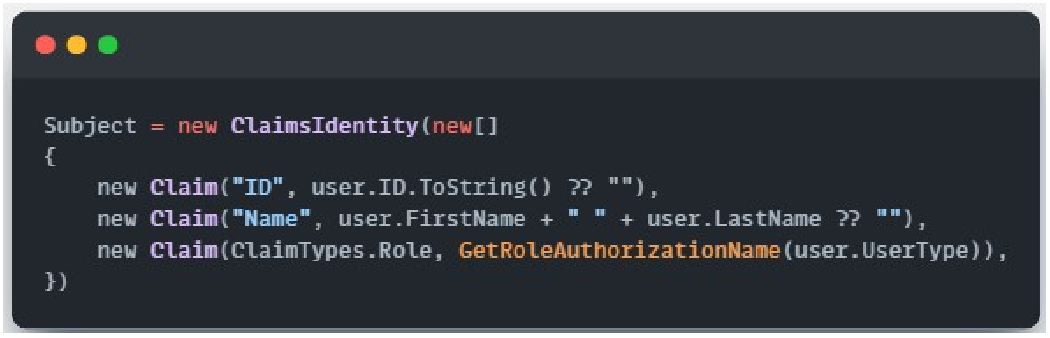
After every login in my application there is generated a JWT Token sent to the client and saved in the local storage of the client device.

The data stored in the JWT claims:

- ID (user id as a string)

- Name (first name + last name)

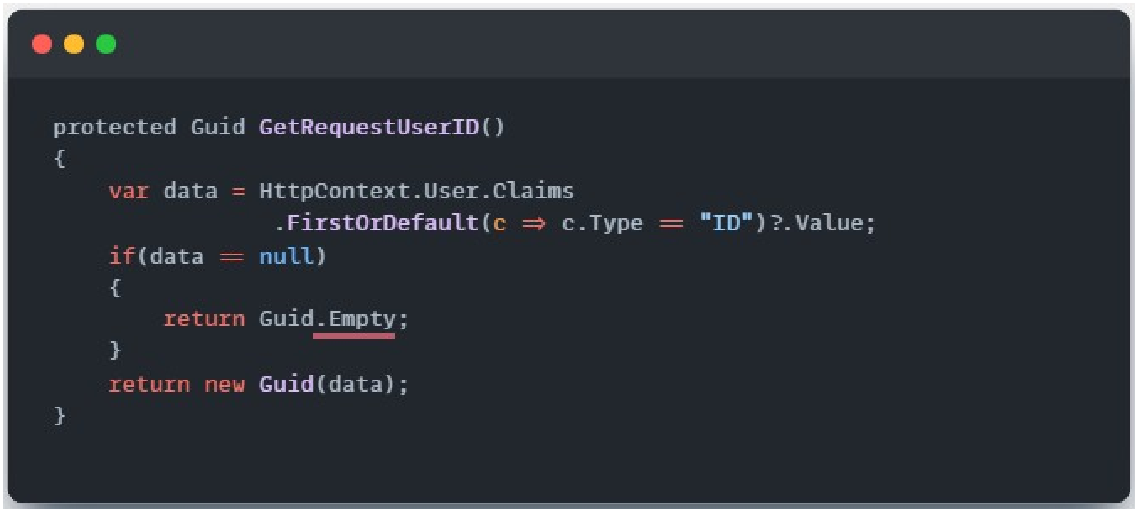
- Role (Admin/Delivery/User)



```
Subject = new ClaimsIdentity(new[]
{
    new Claim("ID", user.ID.ToString() ?? ""),
    new Claim("Name", user.FirstName + " " + user.LastName ?? ""),
    new Claim(ClaimTypes.Role, GetRoleAuthorizationName(user.UserType)),
})
```

Figure 18: Define JWT Claims

In my application, once a user successfully logs in, subsequent requests that require the user's ID do not request this information from the frontend. Instead, the user ID is securely retrieved from the claims embedded within the authenticated JWT token. This approach enhances security by preventing client-side tampering with user identification and ensures that the server consistently knows the context of the authenticated user for each request.

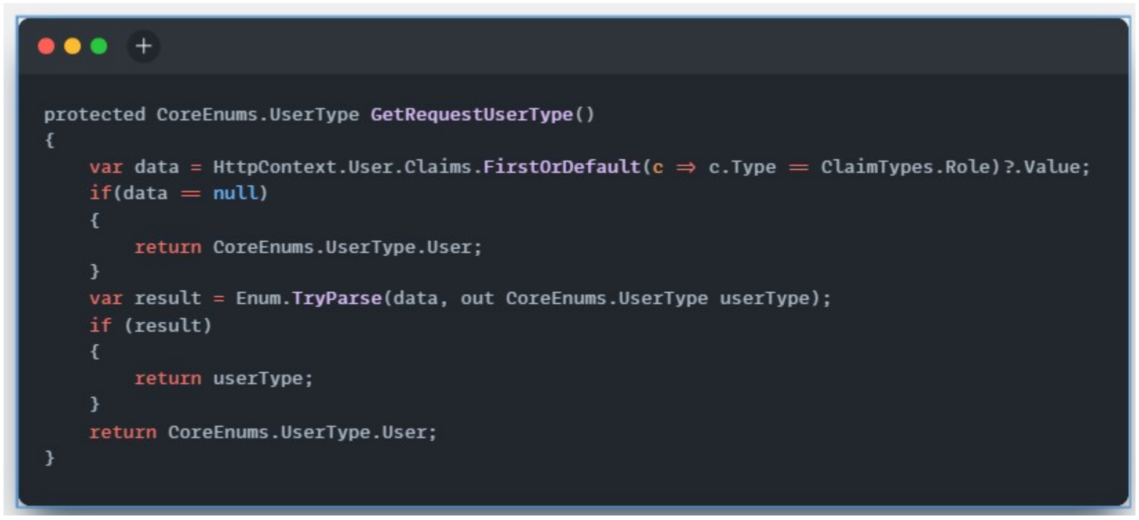


```
protected Guid GetRequestUserID()
{
    var data = HttpContext.User.Claims
                .FirstOrDefault(c => c.Type == "ID")?.Value;
    if(data == null)
    {
        return Guid.Empty;
    }
    return new Guid(data);
}
```

Figure 19: Get UserID from JWT

This method is implemented in BaseController, which every controller inherits. This enables the possibility to check for endpoints where the user has access to that data (e.g., based on the user ID, he can retrieve only orders from that account, the user ID is not sent as a parameter of the request).

Also, the role of the user is decided in the same way:



```
protected CoreEnums.UserType GetRequestUserType()
{
    var data = HttpContext.User.Claims.FirstOrDefault(c => c.Type == ClaimTypes.Role)?.Value;
    if(data == null)
    {
        return CoreEnums.UserType.User;
    }
    var result = Enum.TryParse(data, out CoreEnums.UserType userType);
    if (result)
    {
        return userType;
    }
    return CoreEnums.UserType.User;
}
```

Figure 20: Get User Role from JWT

The same role is also used to check that a user has access to a specific endpoint (e.g., [Authorize(Roles = CoreEnums.UserTypeNames.BackOfficeAdmin)] used to protect analytics endpoints from anyone that tries to access these endpoints without the specified role).

5.2 Use case diagrams

5.2.1 Administrator

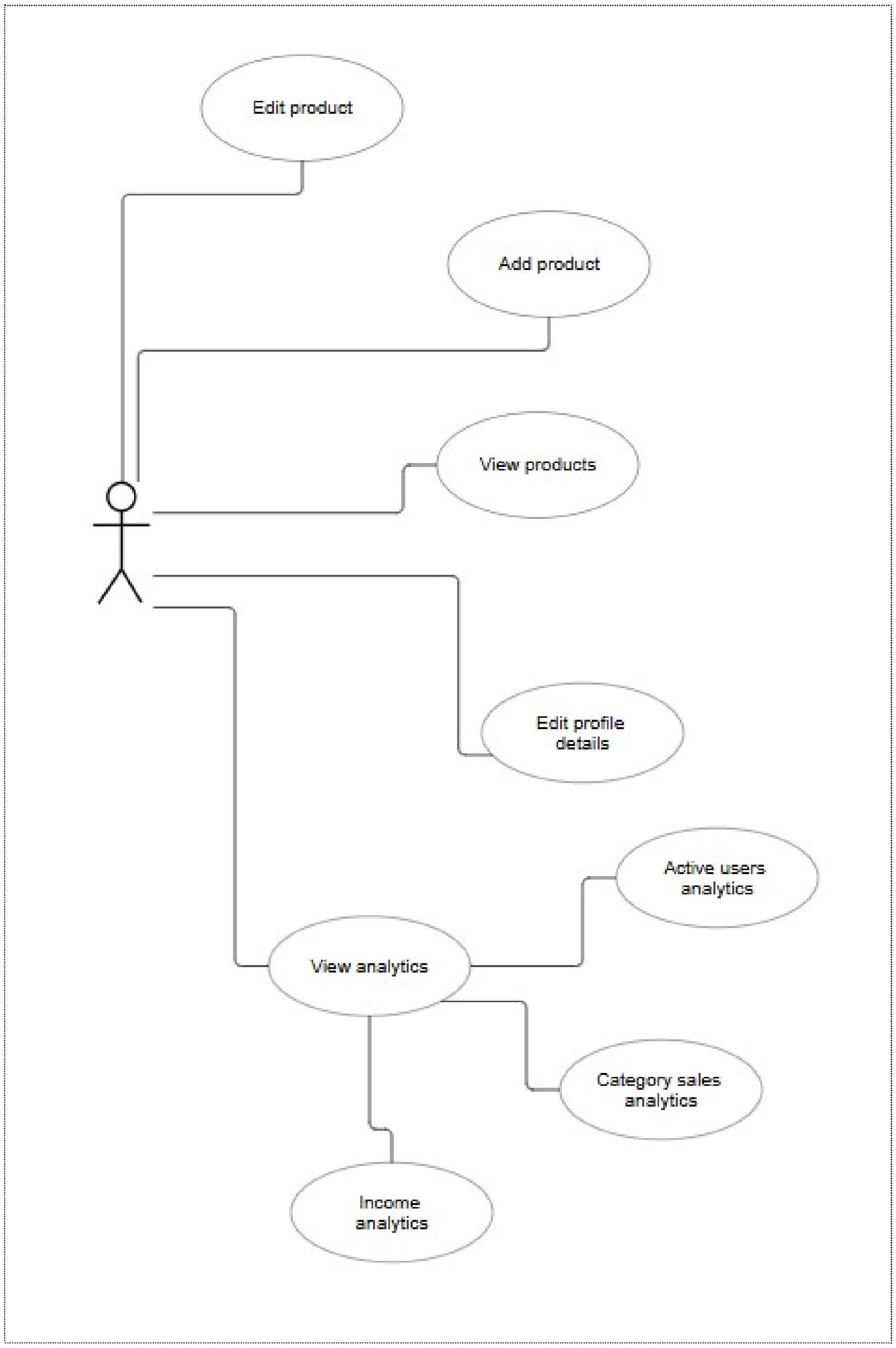


Figure 21: Administator use case

5.2.2 Delivery User

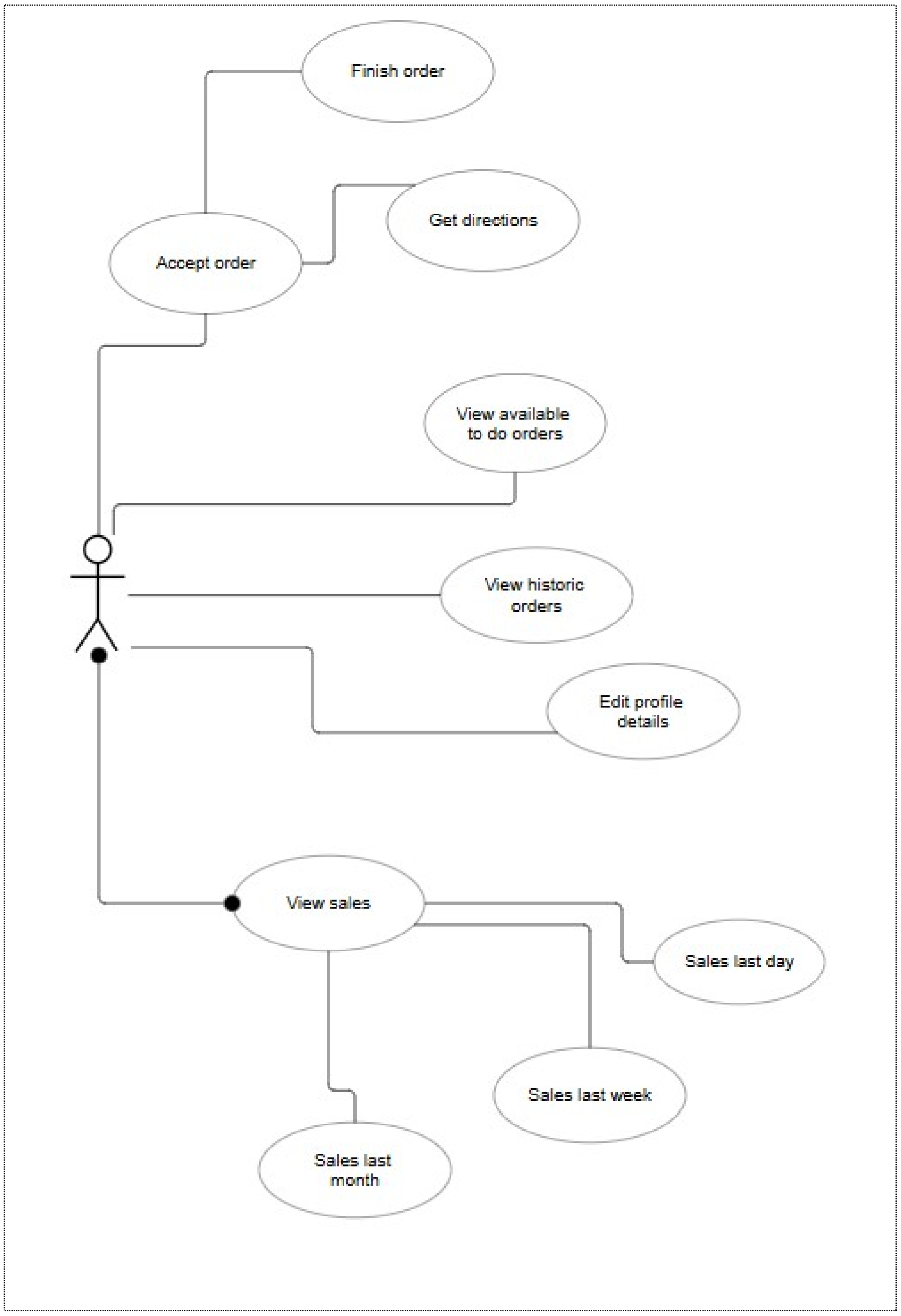


Figure 22: Delivery User use case

5.2.3 Customer/User

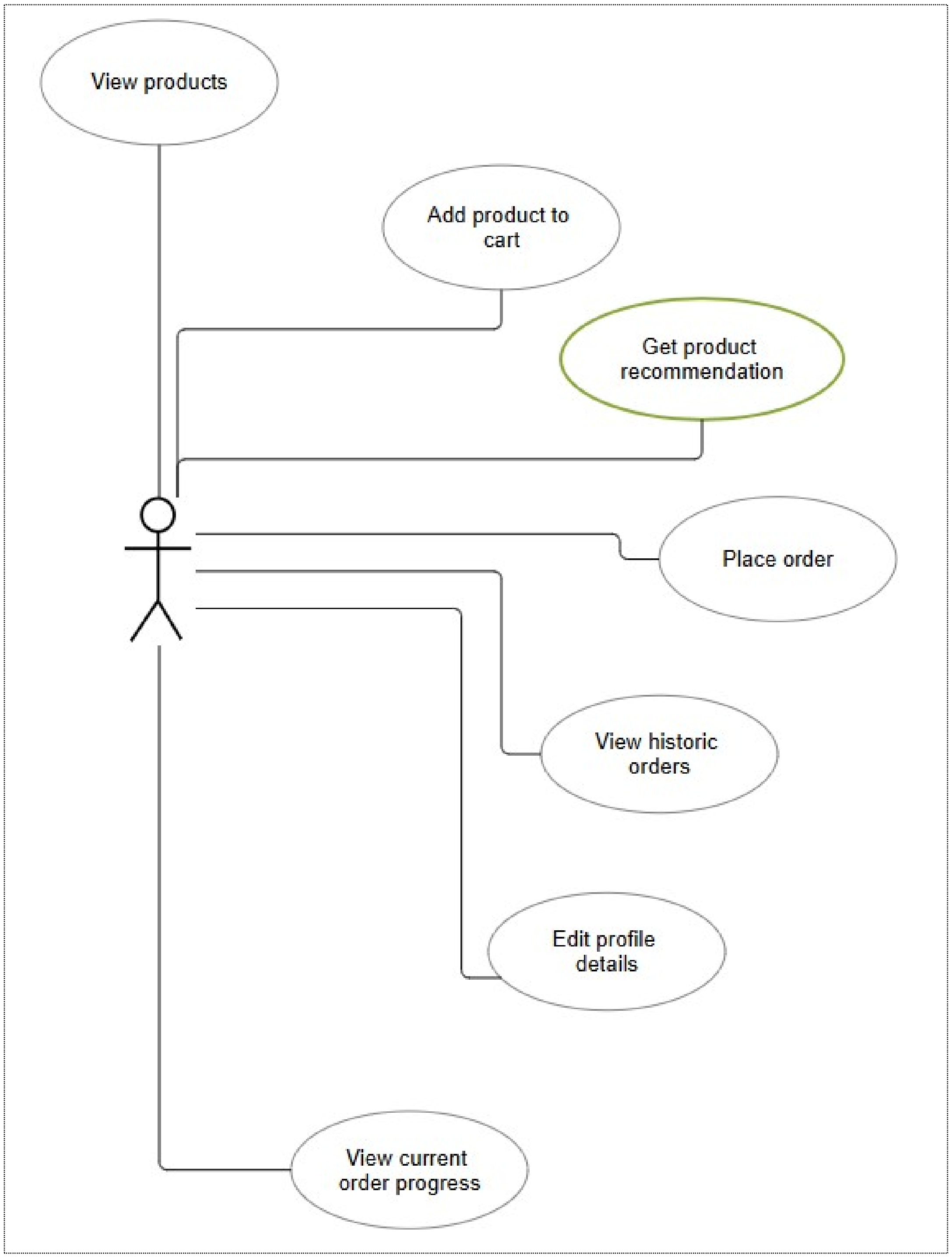


Figure 23: Customer User use case

5.3 Recommendation system

5.3.1 General

Fundamentally, the entirety of transactional data spanning the preceding fourteen-day period is processed and algorithmically transformed into the weighted edges of a graph. Each edge within this graph represents a direct association between two distinct products, with its assigned weight reflecting the frequency of their co-occurrence across orders. Specifically, the weight quantifies the number of shared orders in which both connected products were purchased together.

5.3.2 Database considerations

To ensure optimal performance and accommodate the potentially vast datasets, the computation and persistence of each edge's weight within the recommendation system are managed by an automated SQL Server Agent job. This job is scheduled for hourly execution, performing a systematic refresh of the existing relationships by first clearing outdated entries and then inserting the newly computed, updated ones.

There are 3 essential fields for a generic edge saved in the database:

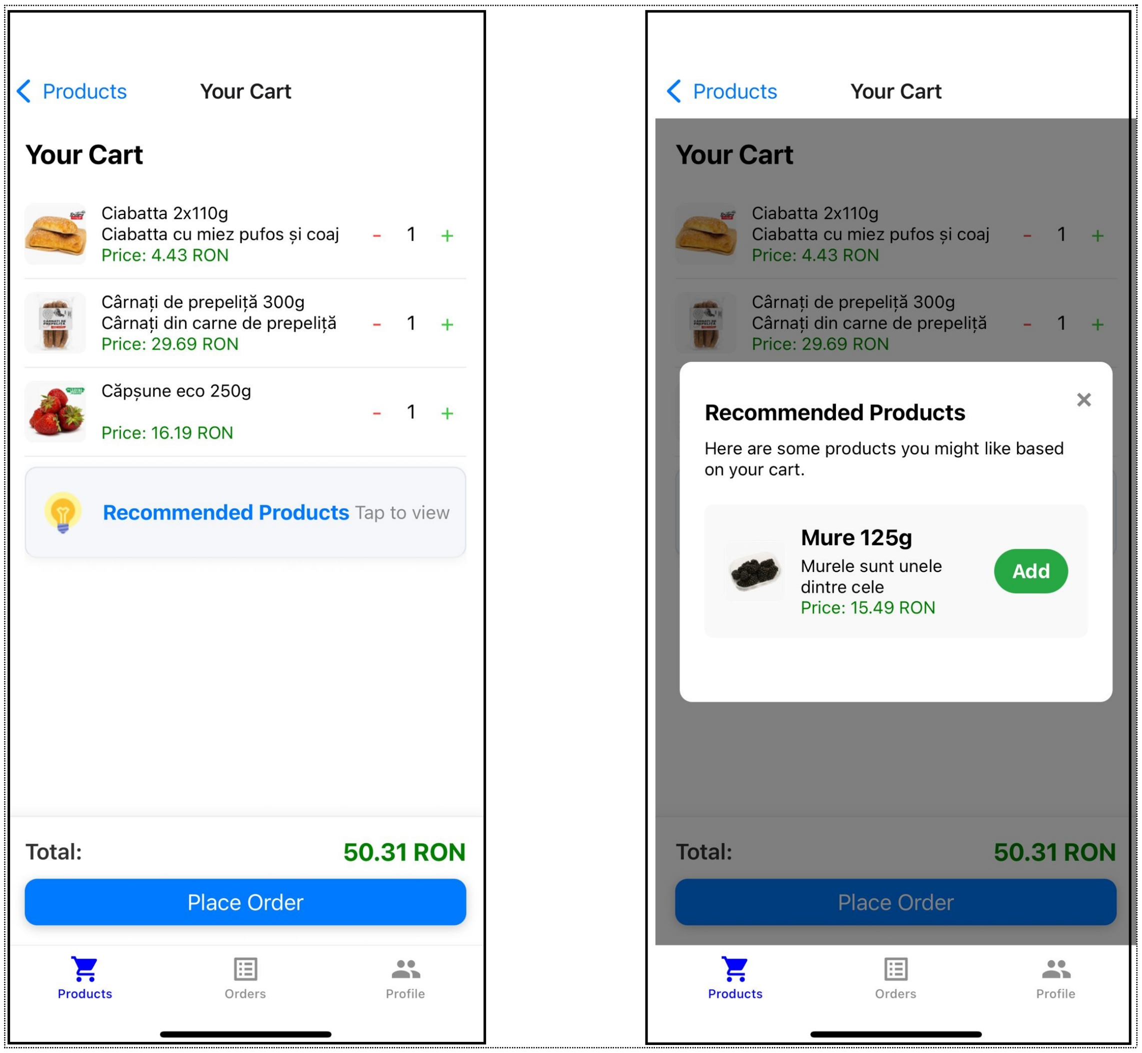
Field	Scope
Product1ID (Guid)	The 2 ids are FKs to the products represented. From the construction of the entities is known that Product1ID < Product2ID.
Product2ID (Guid)	
EdgeValue (decimal)	Edge value is mainly the number of orders containing this unique pair of products.

Table 3: Recommendation system main columns database

Edge value was defined as decimal for later implementations of the edge value adding points maybe for the category matching.

5.3.3 Data computation

To generate product recommendations, the client application transmits the contents of the user's current shopping cart to the backend service. The backend then computes top-scoring product suggestions based on the items already within the cart, aiming to enhance the user's shopping experience.


Products Your Cart

Your Cart

Ciabatta 2x110g
Ciabatta cu miez pufos și coaj
Price: 4.43 RON
– 1 +

Cârnați de prepeliță 300g
Cârnați din carne de prepeliță
Price: 29.69 RON
– 1 +

Căpșune eco 250g
Price: 16.19 RON
– 1 +

Recommended Products Tap to view

Total: 50.31 RON

Place Order

Products Orders Profile

Products Your Cart

Your Cart

Ciabatta 2x110g
Ciabatta cu miez pufos și coaj
Price: 4.43 RON
– 1 +

Cârnați de prepeliță 300g
Cârnați din carne de prepeliță
Price: 29.69 RON
– 1 +

Recommended Products

Here are some products you might like based on your cart.

Mure 125g
Murele sunt unele dintre cele
Price: 15.49 RON
Add

Total: 50.31 RON

Place Order

Products Orders Profile

Figure 24: Product recommendation example

In the above example, another order exists in the database that contains “Capsune eco 250g” together with “Mure 125g”. This creates a link with weight 1 between these 2 products.

The selection of the recommended products will be explained using the following weighted graph:

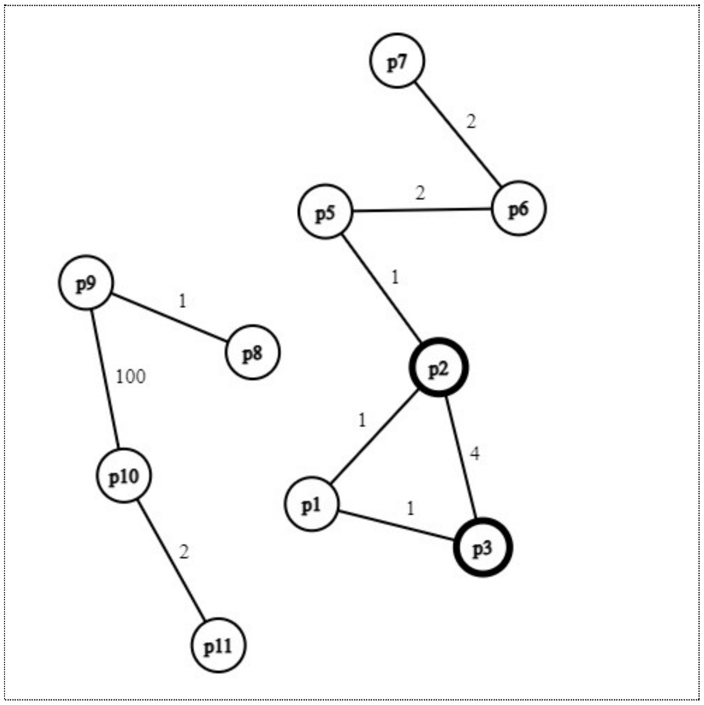


Figure 25: Weighted graph recommendation

The above weighted graph is the result of the database edges computation during the SQL Server Agent Job. In the current example, if the cart items are **p2** and **p3**, the top recommendation products will be **p1** with a score of **2** and **p5** with a score of **1**. The score is computed as the sum of all direct edges going from a selected cart item.

5.4 Code first approach

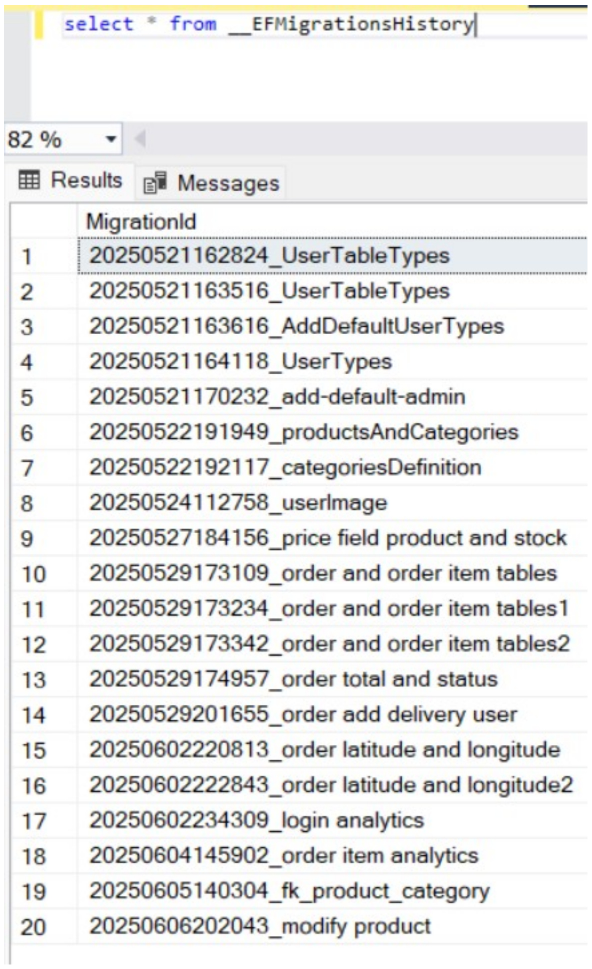
In developing the application's data persistence layer, a Code-First approach was strictly adopted. This methodology dictates that the database schema is derived directly from the application's C# (or similar object-oriented) data models, rather than the schema dictating the models. This provided a highly developer-centric workflow, allowing for the definition of entities, relationships, and data types directly within the application's codebase.

To synchronize these evolving code models with the underlying Microsoft SQL Server 2019 database, auto-generated migration scripts were extensively utilized. Whenever changes were made to the data models (e.g., adding a new entity, modifying a property, establishing new relationships), the

ORM (Object-Relational Mapper) could automatically detect these differences and generate corresponding migration scripts. These scripts, containing the necessary SQL commands to create, alter, or drop database objects, were then systematically applied to the database. This approach ensured that the database schema remained consistently in sync with the application's data structures, facilitating controlled evolution, enabling version control of schema changes, and simplifying collaborative development by providing a clear, trackable history of database modifications.

A significant advantage of this migration-driven approach is the ease with which a completely new database instance can be created. By simply applying the sequence of auto-generated migrations, a fresh database is automatically provisioned and structured to match the latest application models. This process can also be configured to include default data, such as initial user roles, core product categories, or system configurations, ensuring that a newly created database is immediately populated with essential seed data and ready for use. This capability drastically simplifies environment setup for development, testing, and new deployments, ensuring consistency and reducing manual configuration effort.

Furthermore, to meticulously track the evolution of the database schema, all applied migrations are recorded within a dedicated table, typically named __EFMigrationsHistory (for Entity Framework Core). This internal system table serves as a comprehensive log, storing details about each migration that has been successfully applied to the database, including its unique identifier and the timestamp of its application. This mechanism is crucial for the ORM to determine the current state of the database schema, identify which migrations still need to be executed, and prevent the re-application of already deployed schema changes, thereby ensuring a controlled and consistent database update process.


```sql
select * from __EFMigrationsHistory
```

	MigrationId
1	20250521162824_UserTableTypes
2	20250521163516_UserTableTypes
3	20250521163616_AddDefaultUserTypes
4	20250521164118_UserTypes
5	20250521170232_add-default-admin
6	20250522191949_productsAndCategories
7	20250522192117_categoriesDefinition
8	20250524112758_userImage
9	20250527184156_price field product and stock
10	20250529173109_order and order item tables
11	20250529173234_order and order item tables1
12	20250529173342_order and order item tables2
13	20250529174957_order total and status
14	20250529201655_order add delivery user
15	20250602220813_order latitude and longitude
16	20250602222843_order latitude and longitude2
17	20250602234309_login analytics
18	20250604145902_order item analytics
19	20250605140304_fk_product_category
20	20250606202043_modify product

Figure 26: Migrations history on database

5.5 Performance

Throughout the application's development, a consistent practice when retrieving data models from the database was the deliberate use of the **.AsNoTracking()** method. This directive, typically employed with Object-Relational Mappers (ORMs) like **Entity Framework Core**, explicitly instructs the database context not to track the entities returned by a query.

The primary benefit of this approach stems from scenarios where the retrieved data is intended solely for **read-only operations** and will not be modified or saved back to the database. By opting out of change tracking, the application significantly reduces overhead. This translates directly into **improved performance** due to faster query execution, as the ORM avoids the computational cost of setting up change tracking proxies and detecting modifications. Furthermore, it leads to **enhanced**

memory efficiency, as the entities are not cached within the context's change tracker, thereby consuming less memory. Crucially, **.AsNoTracking()** also eliminates the risk of **unintended side effects**, preventing any accidental modifications to the retrieved objects from being inadvertently persisted to the database. This judicious use of **.AsNoTracking()** contributes to a more efficient, responsive, and robust application architecture, particularly for data retrieval operations.

5.6 React Native Maps

"Maps play a pivotal role in modern mobile applications, providing users with a spatial context that enhances the overall user experience. In the realm of React Native, integrating maps into your app can seem like a complex task, but with the right tools and libraries, it becomes a seamless process. This article serves as your guide to harnessing the power of maps in React Native, exploring key concepts, libraries, and best practices." [14]

The React Native Maps library is an integral component of my application, serving critical geospatial functionalities across two distinct screens: the Place Order screen and the Delivery Order screen.

On the Place Order screen, the map provides a crucial interface for customers to accurately specify their delivery location. Users interact with the map to pinpoint their exact address, visually confirm their position, and potentially drop a custom pin for precise delivery instructions. This enhances the ordering process by minimizing address ambiguities and improving delivery accuracy, directly contributing to a smoother user experience.

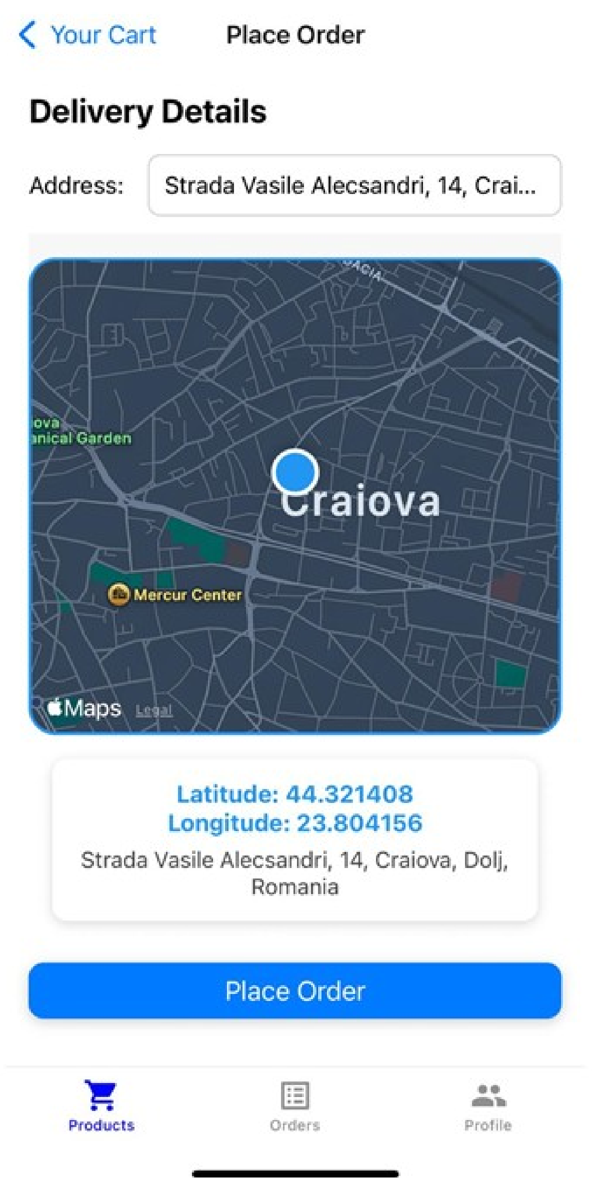


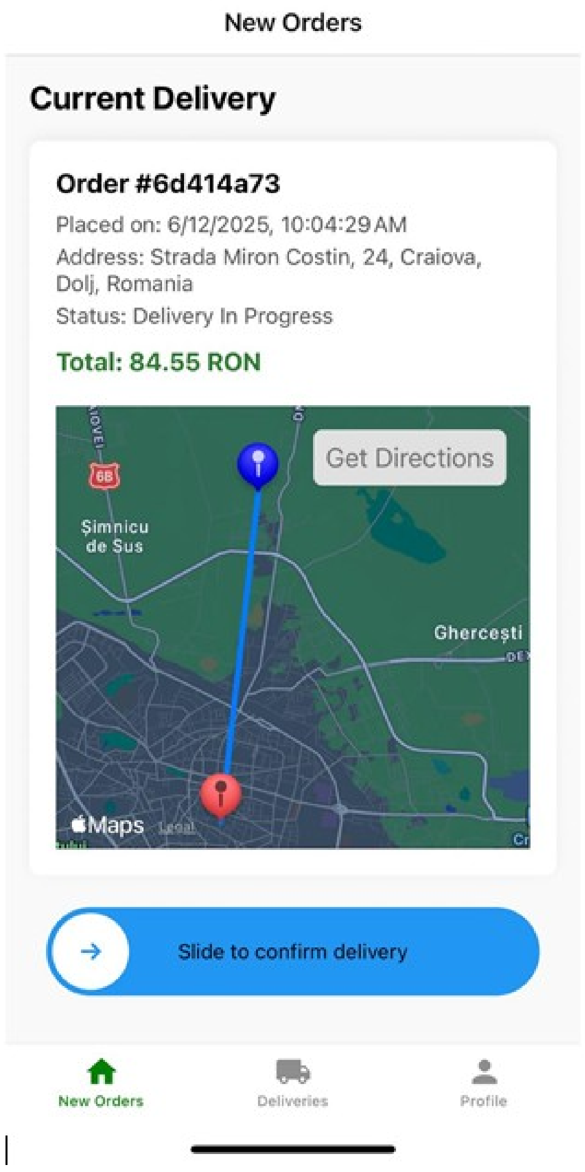
Figure 27: Place Order React Native Maps

While implementing mapping functionalities, it's notable that an API key is not explicitly required for these native integrations. This is because the Native Stack Navigator leverages the underlying operating system's map capabilities directly (such as Apple Maps on iOS and Google Maps services on Android, where applicable via native module configuration), which often handle API key provisioning implicitly or through platform-level developer accounts, simplifying the setup process for basic map display.

As presented in the above image (Figure 27: Place Order React Native Maps) the application utilizes advanced geospatial capabilities to enhance the user experience, particularly in address selection. Beyond simply displaying a map, users are empowered to directly select their precise location by dropping a pin on the map interface.

Following the pin placement, the application automatically performs a **reverse geocoding operation.** This critical process converts the selected geographic coordinates (latitude and longitude) into a human-readable street address. Specifically, the code ***await Location.reverseGeocodeAsync({ latitude, longitude });*** is used to fetch the corresponding address details directly from the pinpointed location. This seamless integration ensures that users can accurately define their delivery or service point without manual address entry, significantly streamlining the ordering process and reducing potential errors associated with incorrect addresses.

On the Delivery Order screen, the React Native Maps component serves a different, but equally vital, role for delivery personnel. It displays the optimal delivery route, shows the current location of the delivery driver, and helps visualize the customer's delivery address. This functionality is essential for efficient logistics, enabling drivers to navigate effectively, track their progress, and ensure timely deliveries. The integrated mapping capabilities empower both customers with precise location selection and delivery staff with efficient route guidance, streamlining the entire order fulfillment process.


New Orders

Current Delivery

Order #6d414a73

Placed on: 6/12/2025, 10:04:29 AM

Address: Strada Miron Costin, 24, Craiova, Dolj, Romania

Status: Delivery In Progress

Total: 84.55 RON

Figure 28: Delivery Order React Native Maps

The provided image illustrates the Delivery Order screen, a critical interface designed to facilitate efficient order fulfillment. On this screen, the integrated map prominently displays essential navigational information through two distinct markers: one clearly indicating the precise delivery destination, and another representing the real-time current location of the delivery personnel. This visual clarity is crucial, allowing drivers to immediately ascertain their position relative to the drop-off point and helping them orient themselves for the delivery task.

Complementing this visual guidance, a dedicated "Get Directions" button is strategically positioned on the interface. Activating this feature seamlessly integrates with the device's native mapping applications, launching either Apple Maps (for iOS users) or Google Maps (for Android users) with the route already pre-configured from the driver's current location to the designated delivery address. This direct deep-linking capability eliminates the need for manual address input into external navigation apps, providing immediate and streamlined turn-by-turn guidance, thereby significantly enhancing the efficiency and accuracy of deliveries.

Platform	Default Map Service / Behavior
iOS	Apple Maps
Android	Google Maps
Web	The core react-native-maps package does not function directly, requiring platform-specific solutions or aliasing (e.g., react-native-web-maps) for map display.

Table 4: React Native Maps Platform Compatibility

Currently, web access is limited due to the absence of a functional map view, as the underlying native map component lacks direct support for the web platform. Furthermore, upon initial attempts at integration, the web platform consistently crashed whenever React Native Maps was imported into a file.

The web platform's lack of support for the native map component necessitated a singular solution: implementing resolver logic within the **Metro bundler's configuration**. This redirection of module imports to a web-compatible alternative was crucial for restoring map functionality and stabilizing the web application.

5.7 Navigators

5.7.1 General

In the application, two distinct types of native navigators are employed to manage screen flow and user experience: the Tab Navigator and the Stack Navigator. These navigators are strategically utilized to provide intuitive navigation paths and a native look and feel across different sections of the application.

These navigators form the backbone of the application's overall structure and organization. They define how users move between different views, ensure a coherent user journey, and contribute significantly to the app's architectural clarity and maintainability by logically segmenting its various functionalities.

5.7.2 Stack Navigator

The Native Stack Navigator facilitates screen transitions within your application by layering new screens atop a navigational stack. It is pre-configured to deliver the familiar platform-specific user experience, with new screens sliding from the right on iOS and exhibiting a fade-in and scale animation from the center on Android. Additionally, iOS offers an option for screens to slide in from the bottom, adopting a modal presentation style. [13]

Crucially, this navigator leverages native navigation primitives—such as UINavigationController on iOS and Fragment on Android—for its underlying navigation operations. Unlike JavaScript-based stack navigators that re-implement animations and gestures, the Native Stack Navigator relies directly on these platform primitives, resulting in a more authentic native feel and optimized performance. Consequently, it is the preferred choice when prioritizing a native navigation experience and performance over extensive customization options. [13]

I used the stack navigator, mainly, for key transactional flows, specifically the cart and place order screen, and it is also employed by the profile screen. This navigator provides a robust capability for opening new screens with custom parameters while fully supporting native back gestures.

5.7.3 Tab Navigator

Upon successful login, each user role is presented with a customized bottom tab navigator. This navigator is specifically designed to provide intuitive access to the screens pertinent to their respective roles.

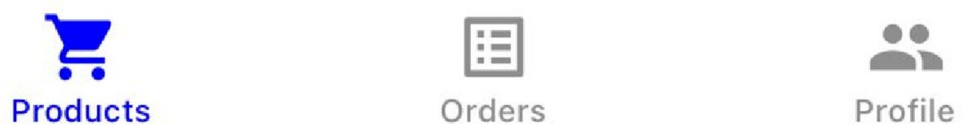


Figure 29: Bottom tab navigator

The administration user tabs:

- ➢ Products
- ➢ Sales Analytics
- ➢ Profile

The delivery user tabs:

- ➢ New orders
- ➢ Deliveries
- ➢ Profile

The customer user tabs:

- ➢ Products
- ➢ Orders
- ➢ Profile

By leveraging this type of navigator, I was able to craft an intuitive user interface that prominently displays key functionalities and offers expedited access. To complement this functionality, the design of the tabs and their associated icons within the bottom tab navigator is highly flexible. Their styling can be easily modified to align with specific branding requirements or evolving UI/UX preferences, ensuring a consistent and adaptable user interface across all roles.

5.8 Uploaded Images

In my application, a crucial optimization step is applied to every image uploaded by users before it is persisted in the database. Recognizing that high-resolution images can consume significant storage space and impact application performance (especially during retrieval and display), a backend resizing and compression process is implemented.

When a user uploads an image, the raw, full-size file is transmitted to the backend. Upon receiving the image, the backend then performs an intelligent resizing operation. The core principle of this resizing is to maintain the original aspect ratio of the image. This means that if an image is, for example, 1920x1080 pixels (a 16:9 ratio), it will be resized to a smaller dimension (e.g., 300x169 pixels) while strictly preserving that 16:9 ratio. This prevents distortion, stretching, or squishing of the image, ensuring that the visual integrity of the user's uploaded content is maintained.

Following the resizing, the image undergoes compression. This step further reduces the file size by optimizing the image data, often by reducing quality slightly in a way that is imperceptible to the human eye for most use cases, or by stripping unnecessary metadata. The combination of resizing and compression significantly lowers the storage footprint in the Microsoft SQL Server 2019 database and reduces bandwidth consumption during both upload and download operations. This proactive approach leads to faster loading times, a smoother user experience, and more efficient resource utilization for the application as a whole.

This optimized backend image processing is specifically applied to critical visual assets within the application, including both product images and user images. By consistently redimensioning and compressing these two types of images, the system ensures efficient storage and rapid loading times for frequently accessed visual content, directly benefiting the overall performance and user experience.

5.9 Other Native Components

5.9.1 Keyboard Avoiding View

In the application, the **KeyboardAvoidingView** component from **React Native** is strategically employed across various screens, particularly wherever user input fields might be present. This component is crucial for significantly enhancing the user experience by preventing the on-screen keyboard from obscuring active input fields or other vital user interface elements.

Without **KeyboardAvoidingView**, when the software keyboard appears, it often pushes up or covers content at the bottom of the screen, making it impossible for users to see what they are typing or to access buttons located below the input area. By wrapping content within KeyboardAvoidingView, the application automatically adjusts its layout—either by padding, adjusting height, or changing position—to ensure that the relevant input field remains visible and accessible above the keyboard. This is especially vital on screens such as login and registration forms, search bars, or any detailed forms where users enter text, ensuring a smooth and uninterrupted data entry process. Its implementation is key to maintaining the application's usability and responsiveness, adapting dynamically to the presence of the virtual keyboard.

5.9.2 Flat List

In the application, the **FlatList** component is extensively utilized for efficiently rendering long lists of data across various screens. As a core React Native component, FlatList is specifically designed to overcome the performance limitations of simpler list rendering approaches, such as mapping over an array within a ScrollView, especially when dealing with large or dynamically loaded datasets.

Its primary advantage lies in its virtualization capabilities. Instead of rendering all list items at once (which can lead to significant memory consumption and performance bottlenecks, particularly on mobile devices), FlatList only renders the items that are currently visible within the viewport, plus a small buffer of items just outside it. As the user scrolls, it intelligently unmounts items that move off-screen and mounts new ones coming into view. This optimization ensures smooth scrolling performance and reduced memory footprint, even for lists containing hundreds or thousands of items.

The FlatList component requires two main props: data, which is the array of items to be rendered, and renderItem, a function that specifies how each individual item should be displayed. Additionally, the keyExtractor prop is vital for providing unique keys to each item, enabling React Native to efficiently track and re-render items as needed. In the context of this delivery application, FlatList is leveraged in screens displaying product catalogs, order histories for customers, lists of available delivery tasks for drivers, and user management interfaces for administrators, ensuring a fluid and responsive user experience for all list-based content.

5.9.3 Slider

In the application, the react-native-slide-to-unlock component (imported as Slider) is strategically implemented on the Delivery Order screen to confirm the successful delivery of an order by the delivery personnel. This component provides an intuitive and secure mechanism for executing a critical action.

Mimicking the familiar "slide to unlock" gesture prevalent on mobile devices, this slider serves as a deliberate confirmation step. Instead of a simple tap, which could be accidental, the user is required to perform a full, intentional swipe gesture to finalize the order's delivery status. This design choice is paramount for preventing erroneous confirmations, ensuring that an order is only marked as delivered after explicit user intent. The successful completion of the slide not only triggers the backend update for the order status but also provides clear visual and tactile feedback to the delivery user, contributing to a more reliable and user-friendly experience for this irreversible action.

5.9.4 Charts

The **BarChart** and **LineChart** components from the **react-native-chart-kit** library are instrumental for presenting key operational data visually within the application. These charting tools transform complex datasets into intuitive graphical representations, enabling users to quickly glean insights and make informed decisions. To render these charts effectively, essential data points such as labels, datasets, and legend information must be explicitly provided to the component.

6 TESTING

Testing is crucial in software development, serving as the cornerstone for delivering a high-quality and reliable product. Its fundamental importance lies in its ability to proactively identify and rectify defects, errors, or unmet requirements early in the development cycle, which significantly reduces the cost and effort compared to post-deployment fixes. Beyond identifying bugs, comprehensive testing ensures the software meets all functional and non-functional specifications, including performance, security, and usability. A thoroughly tested application mitigates risks of failures, data loss, and vulnerabilities, thereby enhancing user satisfaction, building trust, and safeguarding the product's reputation by ensuring consistent, predictable, and robust operation.

6.1 General

The quality assurance process for the application primarily revolved around a focused manual testing strategy, carefully executed across its distinct platform implementations. This approach was chosen to ensure a robust user experience and functional integrity, given the interactive nature of a delivery application and the nuances of various device environments. The emphasis was on directly simulating user interactions and observing system behavior in real-time, allowing for immediate feedback and iterative refinement during the development cycle.

For the **iOS platform**, testing was conducted exclusively on a physical device. This decision was crucial for validating the application's performance in a genuine user context. A physical device allowed for accurate assessment of critical functionalities such as real-world GPS accuracy for location tracking and navigation (essential for both the Place Order and Delivery Order screens), responsiveness to native touch gestures and the seamless operation of device-specific features like photo gallery access for profile pictures or product uploads. Testing on a physical device provided an unfiltered view of the app's performance under actual network conditions and battery consumption, ensuring that the user experience was optimized beyond what emulated environments could offer.

On the **Android platform**, the testing paradigm leveraged the robust emulator provided by Android Studio. This setup offered distinct advantages, primarily the ability to simulate a wide array of device configurations, including different screen sizes, resolutions, and Android operating system versions. This facilitated comprehensive UI/UX testing across a diverse virtual device list, ensuring the application's responsiveness and visual consistency on various Android form factors. The emulator also proved invaluable for rapid debugging and iterative testing cycles, as it allowed for quick deployment and observation of changes without the need for constant physical device handling. While highly effective for functional and layout verification, the emulator's performance might not always perfectly mirror that of a physical device under extreme loads.

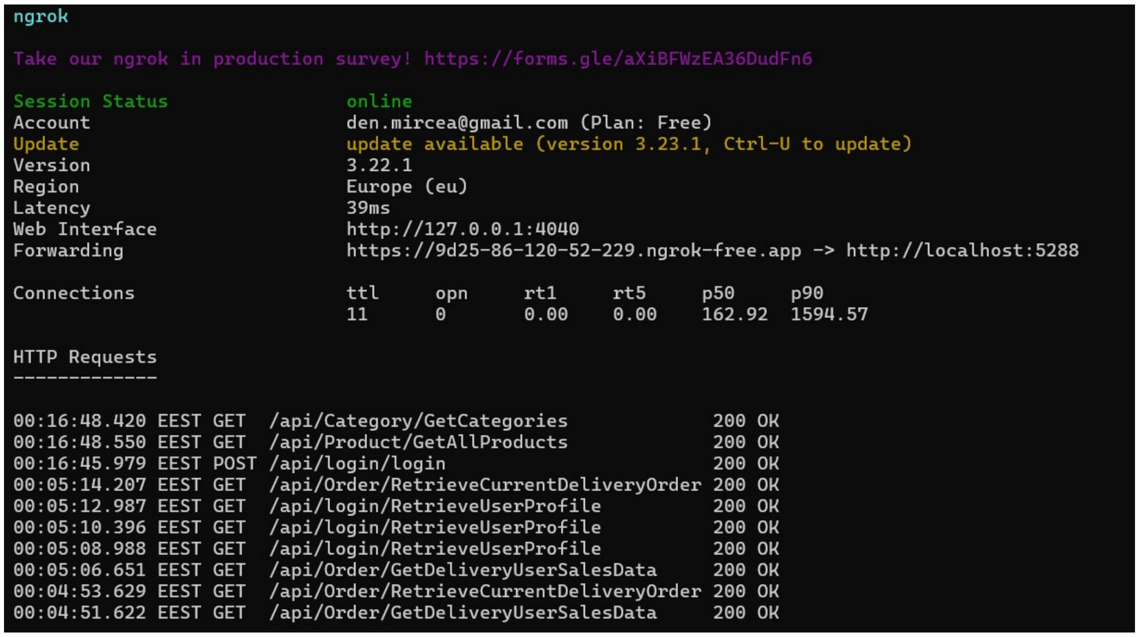
Finally, for the **web platform**, testing was performed directly within a standard web browser on my development machine. This environment provided immediate feedback loops, leveraging built-in browser developer tools for efficient debugging and inspection of the application's structure, styles, and network requests. Given the initial challenges with **react-native-maps** compatibility on the web, this direct browser testing was instrumental in verifying the successful implementation of the Metro bundler resolver logic that enabled map functionality on the other platforms. It also allowed for quick checks on responsive design, ensuring that the application's layout adapted correctly to different browser window sizes and aspect ratios.

Overall, this multi-platform manual testing approach was effective in identifying and rectifying functional and user experience issues specific to each environment. It provided a direct, human-centric validation of the application's readiness, allowing for a thorough assessment of interactions, visual fidelity, and the overall usability.

6.2 Using API outside the network

During the development and testing phases, a key challenge involved enabling the mobile application (running on physical devices or emulators) to communicate with the backend API, which was hosted locally on my development machine. To overcome network connectivity limitations, specifically when the mobile device was not on the same local network as the host computer, ngrok was strategically utilized.

Ngrok provided a secure tunneling solution, forwarding a local port from my development environment to a public, internet-accessible address. This public URL allowed the mobile application to send requests to the locally running API, effectively simulating a deployed backend environment without the need for actual server hosting. This capability was instrumental in comprehensively testing API integration and ensuring seamless data exchange between the frontend and backend, regardless of their immediate network proximity.

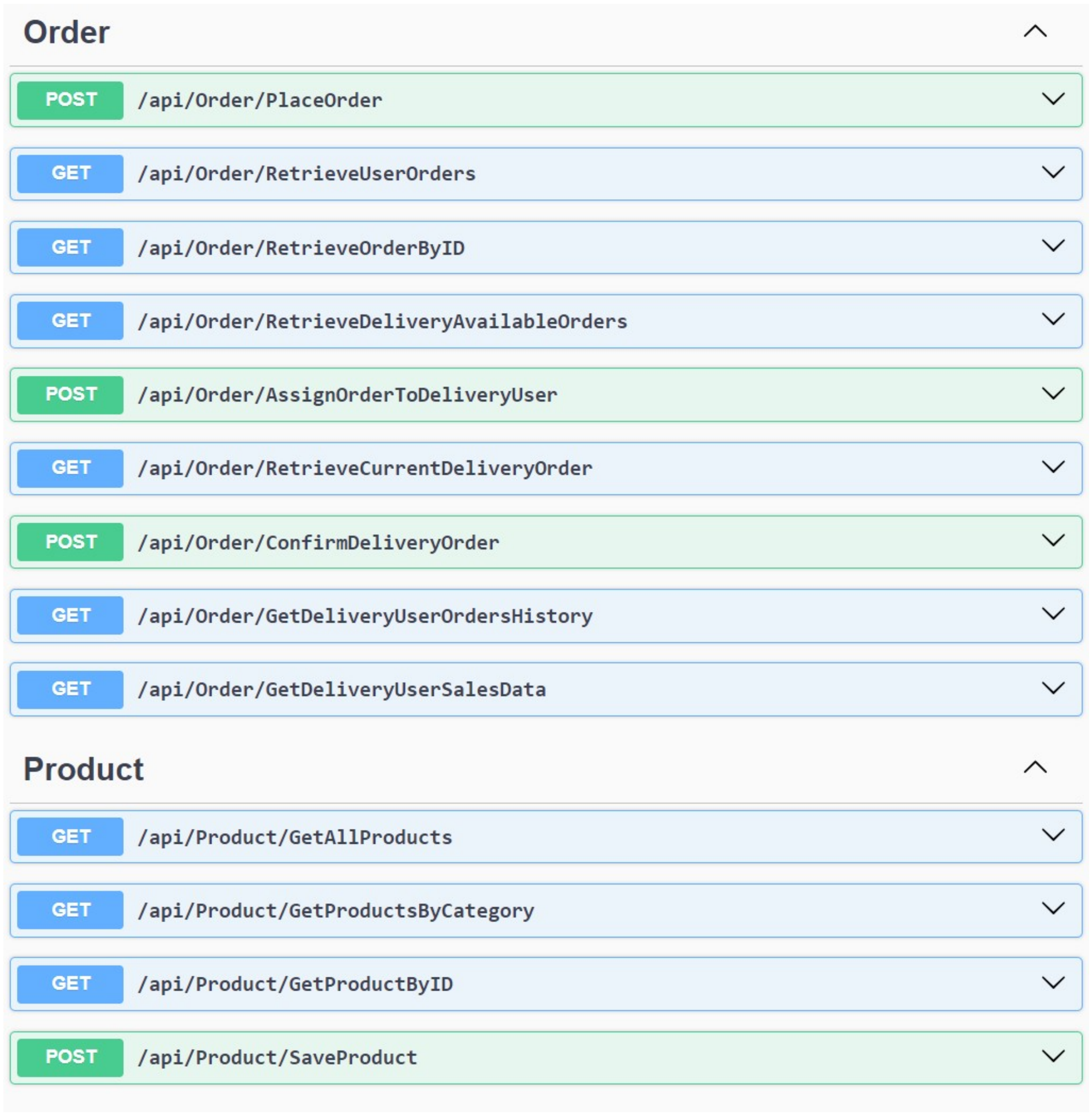


```
ngrok

Take our ngrok in production survey! https://forms.gle/aXiBFWzEA36DudFn6

Session Status                online
Account                       den.mircea@gmail.com (Plan: Free)
Update                        update available (version 3.23.1, Ctrl-U to update)
Version                       3.22.1
Region                        Europe (eu)
Latency                       39ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://9d25-86-120-52-229.ngrok-free.app -> http://localhost:5288

Connections                   ttl     opn     rt1     rt5     p50     p90
                              11      0       0.00    0.00    162.92  1594.57

HTTP Requests
-------------

00:16:48.420 EEST GET  /api/Category/GetCategories             200 OK
00:16:48.550 EEST GET  /api/Product/GetAllProducts             200 OK
00:16:45.979 EEST POST /api/login/login                        200 OK
00:05:14.207 EEST GET  /api/Order/RetrieveCurrentDeliveryOrder 200 OK
00:05:12.987 EEST GET  /api/login/RetrieveUserProfile          200 OK
00:05:10.396 EEST GET  /api/login/RetrieveUserProfile          200 OK
00:05:08.988 EEST GET  /api/login/RetrieveUserProfile          200 OK
00:05:06.651 EEST GET  /api/Order/GetDeliveryUserSalesData     200 OK
00:04:53.629 EEST GET  /api/Order/RetrieveCurrentDeliveryOrder 200 OK
00:04:51.622 EEST GET  /api/Order/GetDeliveryUserSalesData     200 OK
```

Figure 30: Ngrok port forwarding

6.3 API Testing

In parallel with the application's client-side testing, the backend API's functionality was verified using Swagger. This invaluable tool, which generates interactive API documentation from the codebase, allowed for direct interaction with individual API endpoints. By using Swagger UI, I could send requests to various API routes, inspect responses, and confirm data structures, ensuring that the backend logic performed as expected in isolation. This method facilitated rapid debugging and validation of the API's capabilities before and during its integration with the mobile application, providing a comprehensive view of the backend's reliability and adherence to specifications.



Order

POST /api/Order/PlaceOrder

GET /api/Order/RetrieveUserOrders

GET /api/Order/RetrieveOrderByID

GET /api/Order/RetrieveDeliveryAvailableOrders

POST /api/Order/AssignOrderToDeliveryUser

GET /api/Order/RetrieveCurrentDeliveryOrder

POST /api/Order/ConfirmDeliveryOrder

GET /api/Order/GetDeliveryUserOrdersHistory

GET /api/Order/GetDeliveryUserSalesData

Product

GET /api/Product/GetAllProducts

GET /api/Product/GetProductsByCategory

GET /api/Product/GetProductByID

POST /api/Product/SaveProduct

Figure 31: Swagger UI

7 CONCLUSIONS AND FUTURE IMPROVEMENTS

Developing this application presented a unique opportunity for significant professional growth and learning. This project allowed a deeper understanding of full-stack application architecture, from conceptualizing data flows to implementing robust backend logic. I significantly enhanced my proficiency in cross-platform development principles, navigating the unique challenges of building for diverse environments while ensuring a cohesive user experience. Furthermore, I gained invaluable experience in secure system design and efficient resource management, alongside hands-on practice with integrating complex third-party services and rigorous quality assurance methodologies. Every aspect of this endeavor, from initial design to final deployment considerations, served as a comprehensive learning experience.

This application successfully delivers a robust, multi-platform solution for managing product discovery, ordering, and delivery. By leveraging React Native, the development process achieved significant efficiency, allowing for a single codebase to target iOS, Android, and Web platforms. The strategic use of JWT tokens ensures secure and stateless authentication, while the backend-driven image resizing optimizes data storage and performance. The integration of native navigators (Stack and Tab) provides an intuitive and familiar user experience across mobile devices, and the sophisticated map functionalities, including reverse geocoding and native navigation app integration, streamline both order placement and delivery execution. Tools like ngrok and Swagger proved invaluable during development, simplifying API testing and ensuring backend reliability.

Looking ahead, several key areas have been identified for future enhancements to further elevate the application's capabilities and user experience:

Future Improvements:

- Real-time Order Tracking Enhancements: Implement more granular real-time updates for delivery status, potentially including live driver location on the customer's map view. This would require integrating WebSocket technology for persistent connections.
- Payment Gateway Expansion: Integrate additional payment methods (e.g., Apple Pay, Google Pay, local payment solutions common in Romania) to offer greater flexibility and convenience to customers.
- Advanced Recommendation Engine: Enhance the current product recommendation logic to incorporate more sophisticated algorithms, such as machine learning models, to provide highly personalized suggestions based on user behavior and preferences.

- Automated Testing Implementation: Introduce a comprehensive suite of automated tests (unit, integration, and end-to-end tests) to improve code quality, accelerate regression testing, and ensure application stability during continuous development.

- Offline Capability: Develop limited offline functionality for critical features, such as viewing cached menus or order history, to improve user experience in areas with poor network connectivity.

- Admin Dashboard Enhancements: Expand the administrative dashboard with more detailed analytics, custom report generation, and advanced tools for managing peak delivery periods or resolving complex order issues.

- Internationalization and Localization: Implement full multi-language support and region-specific localization for currencies, dates, and other relevant content, preparing the application for potential expansion beyond Romania.

8 BIBLIOGRAPHY

[1] M. Budziński, "What Is React Native? Complex Guide for 2024," 2024. [Online]. Available: https://www.netguru.com/glossary/react-native.

[2] "Learn once, write anywhere.," 2025. [Online]. Available: https://reactnative.dev/.

[3] "Get Started with React Native," 14 04 2025. [Online]. Available: https://reactnative.dev/docs/environment-setup.

[4] "Create amazing apps that run everywhere," 2025. [Online]. Available: https://docs.expo.dev/.

[5] R. B, "Expo: A Gateway to Simplified Cross-Platform App Development," 24 02 2024. [Online]. Available: https://medium.com/@talktorahul.b/expo-a-gateway-to-simplified-cross-platform-app-development-820184506c2f.

[6] "Learn .NET," 2025. [Online]. Available: https://dotnet.microsoft.com/en-us/learn.

[7] "Entity Framework documentation hub," [Online]. Available: https://learn.microsoft.com/en-us/ef/.

[8] R. Patel, "A Beginner's Guide to Entity Framework Core (EF Core)," [Online]. Available: https://medium.com/@ravipatel.it/a-beginners-guide-to-entity-framework-core-ef-core-5cde48fc7f7a.

[9] A. H. C. S. Rajul A, "Microsoft SQL Server," 15 03 2024. [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/SQL-Server.

[10] "Visual Studio 2022," [Online]. Available: https://visualstudio.microsoft.com/#vs-section.

[11] "GitHub," [Online]. Available: https://en.wikipedia.org/wiki/GitHub.

[12] Microsoft, "Migrations Overview," 1 12 2023. [Online]. Available: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli.

[13] donni106, "Native Stack Navigator," [Online]. Available: https://github.com/software-mansion/react-native-screens/tree/main/native-stack.

[14] Ö. Bilgili, "A Simple Guide to Using Maps in React Native," 27 Dec 2023. [Online]. Available: https://omurbilgili.medium.com/a-simple-guide-to-using-maps-in-react-native-f5f954ca7f16.

[15] S. Hansford, "What is ngrok?," [Online]. Available: https://ngrok.com/docs/what-is-ngrok/.

A. SOURCE CODE

The source code of the application is available as a GitHub repository:
https://github.com/denmircea/Licenta

B. SITE-UL WEB AL PROIECTULUI

The project does not offer a published application in a mobile store app, but can be downloaded from the following repository: https://github.com/denmircea/Licenta

C. CD / DVD

Autorul atașează în această anexă obligatorie, versiunea electronică a aplicației, a acestei lucrări, precum şi prezentarea finală a tezei.

INDEX