|  |  |  |
| --- | --- | --- |
|  | 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 ȘI TEHNOLOGIA INFORMAȚIEI |  |

Online Grocery Store with AI-Based Recommendations

Mircea-Denis Brașoveanu

COORDONATOR ȘTIINȚIFIC

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,

30.06.2025

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de [Calculatoare și Tehnologia Informației / Automatică și Electronică / Mecatronică și Robotică] | Aprobat la data de  …………………  Şef de departament,  Prof. dr. ing.  Nicolae Enescu/  Comin IONETE/  Dorian COJOCARU |

**PROIECTUL DE DIPLOMĂ**

|  |  |
| --- | --- |
| Numele și prenumele studentului/-ei: | Brașoveanu Mircea-Denis |
| Enunțul temei: | Online Grocery Store with AI-Based Recommendations |
| Datele de pornire: | [*Descrierea datelor inițiale de la care s-a început activitatea de cercetare/dezvoltare a tezei*] |
| Conținutul proiectului: | [*Descrierea succintă a conținutului fiecărui capitol al lucrării*] |
| Material grafic obligatoriu: |  |
| Consultații: | [*Periodice/zilnice/săptămânale/lunare*] |
| Conducătorul științific  (titlul, nume și prenume, semnătura): | Șef lucrări/Conferențiar/Profesor dr. ing. ................. |
| Data eliberării temei: | 15.10.2022 |
| Termenul estimat de predare a proiectului: | 01.06.2023 |
| Data predării proiectului de către student și semnătura acestuia: |  |

|  |  |  |
| --- | --- | --- |
|  | UNIVERSITATEA DIN CRAIOVA  Facultatea de Automatică, Calculatoare şi Electronică  Departamentul de [Calculatoare și Tehnologia Informației / Automatică și Electronică / Mecatronică și Robotică] |  |

**REFERATUL CONDUCĂTORULUI ȘTIINȚIFIC**

|  |  |
| --- | --- |
| Numele și prenumele candidatului/-ei: |  |
| Specializarea: | [*Denumirea oficială a specializării absolvite de candidat*] |
| Titlul proiectului: | [*Titlul lucrării*] |
| 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: [*se detaliază*] |

În urma analizei lucrării candidatului au fost constatate următoarele:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Nivelul documentării | | Insuficient  □ | Satisfăcător □ | Bine  □ | Foarte bine  □ |
| Tipul proiectului | | Cercetare  □ | Proiectare  □ | Realizare practică □ | Altul  [*se detaliază*] |
| Aparatul matematic utilizat | | Simplu  □ | Mediu  □ | Complex □ | Absent  □ |
| Utilitate | | Contract de cercetare □ | Cercetare internă □ | Utilare  □ | Altul  [*se detaliază*] |
| 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 both regular users and administrators. A key feature of the app is a 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 both the user and admin 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, admin panel, 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**

[1 INTRODUCTION 1](#_Toc200663253)

[1.1 Scope 1](#_Toc200663254)

[1.2 Motivation 1](#_Toc200663255)

[2 THEORETICAL CONTEXT AND TECHNOLOGIES 2](#_Toc200663256)

[2.1 Application Logo 2](#_Toc200663257)

[2.2 React Native 3](#_Toc200663258)

[2.3 Expo 5](#_Toc200663259)

[2.4 .NET 6](#_Toc200663260)

[2.5 Entity Framework 7](#_Toc200663261)

[2.6 SQL Server 9](#_Toc200663262)

[2.7 Visual Studio 10](#_Toc200663263)

[2.8 Visual Studio Code 11](#_Toc200663264)

[2.9 GitHub 12](#_Toc200663265)

[3 Objectives and benchmarks 13](#_Toc200663266)

[3.1 Problem Description 13](#_Toc200663267)

[3.2 Objectives 13](#_Toc200663268)

[3.3 Constraints 14](#_Toc200663269)

[3.4 Identifying User Problems 14](#_Toc200663270)

[3.5 Functional Requirements 14](#_Toc200663271)

[3.6 Non-Functional Requirements 14](#_Toc200663272)

[4 Software Architecture 15](#_Toc200663273)

[4.1 General Architecture 15](#_Toc200663274)

[4.2 Database Structure 15](#_Toc200663275)

[4.2.1 General 15](#_Toc200663276)

[4.2.2 Stored Procedures 17](#_Toc200663277)

[4.2.3 SQL Server Agent Jobs 17](#_Toc200663278)

[4.3 Backend Structure 18](#_Toc200663279)

[4.3.1 General 18](#_Toc200663280)

[4.3.2 Controllers 19](#_Toc200663281)

[4.3.3 Migrations 20](#_Toc200663282)

[4.4 Frontend Structure 20](#_Toc200663283)

[4.4.1 General 20](#_Toc200663284)

[4.4.2 API utils 21](#_Toc200663285)

[4.4.3 Metro bundler config 22](#_Toc200663286)

[4.4.4 App Configuration 22](#_Toc200663287)

[4.4.5 Package Configuration 24](#_Toc200663288)

[5 Software Implementation 25](#_Toc200663289)

[5.1 Security 25](#_Toc200663290)

[5.1.1 Password encryption 25](#_Toc200663291)

[5.1.2 JWT Token 26](#_Toc200663292)

[5.2 Use case diagrams 29](#_Toc200663293)

[5.2.1 Administrator 29](#_Toc200663294)

[5.2.2 Delivery User 30](#_Toc200663295)

[5.2.3 Customer/User 31](#_Toc200663296)

[5.3 Recommendation system 32](#_Toc200663297)

[5.3.1 General 32](#_Toc200663298)

[5.3.2 Database considerations 32](#_Toc200663299)

[5.3.3 Data computation 32](#_Toc200663300)

[5.4 Other functionalities 34](#_Toc200663301)

[5.5 React Native Maps 34](#_Toc200663302)

[5.6 Navigators 34](#_Toc200663303)

[5.6.1 General 34](#_Toc200663304)

[5.6.2 Stack Navigator 35](#_Toc200663305)

[5.6.3 Tab Navigator 35](#_Toc200663306)

[6 Conclusions and future improvements 37](#_Toc200663307)

[7 Bibliography 37](#_Toc200663308)

[A. Source Code 39](#_Toc200663309)

[B. Site-ul web al proiectului 40](#_Toc200663310)

[C. CD / DVD 41](#_Toc200663311)

[Index 42](#_Toc200663312)

**FIGURES LIST**

[Figure 1: Logo 2](#_Toc200663228)

[Figure 2: React Native 3](#_Toc200663229)

[Figure 3: Cross-platform apps with one codebase [2] 3](#_Toc200663230)

[Figure 4: React Native is being used in thousands of apps [2] 4](#_Toc200663231)

[Figure 5: Expo 5](#_Toc200663232)

[Figure 6: .NET 6](#_Toc200663233)

[Figure 7: Entity Framework 7](#_Toc200663234)

[Figure 8: SQL Server 9](#_Toc200663235)

[Figure 9: Visual Studio 10](#_Toc200663236)

[Figure 10: Visual Studio Code 11](#_Toc200663237)

[Figure 11: GitHub 12](#_Toc200663238)

[Figure 12: Database ERD (entity-relationship diagram) 16](#_Toc200663239)

[Figure 13: Stored Procedures Structure 17](#_Toc200663240)

[Figure 14: SQL Server Agent Job Configuration 17](#_Toc200663241)

[Figure 15: General Backend Structure 18](#_Toc200663242)

[Figure 16: General Frontend Structure 20](#_Toc200663243)

[Figure 17: Define JWT Claims 27](#_Toc200663244)

[Figure 18: Get UserID from JWT 27](#_Toc200663245)

[Figure 19: Get User Role from JWT 28](#_Toc200663246)

[Figure 20: Administator use case 29](#_Toc200663247)

[Figure 21: Delivery User use case 30](#_Toc200663248)

[Figure 22: Customer User use case 31](#_Toc200663249)

[Figure 23: Product recommendation example 33](#_Toc200663250)

[Figure 24: Weighted graph recommendation 34](#_Toc200663251)

[Figure 25: Bottom tab navigator 35](#_Toc200663252)

**TABLES LIST**

[Table 1: Password hashing 26](#_Toc200496210)

[Table 2: Password hashed with salt secret 26](#_Toc200496211)

[Table 3: Recommendation system main columns database 32](#_Toc200496212)

# INTRODUCTION

## Scope

The scope of this thesis is designing and creating 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, which was developed using a custom-made algorithm to provide recommendations for users based on the system orders.

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.

## 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’ve 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 make 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.

# THEORETICAL CONTEXT AND TECHNOLOGIES

## 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.

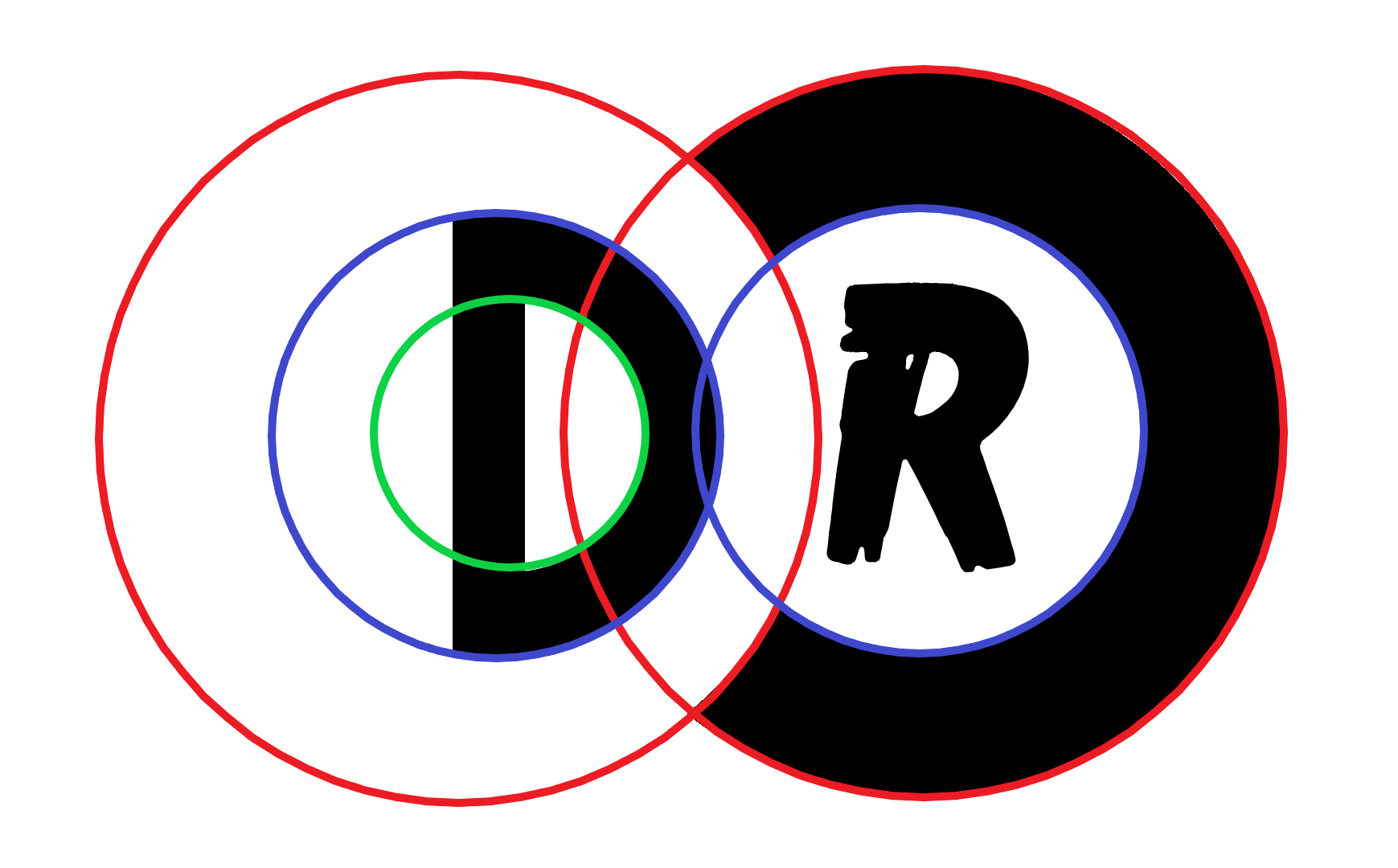
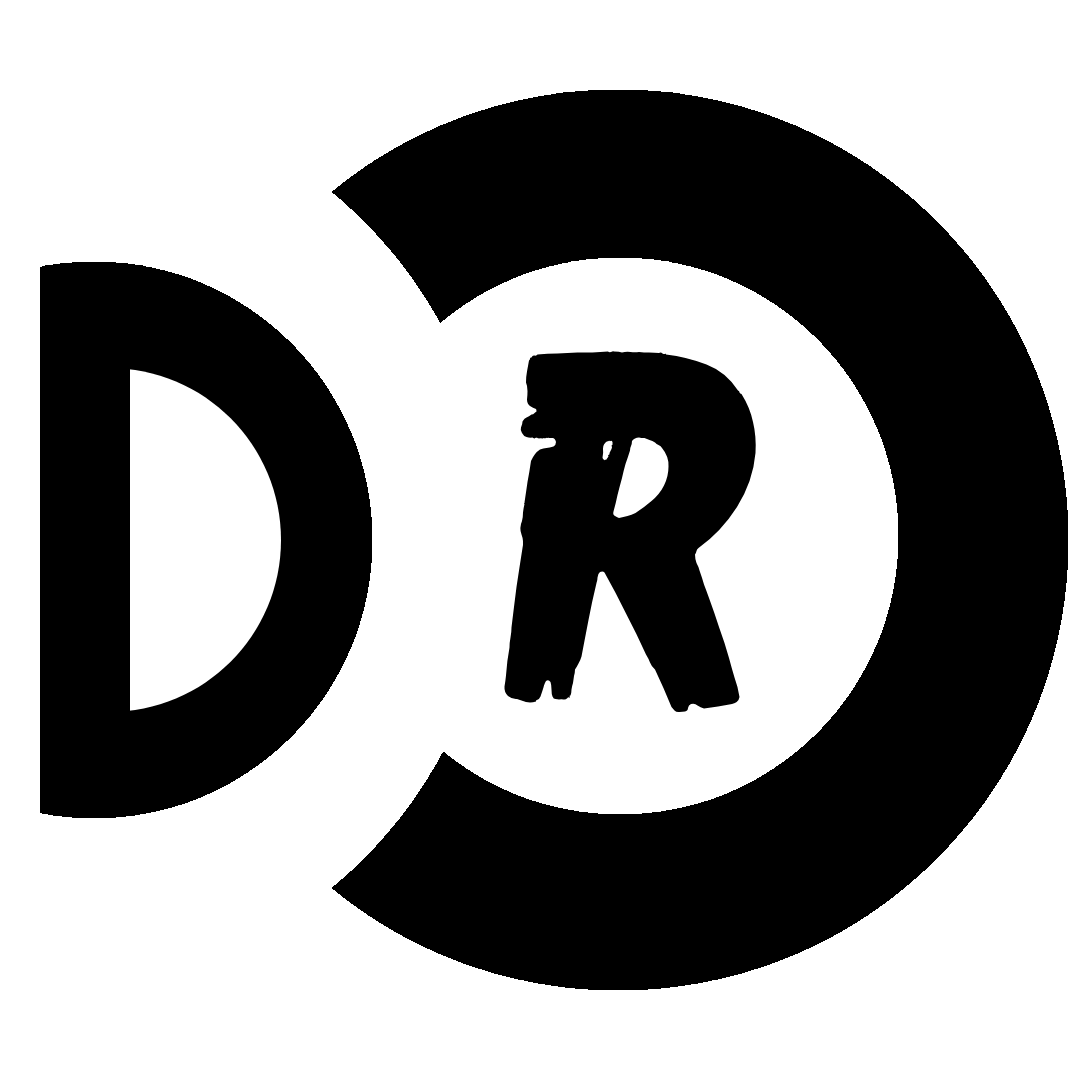


Figure : Logo

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.

## React Native

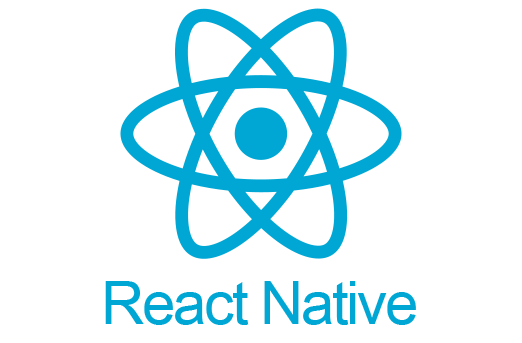


Figure : React Native

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.

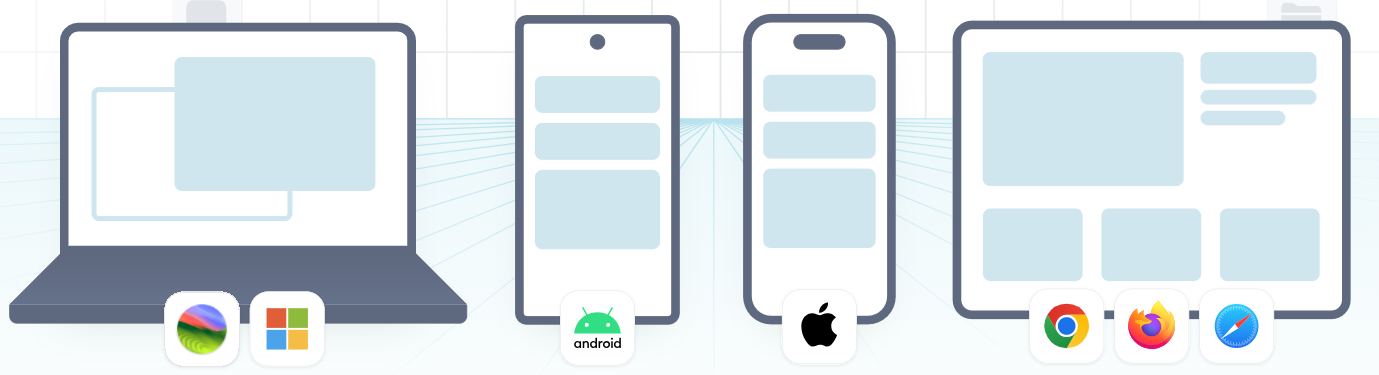


Figure : Cross-platform apps with one codebase [2]

“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]

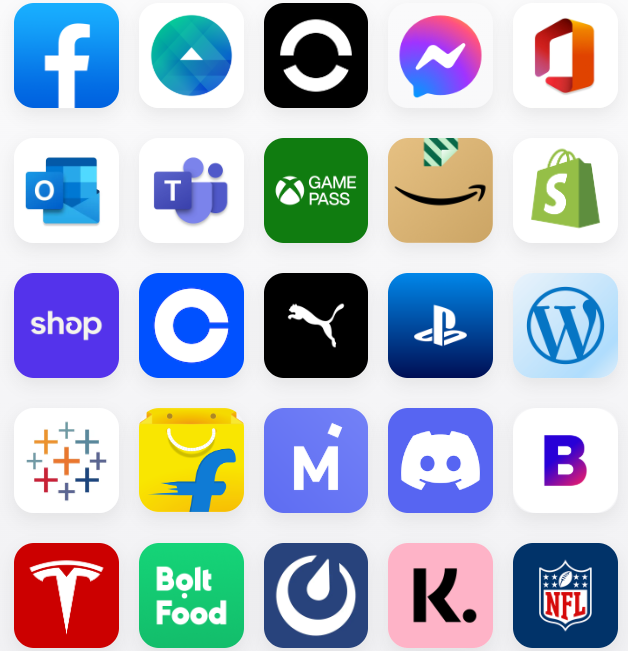


Figure : React Native is being used in thousands of apps [2]

“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]

## Expo

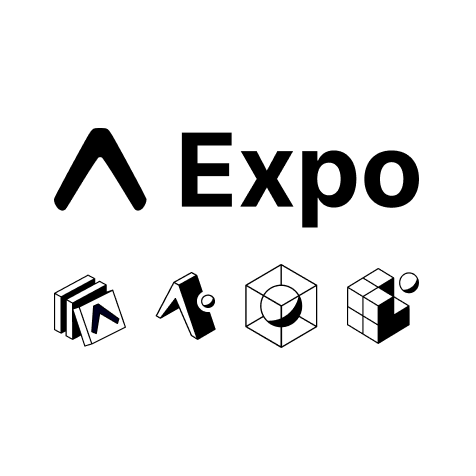


Figure : Expo

“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]

## .NET

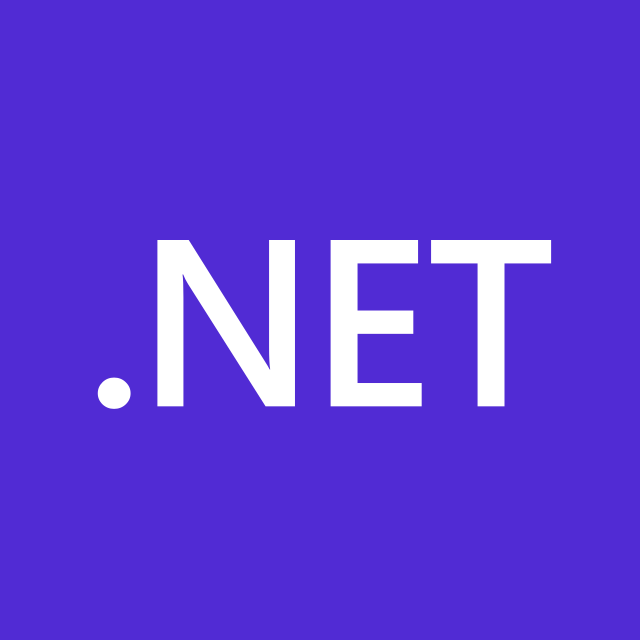


Figure : .NET

“.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'.

## Entity Framework



Figure : 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.

## SQL Server

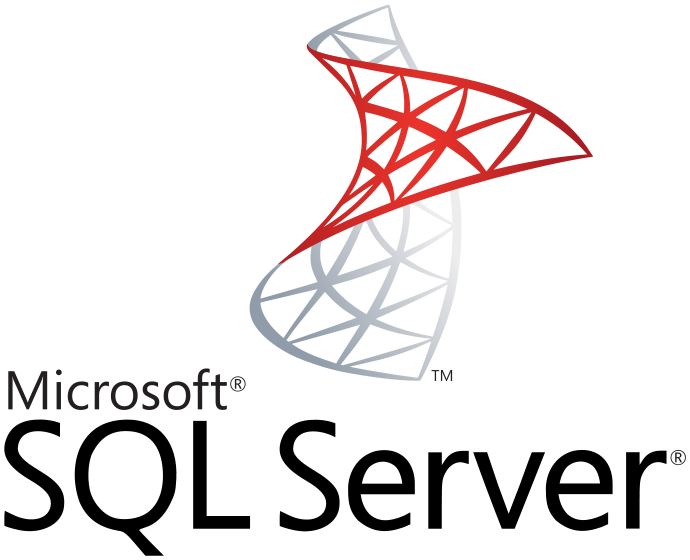


Figure : SQL Server

“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.

## Visual Studio

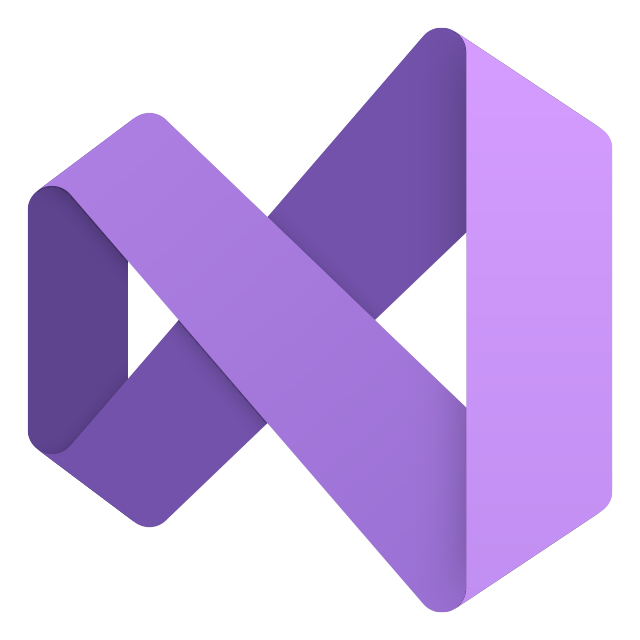


Figure : Visual Studio

“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]

## Visual Studio Code

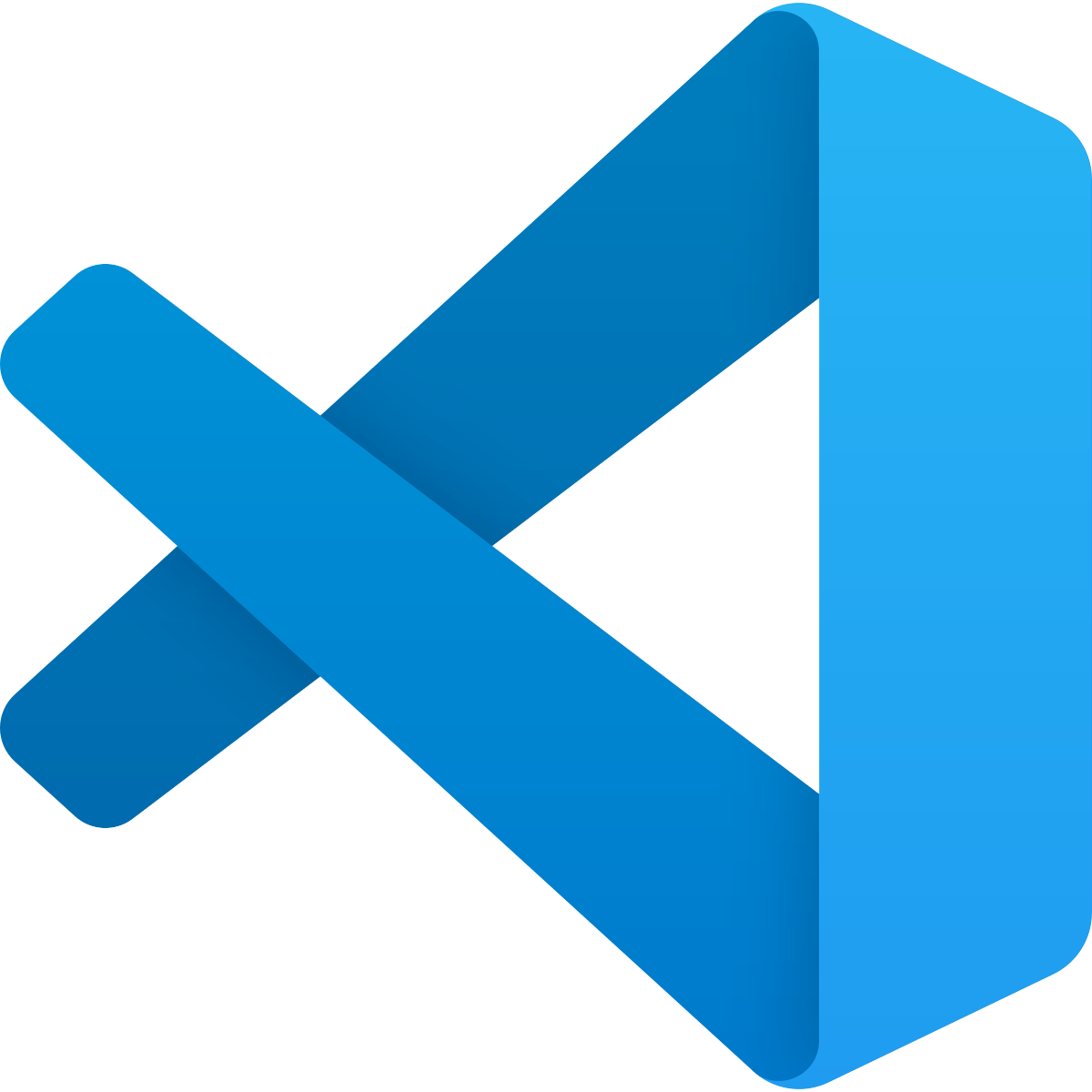


Figure : Visual Studio Code

“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.

## GitHub

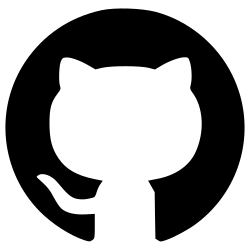


Figure : GitHub

“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.

# Objectives and benchmarks

## 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.

## 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.

## 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.

Another important rule is about how people use the app: everyone must create and keep an active account. This helps us offer a personalized experience and ensures your information is kept safe and secure within the system.

## Identifying User Problems

## Functional Requirements

## Non-Functional Requirements

# Software Architecture

## General Architecture

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.

## Database Structure

### 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 : Database ERD (entity-relationship diagram)

### 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.

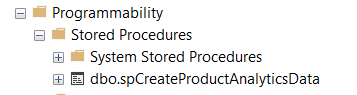


Figure : Stored Procedures Structure

### 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.

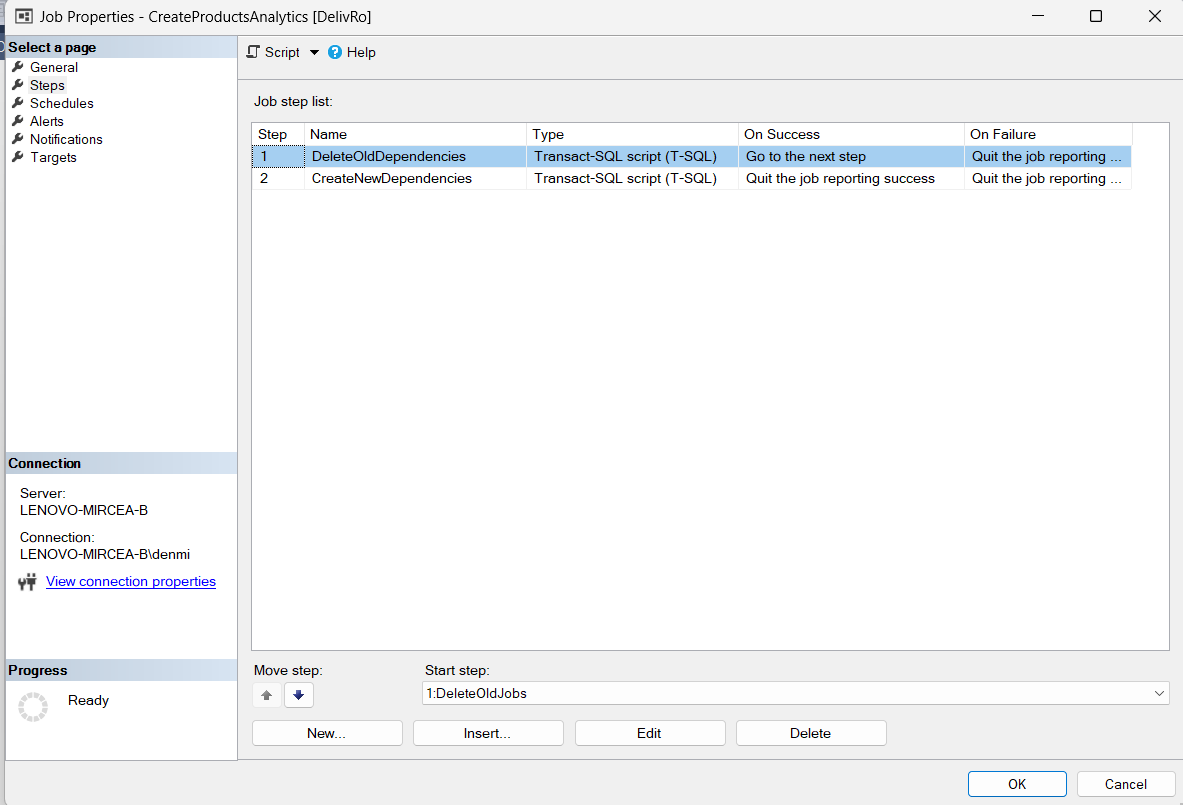


Figure : SQL Server Agent Job Configuration

## Backend Structure

### General

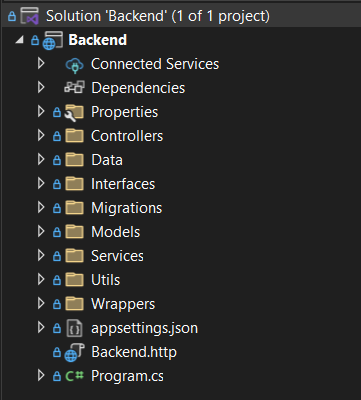


Figure : General Backend Structure

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.

### 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.

### 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]

## Frontend Structure

### General

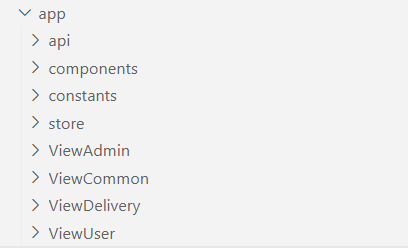


Figure : General Frontend Structure

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.

### 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.

### 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.

### 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"

}

}

}

### 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.

# Software Implementation

## Security

### 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 : 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 : Password hashed with salt secret

### 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)

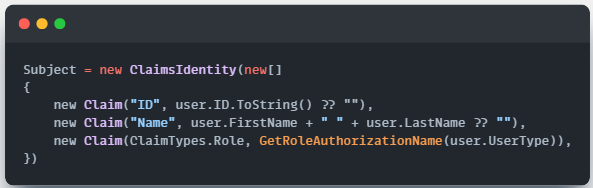


Figure : Define JWT Claims

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.



Figure : 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:

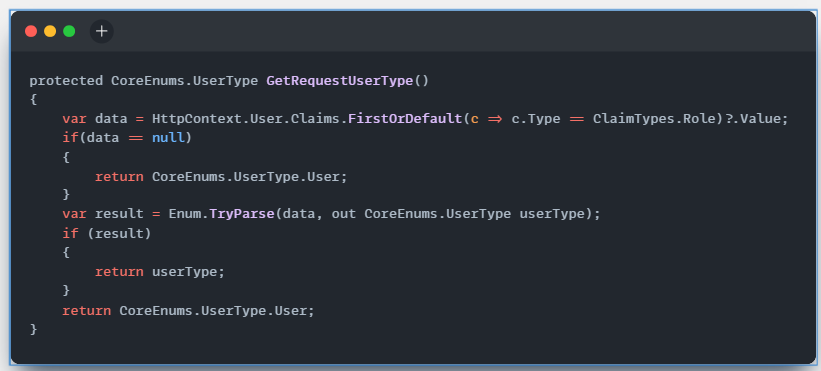


Figure : Get User Role from JWT

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).

## Use case diagrams

### Administrator

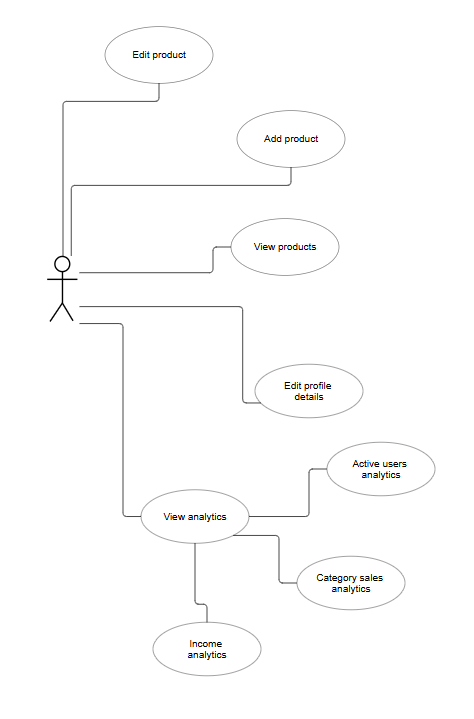


Figure : Administator use case

### Delivery User

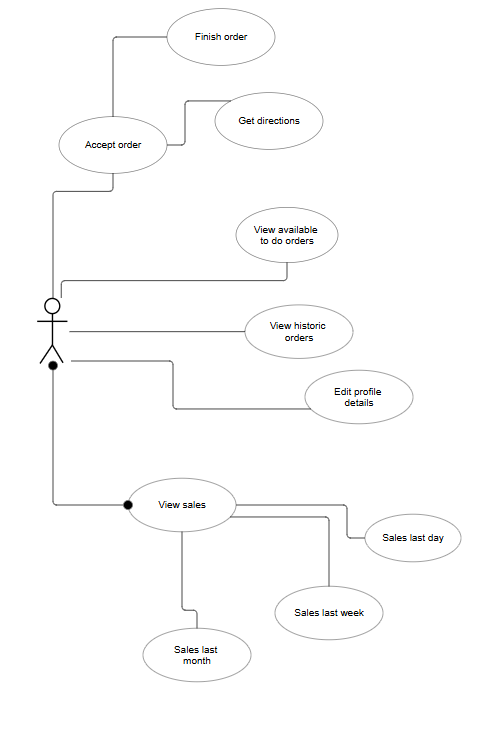


Figure : Delivery User use case

### Customer/User

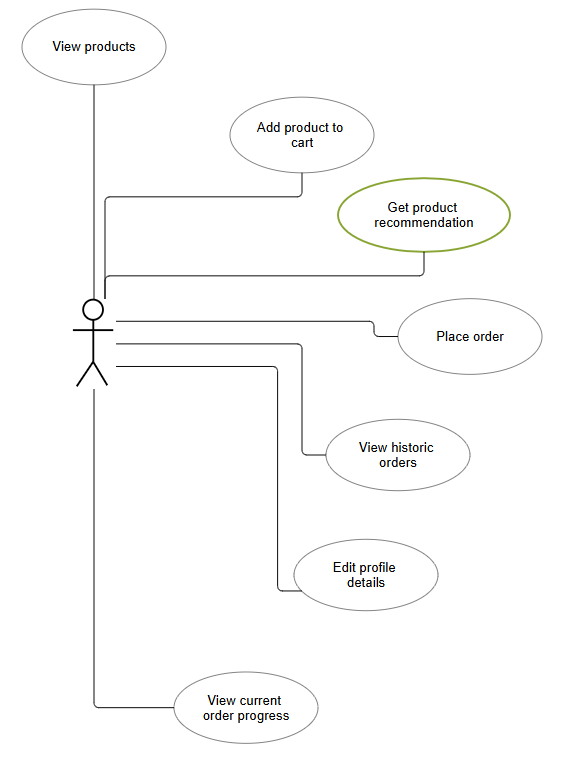


Figure : Customer User use case

## Recommendation system

### 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.

### 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:

|  |  |
| --- | --- |
| 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 : 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.

### 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.

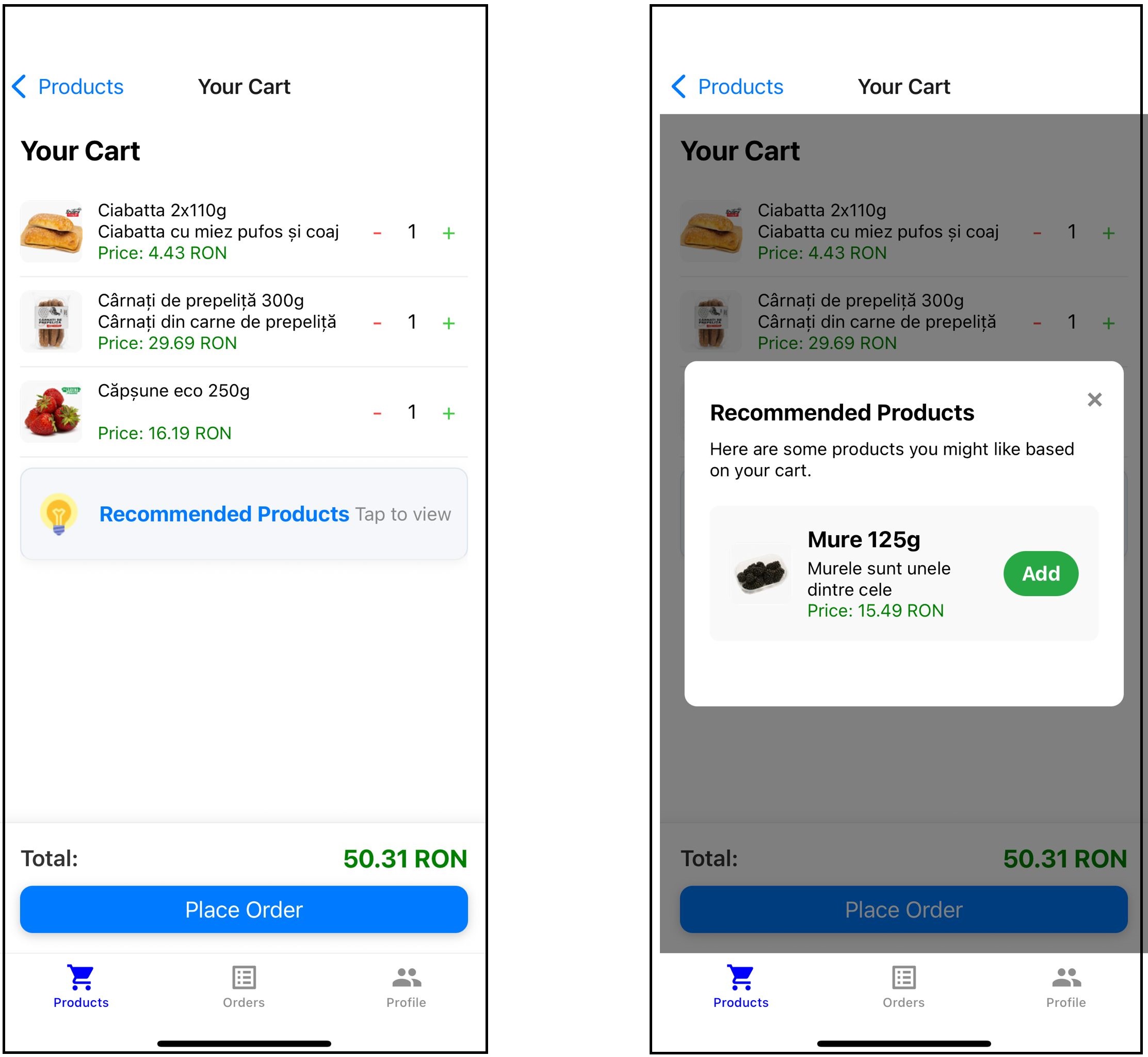


Figure : Product recommendation example

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:

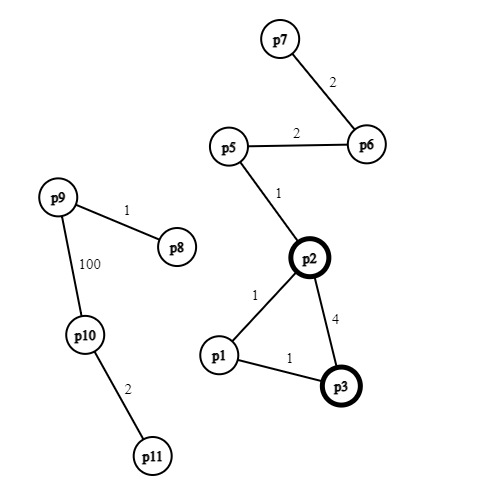


Figure : Weighted graph recommendation

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.

## Other functionalities

## React Native Maps

## Navigators

### 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.

### 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.

### 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.

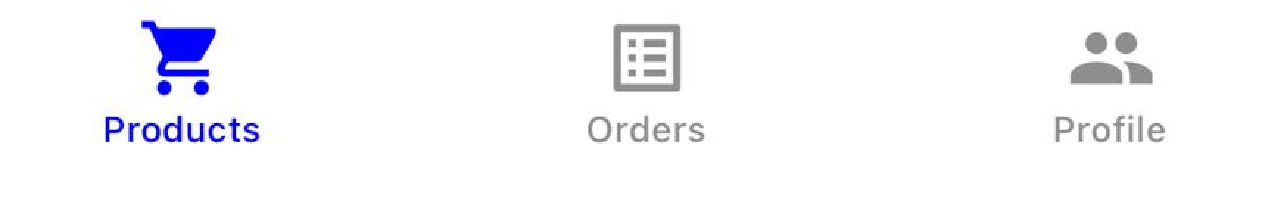


Figure : Bottom tab navigator

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.

# Conclusions and future improvements

# 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. |

# Source Code

# Site-ul web al proiectului

# CD / DVD

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



# Index

B

Bibliografie 9

C

CUPRINSUL xi

D

Dimensiuni 3

F

Figuri 4

Formulele matematice 4

I

Ilustrațiile 4

L

Legenda 6

LISTA FIGURILOR xii

LISTA TABELELOR xiii

R

Referințe web 10

S

Structura documentului 2

T

Tabele 5