Market Tracker

André Graça
Diogo Santos
Daniel Caseiro

Advisors: **Filipe Freitas**

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

July 2024

Instituto Superior de Engenharia de Lisboa

Market Tracker

47224 André da Silva Graça

A47224@alunos.isel.pt

48459 Diogo Santos

A48459@alunos.isel.pt

46052 Daniel André Caseiro

A46052@alunos.isel.pt

Advisors: Professor Filipe Freitas

Project report carried out under the Project and Seminar
Computer Science and Computer Engineering Bachelor's degree

July 2024

Resumo

A tecnologia evoluiu rapidamente nos últimos anos devido ao desenvolvimento contínuo da sociedade. Agora mais do que nunca, está cada vez mais difícil organizar as nossas compras diárias e, ao mesmo tempo, localizar os melhores produtos aos preços mais baixos, à medida que os supermercados continuam a crescer e a diversificar as suas ofertas de produtos.

De forma a resolver este problema, a nossa plataforma tenta simplificar a experiência de compras de supermercado. Similarmente ao conhecido serviço KuantoKusta[1], a nosso permite que os usuários encontrem os custos de produtos específicos em vários supermercados, exibindo as possibilidades disponíveis. Adicionalmente, oferecemos uma criação colaborativa de listas de compras, permitindo aos usuários compartilhar listas com amigos, familiares ou colegas. As listas ajudam a criar uma experiência de compra mais cooperativa e eficiente. Estas Listas podem ser editas e atualizadas em tempo real para atender às necessidades e preferências de cada usuário. Ao comparar a nossa plataforma com outros serviços semelhantes, nos destacamo-nos pela flexibilidade das listas. Os usuários podem encontrar os produtos desejados pelo melhor preço em várias lojas, economizando tempo e dinheiro.

O desenvolvimento desta plataforma inclui uma Web API (Application Programming Interface) [2] escalável para suportar implementações de futuras operações. A Web API está hospedada em um servidor que corre no protocolo HTTP[3]. Este lida com os dados que são necessários para manter a plataforma funcionando corretamente. A framework Android [4] é usada na interface do usuário (UI) que é mostrada ao usuário. Finalmente, uma base de dados relacional é usada para garantir a persistência de dados.

Palavras-chave: Compras de supermercado; Comparação de preço; Listas de compras colaborativas; Web API; framework Android; Persistência de dados.

Abstract

Technology has evolved rapidly in these past years due to the continuous development of society. Now more than ever, is getting harder to organize our daily purchases while locating the best products at the lowest rates as supermarkets continue to grow and diversify their product offerings.

In order to solve this problem, our platform tries to simplify the grocery shopping experience. Similar to the well-known service KuantoKusta [1], ours allows users to find the costs of specific products across multiple supermarkets, displaying the available possibilities. Additionally, we provide collaborative shopping list generation, allowing users to share lists with friends, family, or colleagues. The lists help to create a more cooperative and efficient shopping experience. These lists can be edited and updated in real time to suit the needs and preferences of each user. When comparing our platform with other similar services, we stand out in the flexibility of lists. Users may find desired products at the greatest price across multiple stores, saving them both time and money.

The development of this platform includes a scalable Web API (Application Programming Interface) [2] to support future feature implementations. The Web API is hosted on a server that runs on the HTTP protocol[3]. It handles the data that is required to keep the platform operating properly. The Android [4] framework is used in the user interface (UI) that is shown to the user. Finally, a relational database is used to guarantee data persistence.

Keywords: Grocery shopping; Price comparison; Collaborative shopping lists; Web API; Android framework; Data persistence.

Contents

List of Figures

Listings

Chapter 1

Introduction

The following chapter introduces the motivation behind the project and also the objectives and main functionalities that are to be developed.

1.1 Motivation

The motivation behind the development of MarketTracker arises from the shopping necessity of our technological society and the academic rigor required for our final bachelor's project. Nowadays, where time is the most precious resource, optimizing daily tasks such as grocery shopping has become increasingly vital. In response to this need, we created MarketTracker, a tool that will transform grocery shopping trips by enabling users to create economical, personalized shopping lists that save time and money.

Customers in today's market are faced with an enormous selection of products and varying pricing at various supermarkets. Considering all of this complexity makes it difficult to make wise purchases. Enabling people to easily go through this complexity is one of our main goals. MarketTracker encourages users to strategically plan their shopping visits by providing a comprehensive comparison of product pricing across multiple stores. Our goal in introducing this feature is to enable customers to make the most economical shopping lists that are customized to meet their individual requirements.

Another factor supporting MarketTracker's rise is its adaptability. With the use of a smartphone application, we enable consumers to easily track all relevant products and do real-time pricing comparisons while staying updated on their location. Users can dynamically modify their shopping lists to reflect changes in prices or product availability thanks to this mobile accessibility, greatly improving their entire shopping experience.

Moreover, MarketTracker promotes better user collaboration. Grocery shopping becomes a more productive and social activity when friends, family, or coworkers can collaborate to make and share shopping lists. Families or groups living together will find these lists especially helpful, as they are updated and modified in real time to guarantee that all parties are in agreement. This cooperative feature not only simplifies the purchasing procedure but also

strengthens the social ties between users.

By addressing these elements — necessity, technical adaptability, and academic difficulty — MarketTracker aims to offer a useful solution that boosts productivity, promotes collaboration, and saves money for its customers. Our ultimate goal is to improve people's quality of life by significantly changing the way they approach their weekly grocery shopping.

1.2 Goals

MarketTracker's main objective is to transform grocery shopping by offering a platform that simplifies and speeds up the procedure. Our goal is to encourage customers to make knowledgeable purchases by providing them with realtime pricing comparisons from several supermarkets. Users can now create cost-effective and customized shopping lists to make sure they get the most for their money.

In addition, in an effort to improve teamwork, we enable users to make and share shopping lists to friends, family, or coworkers, making shopping more effective and social. Another important goal is to make sure that our platform is easy to use and accessible on smartphones, enabling quick and easy mobile access to price updates and list updates.

Finally, the project aims to test and expand our technological abilities as an academic project by creating a scalable Web API, a reliable server architecture, and an intuitive Android-based user experience.

1.3 Main functionalities

There are a few functionalities that our service offers, with special focus on the following:

- Search and compare products' prices over time
- Forming shopping lists of products and get the best deals for each according to certain filters
- Manage products in each Operator's stores

1.4 Report Structure

This project report is divided into the following chapters, describing the technologies selected, how it's architecture was designed and it's implementation details:

- The project's service functionalities are described in Chapter 2, along with a breakdown of the various user roles.
- The project's architecture for the server and client components is summarized in Chapter 3, along with a general explanation of their functionalities. It also explains the database

model, emphasizing the relationships between various items, and talks about the chosen technologies and the reasoning behind their selection.

- In-depth information about the server implementation is provided in Chapter 4, which also covers the development process, error-handling techniques, authorization and authentication procedures, and layers and structure of the system.
- The client implementation is covered in Chapter 5, which also introduces the Android framework and its elements and details the submission and processing of API calls, including error handling. It also discusses the use of real-time updates and client-side authorization and authentication procedures.
- Chapter 6 describes the types of scripts that our team is willing to develop to integrate stores in our service.
- Finally Chapter 7 describes what was accomplished so far and the work we still have to do to complete this project.

Chapter 2

Functionalities

The following section presents all functionalities that the application will be capable of executing, types of users it will support and operations each can do.

Apart from a few operations accessible to unauthenticated users, authentication and authorization is required for the majority of them.

The system supports 3 user types: Client, Operator, and Moderator.

Unauthenticated users can only view products, their prices and reviews.

- **Client**: Capable of creating product lists and specifying the stores for price sourcing. Search for the best offers for each product on the list, leave product reviews, and manage their lists by adding/removing other clients, as well as modifying or deleting lists.
- **Operator**: Able to list and insert products into the system if they do not already exist, associate prices and promotions with them, and discontinue them as needed.
- **Moderator**: Able to modify details for all products, approve or reject new Operators and create new Stores, Companies, Cities and Categories in the system.

Our service offers the following functionalities:

- **Real-Time Price Comparison**
  - Compare prices of specific products across multiple supermarkets.
  - Access up-to-date price information to make informed purchasing decisions.
- **Personalized Shopping Lists**
  - Create customized shopping lists tailored to individual preferences and needs.
  - Optimize lists to find the best prices for desired products at different stores.
- **Collaborative Shopping Lists**
  - Share shopping lists with friends, family, or colleagues.
  - Update and modify lists in real-time, ensuring everyone is on the same page.
- **Mobile Accessibility**
  - Use the application conveniently on smartphones.
  - Adjust shopping lists on the go, reflecting changes in prices or product availability.
- **Product Tracking**
  - Track the availability and price changes of products over time.
  - Receive notifications for price drops or stock updates.
- **User-Friendly Interface**
  - Navigate the application with ease using an intuitive Android-based interface.
  - Easy access to functions improves the user experience as a whole.
- **Data Persistence**
  - Ensure data is stored securely and reliably through a relational database.
  - Maintain user preferences and shopping history for future reference.
- **Scalable Web API**
  - Support future feature expansions and integrations with a robust and scalable Web API.
  - Handle data processing and interactions efficiently to ensure smooth operation.

The figure 2.1 represents a possible *Use Case* of our Android Application.

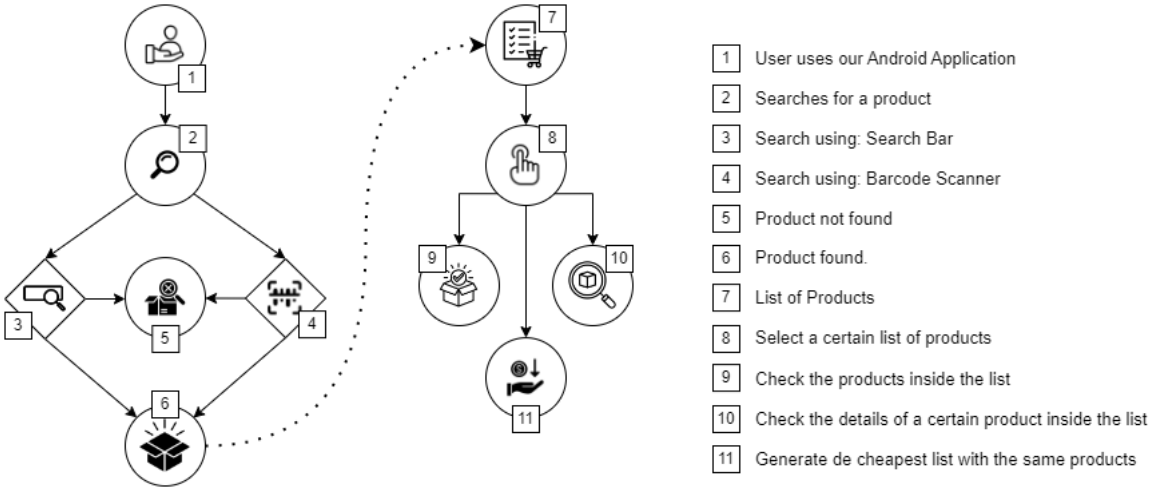

Figure 2.1: Client Use Case

The Figure 2.1 represents a possible use case of our Android application, representing the steps a user might follow when interacting with the app to search or compare products. Explanation of the use case step-by-step:

- **User Starts Interaction**: The user begins by opening and using the Android application (may login) (Step 1).
- **Product Search**: The user searches for a product either through a search bar (Step 2) or by using a barcode scanner (Step 3).
- **Search Results**:
  - If the product is not found, the user is informed (Step 5).
  - If the product is found, the user may add the product to a valid list of products (Step 6).
- **List of Products Actions**:
  - The user can select a specific list of products (Step 7).
  - The user can check the products within the list (Step 8).
  - The user can view details of a particular product in the list (Step 9).
- **Generate Cheapest List**: Finally, the user can generate a list of the cheapest options for the same products (Step 11).

Chapter 3

Architecture & Data Model

This chapter presents how our service and data is organized as well as how it's divided into different modules that interact with each other while showing which technologies we used.

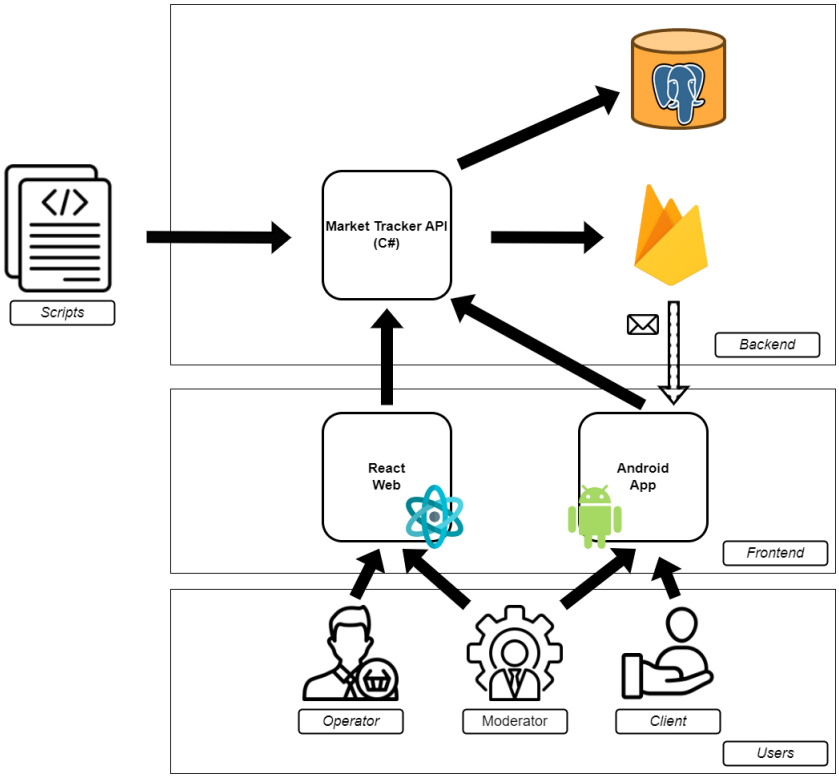

Figure 3.1: Service Architecture.

As figure 3.1 shows, the service is divided into 3 main modules:

- The **Scripts** that grab data from whichever source it might be and inserts them in our database through our API. They should be ran every day to keep the prices up to date.
- The **backend** is the major one, and it's in charge of implementing the service's logic, storing data, and providing an HTTP API that offers a set of capabilities to potential

users.

- The **Frontend** that serves the clients for the server application, allowing users to interact with it via a graphical interface. This technique was adopted in order to accommodate the possibility of numerous types of user engagement, such as mobile or desktop applications. Clients do not need to maintain knowledge about the system or implement its logic in this way because these pieces are already available in the server.

Users can then login in the app most suitable for their purpose. **Clients** have the Android app to make simple searches and additions to their shopping lists, **Operators** can choose the React App to make additions, updates or discontinuing products and **Moderators** can also choose React App to manage Operators and new admissions.

The decision to develop an Android application as the basis for our clients was due to its accessibility in a variety of contexts, such as shopping. The disadvantage of this choice, though, was that it could not support a single implementation, similar to that of a web application, for any device or operating system. However, modern technologies like Compose Multiplatform are starting to overcome this constraint by providing ways to make apps that are more cross-platform and flexible.

3.1 Technologies

This section describes the technologies used in this application on both the backend and frontend sides.

The server application, or backend, is an application that exposes a Web API and was developed using the **C# programming language** [5]. It utilizes the **.NET framework** [6] and the **Entity Framework ORM** [7] for database access, alongside the **ASP.NET MVC** framework for handling HTTP requests [8]. This API serves as the central point of contact for all clients, meaning that all clients (web, mobile, etc.) connect with the same server application. Customers interact with various endpoints to access, change, or create the required resources, where all the system's logic is implemented. An *endpoint* can be simplified as a pair consisting of a path and an HTTP method, which together correspond to a route on the server that performs actions based on its definition.

A **PostgreSQL** [9] relational database is utilized to store the data for this application. This Database Management System (DBMS) was chosen because of past expertise with it and the simplicity with which it can be hosted on various platforms.

We chose to develop an **Android** application because of its portable nature and convenience. We use the toolkit Jetpack Compose to design our UI with **Kotlin** [10] as it is the new modern way recommended by Google [11]. The Android app also uses **Firebase Cloud Messaging**

[12] for real-time notifications and updates, ensuring users receive timely information.

Additionally, we wanted to develop a frontend Portal website for Operators to manage their products that consists of a web application designed to create a User Interface (UI) to access the resources made available by the backend. The client was built with **React** [13], a library for designing user interfaces. With this, we could build a Single Page Application (SPA), which means it consists of a single HTML document that renders different pages over the application's lifespan, depending on the path specified or user interactions. An SPA is chosen over a Multi-Page Application (MPA) because an SPA is faster, as most of its resources are loaded only once.

For testing the backend, **Moq** [14] was used for mocking dependencies, and **FluentAssertions** was utilized to enhance the readability and maintainability of tests by providing a more intuitive way to write assertions.

3.2 Data Model

This section shows how we organize data in our Postgres database so we can perform multiple queries to return information through our API.

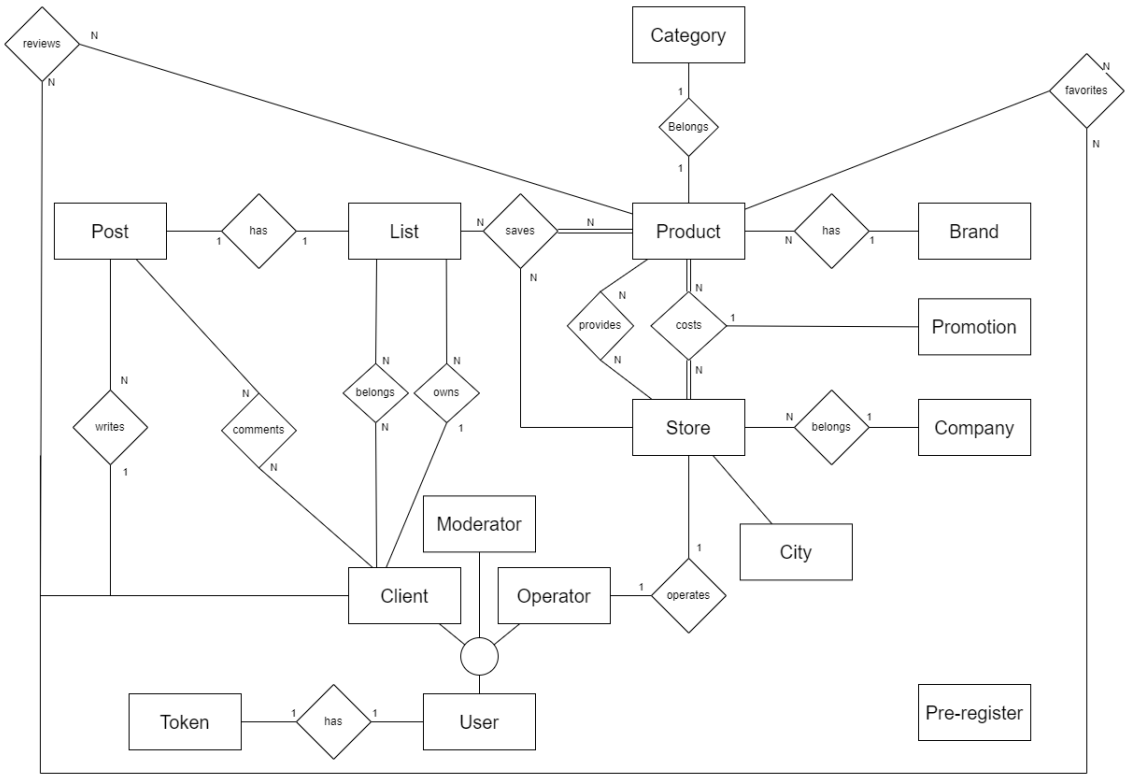

Figure 3.2: EA Model (Simplified).

The figure 3.2 presents a simplified Entity-Association (EA) model starting with the User

table, which includes Clients, Moderators, and Operators. Each user has a unique Token. Clients write and comment on Posts, which belong to Lists that they manage and own. Moderators review Posts. Lists save Products, which are categorized and branded. Products are also associated with Promotions and provided by Stores. Stores belong to Companies and operate in Cities. Each store can be online or a physical one, if the city foreign key is null it means the store is online and therefore there is not city associated with.

We tried our best to both save **space** and **time** when storing and querying prices information correspondingly, to try to give the clients the best user experience. Its worth noting the following aspects:

- **Prices** - Saving prices in the "costs" relation/ N-N table allows us to not only have multiple stores with different prices but to have an existing promotion (or not) to that price, this way also allows us to store different prices for the same product in a store over time as we have an attribute for it.
- **Availability** - The "provides" relation/table makes it more easy to query the availability of a product in a store and store the last time the its price was updated. The reason we don't store this date in the price itself is because if a price value for a product doesn't change, we will not store another equal entry, we just change its last update date in the previously mentioned table.
- **List entries** - Following this logic, we can't have lists' entries associated with a price because it can change over time, to overcome that we associate each entry to a product in a store. If the list it belongs is archived, the shopping list entry will be built with the price of that product in that date, otherwise it will have the current price.

This model efficiently organizes the relationships between users, their activities, and the products and stores they interact with. Additionally, the Pre-register, while unconnected, represents potential future operators in the context of our application.

Chapter 4

Server Implementation

In this chapter it will be described the server's implementation, each layers objective and how they accomplish it.

The server is the applications component that exposes a web API that allows to maintain and provide information required by the client making a request to the server. It manages all of this through three distinct layers:

- **Controller** - responsible for extracting the data necessary from the request to complete the action correspondent to the endpoint associated with the controller;
- **Services** - processes and validates the request, implementing the API logic;
- **Repository** - Responsible for verifying the existence of the resource, inserting, updating or removing data.

4.1 Dependency Injection and Inversion

Dependency injection is a widely recognized approach that facilitates the creation of independent and interchangeable modules. It is a pattern where dependencies are not directly created or required inside a module, but instead passed as parameters while dependency inversion suggests that high-level modules should not depend on low-level modules, but both should depend on abstractions.

The process of dependency injection typically begins in our entry point, such as Program.cs. We invoke an extension function of our services, which adds all necessary dependencies. At this stage, we also determine which data storage the server will use.

```
private static void ConfigureServices(WebApplicationBuilder
    builder){

    ...
    builder.Services.AddFirebaseServices(builder.Configuration);
    builder.Services.AddPgSqlServer(builder.Configuration);
    builder.Services.AddMarketTrackerDataServices();
}
```

Listing 4.1: Dependency injection services extension functions in Program.cs

The data services extension function is to add all corresponding data modules automatically through injecting anywhere they're used.

```
public static IServiceCollection AddMarketTrackerDataServices(
    this IServiceCollection services
)
{
    services.AddScoped<ITransactionManager,
        TransactionManager>();

    services.AddScoped<IProductService, ProductService>();
    services.AddScoped<IProductRepository, ProductRepository>();

    services.AddScoped<ICategoryService, CategoryService>();
    services.AddScoped<ICategoryRepository,
        CategoryRepository>();

    services.AddScoped<IBrandRepository, BrandRepository>();

    services.AddScoped<ITokenService, TokenService>();
    services.AddScoped<ITokenRepository, TokenRepository>();

    services.AddScoped<IUserService, UserService>();
    services.AddScoped<IUserRepository, UserRepository>();

    // more dependencies
}
```

Listing 4.2: Adding every data module using dependency injection and inversion.

By choosing scoped instances for all injected modules, each one will be created only once per HTTP request, ensuring efficiency but also consistency when accessing the data context in the repository layer.

By combining the principles of dependency inversion in our dependency injection approach, we ensure that our system remains modular and adaptable to future requirements and changes. This means that if we decide to change a technology within our system we only need to swap the module that implements the interface all the other modules depend on.

4.2 Error Handling

To maintain an API capable of handling all errors, we created the CustomExceptionHandler middleware so it catches the exceptions thrown by the Service layer, and uses the **Error** inside it to convert to the correspondent **Problem** and set it on the response body with the appropriate content type and status code.

```
if (exception is MarketTrackerServiceException serviceException)
        {
            Problem problem = serviceException.ServiceError switch
            {
                IAlertError alertError =>
                    AlertProblem.FromServiceError(alertError),
                ICategoryError categoryError =>
                    CategoryProblem.FromServiceError(categoryError),
                ICityError cityError =>
                    CityProblem.FromServiceError(cityError),
                ICompanyError companyError =>
                    CompanyProblem.FromServiceError(companyError),
                IListError listError =>
                    ListProblem.FromServiceError(listError),
                IListEntryError listEntryError =>
                    ListEntryProblem.FromServiceError(listEntryError),
                IPreRegistrationError preRegistrationError =>
                    PreRegistrationProblem.FromServiceError(
                     preRegistrationError),
                IProductError productError =>
                    ProductProblem.FromServiceError(productError),
                IStoreError storeError =>
                    StoreProblem.FromServiceError(storeError),
                IUserError userError =>
                    UserProblem.FromServiceError(userError),
                ITokenError tokenError =>
                    TokenProblem.FromServiceError(tokenError),
                IGoogleTokenError googleTokenError =>
                    GoogleProblem.FromServiceError(googleTokenError),
                _ => new ServerProblem.InternalServerError()
            };

            httpContext.Response.StatusCode = problem.Status;

            await
                httpContext.Response.WriteAsync(JsonConvert.SerializeObject(problem),
                cancellationToken);
```

Listing 4.3: CostumExceptionHandler.cs

The **Error** class contains relevant fields that any client may need to automate mechanisms for handling failed responses to requests, for example. It is encapsulated within a ServiceException so that it can be utilized later in the pipeline.

A **Problem** class represents an issue outlined in an RFC (Request for Comments) format [15]. It includes a unique identifier, facilitating bug reporting by API users and helping developers in tracing issues back to their source through logs or relevant data. Alongside this identifier, each Problem instance is assigned a 'type' field that uniquely identifies it within our system, ensuring clarity and specificity in categorizing and addressing different types of problems as outlined in the RFC. The Problem class has also 'title' and 'detail' fields, offering detailed descriptions of the problem as outlined in the RFC documentation. Finally, the 'data' field captures any relevant information associated with the problem, aligning with the RFC guidelines and providing valuable context for clients seeking to understand and respond to failed requests within the RFC framework.


```
{
    "Id": "88d89de6-b8b8-4f0f-80ce-a666d6c050b7",
    "Type": "https://markettracker.pt/probs/product-not-found",
    "Title": "Product not found",
    "Status": 404,
    "Detail": "Product with id 3 not found",
    "Timestamp": "2024-07-08T14:38:21.4636847Z",
    "Data": {
        "Id": "3"
    }
}
```


Figure 4.1: application/json+problem output example.

Each subtype of the Problem class is associated with a mapper function responsible for converting an Error object to the corresponding Problem instance. This mapping functionality is utilized within the middleware to facilitate the conversion process. By ensuring that the Service layer remains only aware of exceptions and Errors, and is unaware of HTTP specifics, we continue to follow the principle of **separation of concerns**. This separation allows for a clear distinction between different layers of the application, promoting maintainability and reusability.

4.3 Web API

The web API can process HTTP requests from the application's client side, extracting all data necessary to accomplish the operation associated with the *endpoint* specified by the request and sending the data to the service layer

When the service delivers the data that the client requested, the interface will map this information into an HTTP response. Each module in charge of this logic was referred to as a controller.

One of our controllers is the ProductController, here we have the operations to create, update, discontinue, search and delete products from our system. It depends solely on the ProductService.

```
[ApiController]
public class ProductController(IProductService productService,
    IProductPriceService productPriceService)
    : ControllerBase
{
    [HttpGet(Uris.Products.Base)]
    public async Task<ActionResult<PaginatedProductOffersOutputModel>>
        GetProductsAsync(
        [FromQuery] ProductsFiltersInputModel filters,
        [FromQuery] PaginationInputs paginationInputs,
        [FromQuery] ProductsSortOption? sortBy
    )
    {
        var paginatedProductOffers =
            await productService.GetBestAvailableProductsOffersAsync(
                paginationInputs.Skip,
                paginationInputs.ItemsPerPage,
                filters.MaxValuesPerFacet,
                sortBy,
                filters.Name,
                filters.CategoryIds,
                filters.BrandIds,
                filters.CompanyIds,
                filters.MinPrice,
                filters.MaxPrice,
                filters.MinRating,
                filters.MaxRating
            );
        return paginatedProductOffers.ToOutputModel();
    }
```

Listing 4.4: ProductController.cs

4.3.1 Authentication

Our Service provides 2 forms of user authentication, by using an email and password combination that is already registered in our database or by using Google OAuth 2.0 authentication [16].

- **Market Tracker** : By using Market Tracker authentication, the user needs to provide their email and password, which were registered previously in our database, if the email and password combination provided by the user is a valid combination, meaning a combination present in our database.
- **Google** : If the user opts by using a Google interface for authentication, the user

will need to provide the token generated by google authentication service, an IdToken, which will be verified by our service with the help of a external library.

In both cases, after the validation of the credentials provided by the user, our service will generate token object, which contains a value representing a session between the user and our service application, this token has a duration of 1 day and if it expires the user will need to provide his credentials in order to keep performing authorization required methods.

4.3.2 Authorization

Our Authorization middleware intercepts every HTTP request before it's routed to the Controller handler and verifies if the resource being accessed is protected.

```
var tokenValue = new Guid(
context.HttpContext.Request.Cookies[AuthenticationDetails.NameAuthorizationCookie]
    ?? string.Empty
);

var authenticatedUser = await
    tokenProcessor.ProcessAuthorizationHeaderValue(tokenValue);

if (authenticatedUser is null)
{
context.Result =
new AuthenticationProblem.InvalidToken().ToActionResult();
    return;
}

if (!authorizedAttribute.Roles.ContainsRole(authenticatedUser.User.Role))
{
context.Result =
new AuthenticationProblem.UnauthorizedResource().ToActionResult();
    return;
}

context.HttpContext.Items[AuthenticationDetails.KeyUser] =
    authenticatedUser;
```

Listing 4.5: Authorization Filter

Our middleware starts by verifying if the handler method has an Authorized Attribute Annotation and, by using .Net running environment, obtains this attribute from the method that will handle this request. If this attribute is not present, meaning that this resource is unprotected, our middleware forfeits the verification and allows total access to the user.

On the other hand, if the attribute is present, meaning the resource is protected, this interceptor will retrieve the token from the cookies of the request, if the latter isn't in the request, a Problem is inserted in the response body, indicating the token is missing in the request.

After retrieving the token, we need to verify the token validity, by verifying its format, its expiration date and if this one is associated to some user in our database, if any verifications fail, our middleware will return a problem, indicating why the token was considered invalid.

After authenticating the user, we pass to Authorization, our middleware will verify if the user has access to the resource, by searching for the user's role in the Authorized Attribute's roles field, if the user's role is not present, that means the user doesn't have the required authority to access or manipulate that resource, in this case, the authorization will fail and a problem will be returned, indicating that the user is not allowed to access that resource. If the user is authorized to access the resource an object, with the information of the user will be added to the request, so the handler method can use this object to perform the operations.

4.4 Service

The Service layer is responsible for receiving parameters extracted from the request by the Controller layer and processing them. This layer serves as the core of the application's logic, where data manipulation, validation, and enforcement of **business rules** take place. We also use this layer to build and return more complex objects called **Results** when a complex operation takes place and a single entity from domain isn't enough to return. In order to validate the request, one or more correspondent Repository modules are used to get the data from when we need it.

The ProductService, for example, depends on a few repositories to build its results and the INotificationService which currently uses Firebase as its implementation.

```
public class ProductService(
    IProductRepository productRepository,
    IBrandRepository brandRepository,
    ICategoryRepository categoryRepository,
    IPriceRepository priceRepository,
    IStoreRepository storeRepository,
    IClientDeviceRepository clientDeviceRepository,
    IPriceAlertRepository priceAlertRepository,
    INotificationService notificationService,
    ITransactionManager transactionManager
) : IProductService
{
    public async Task<PaginatedProductOffers>
        GetBestAvailableProductsOffersAsync(int skip,
         int take, int maxValuesPerFacet, ProductsSortOption? sortBy,
            string? searchName, IList<int>? categoryIds,
         IList<int>? brandIds, IList<int>? companyIds, int? minPrice,
            int? maxPrice, int? minRating, int? maxRating)
```

Listing 4.6: ProductService.cs

4.5 Repository

The Repository layer serves as a crucial bridge for data access within the application. It must be done in a way that the database and service layer are completely independent of one another. This layer keeps the usage of data from a database separate from the Service layer so that there is an abstraction regarding the database, meaning if there is a need to switch databases the only required thing to do is to switch out this module. By having a well known interface contract, we can abstract implementation details.

The use of a transaction manager in the service layer allows us to perform multiple operations in this layer from multiple different repositories using a single database connection.

Some modules in this layer are not limited to basic CRUD operations, they also handles complex queries to construct more elaborate Domain objects.

We also maintain error handling practices like verifying the resource existence before manipulation and handling null returns when necessary.

The ProductRepository must be the most complex and time consuming repositories of our service. It depends solely on the dataContext where we can have a real database serving as source of data or a mocked one.

```
public class ProductRepository(MarketTrackerDataContext dataContext) :
    IProductRepository
{
    private const double SimilarityThreshold = 0.9;

    public async Task<PaginatedFacetedProducts>
        GetAvailableProductsAsync(int skip, int take, int
        maxValuesPerFacet, ProductsSortOption? sortBy = null, string?name
        = null, IList<int>? categoryIds = null,
    IList<int>? brandIds = null, int? minPrice = null, int? maxPrice =
        null, int? minRating = null, int? maxRating = null, IList<int>?
        companyIds = null, IList<int>? storeIds = null, IList<int>?
        cityIds = null)
```

Listing 4.7: ProductRepository.cs

4.6 Matching a Product Across Stores

One of the first and main difficulties on implementing this service was to match the same product on multiple supermarkets. We considered name string matching patterns but this would be both inefficient and inconsistent. We needed something unique that every product had, we came to conclusion that the **EAN** [17] was perfect for this case because not only is it globally unique it is also mandatory for every sold product in the market to have one.

Every producy has an EAN that uniquely identifies it, even if 2 products are similar they will have different identifiers. An EAN can have 8 or 13 digits and can be described in the form of a barcode that is then print on the product's cover and shipped with it. In our Android app we can scan this barcode like in the supermarkets to obtain the product's EAN and redirect the user to the product's page.

Figure 4.2: Matching product's EAN across stores.

4.6.1 Products Searching Algorithm

We wanted to have a flexible searching mechanism that not only filters and returns a handful of products by their name, but also by their specifications and price. To do so, we implemented a bunch of SQL filters that may or not be applied to the search to return not only the product items as well as the searching facets.

4.6.2 Facets

Facets allow users to create categories on a select group of attributes so that they can refine their search. For example, on an index of products, helpful facets might be company and category. Market Tracker also calculates results for each facet. It allows us to display facets and facet counts so that users can filter results.

4.6.3 Difference between Filters and Facets

Filters and facets often get mixed up because there's much overlap, but understanding the difference is essential. Both help to restrict a search to a subset of results.

- Filters provide basic options to help narrow down search results.

- Facets help users refine their search using several categories simultaneously. Faceting is still filtering but displays facets to allow users to choose from a set of useful values, features, and counts.

4.6.4 Product Search by Name

To ensure flexibility in product searches, even when names are misspelled, our service uses two criteria to include products in the search results:

- The product's name contains the user's input.
- The product's name has a high similarity to the user's input based on trigram word similarity.

Products meeting either criterion are added to the paginated result list sent to the user.

4.7 Mobile Notifications

Sending mobile notifications to users is a crucial feature, as it keeps them informed about various price changes on products they chose. These notifications are done through the backend server using **Firebase Cloud Messaging** (FCM).

Firebase Cloud Messaging (FCM) is a cross-platform messaging solution provided by Google, designed for sending push notifications to mobile devices. It allows developers to send messages and notifications to devices running on Android, iOS, and web platforms. FCM handles the complexities of message delivery, such as queuing and retrying messages in case of failures, and ensures that notifications are delivered to users.

We use the **Firebase Admin.NET SDK** to communicate with FCM in order to accomplish this. Creating a Firebase singleton object that can connect to the FCM servers on our project's behalf is the process. Notifications to designated devices can be delivered after this is configured.

We identify each device with an ID generated by the Android client. Additionally, each device has the capability of generating an **FCM token**, which is then sent to and stored in the backend server. This allows us to later send a notification to the device using the respective FCM token.

4.8 Batch Requests

With HTTP batch requests [18], we can bundle multiple individual requests into a single HTTP request, significantly reducing the number of round trips between the client and server. This not only improves the overall efficiency of our system but also minimizes network overhead.

To maintain optimal performance and avoid overwhelming the system, we limit the number of operations allowed in each batch request to a maximum of 10. This ensures that each batch request remains manageable and that the server can effectively process and respond to the operations within it.

4.9 Tests

The purpose of this chapter is to clarify the process and actions performed to maintain the quality of work in this application. As part of the project's test-driven development methodology, every feature on the server is validated. Many unit test suites were developed during the development process to verify the code and assist in finding any flaws or mistakes that might have gone unnoticed.

The unit tests are designed to verify each case scenario for every operation within a module, following the Arrange-Act-Assert (AAA) pattern. In this pattern, the test is divided into three distinct phases:

1. **Arrange**: Set up the necessary preconditions and inputs.
2. **Act**: Execute the operation or behavior being tested.
3. **Assert**: Verify that the operation produced the expected results.

The unit tests are organized in a logical and systematic way, which makes them easier to read, thanks to the use of the AAA pattern.

The **Fluent Assertions** library was used during unit testing write to write assertions that are both simple and expressive.

Since the operation of our modules depends on other application layers, we included a popular library called **Moq**. By modeling dependencies and defining mock behaviors, we were able to guarantee that our unit tests stay isolated and concentrate only on the logic under test. We were able to simulate external dependencies with Moq, which made it easier to do thorough and trustworthy unit testing on our modules.

Using the **InMemory** extension function of the data context of the **Entity Framework**, the database was replicated into a working in-memory mock database to facilitate testing the Repository layer. This is particularly helpful because it frees the developer to concentrate on code rather than the state of the database.

Chapter 5

Client Implementation

The following chapter explains the implementation and thought process behind the client side, that is responsible for the visual element of the project.

The fundamental advantages of mobile apps—most notably, their portability and simplicity of use across a wide range of devices and scenarios—were the reason behind the decision of constructing a mobile application rather than a web application for our project's client-side solution. For example, the user could take his phone containing his shopping list in our app and do his shopping with our app along the way.

5.1 Android

Android is a popular mobile operating system developed by Google and was selected as the platform for our mobile application because it can be written in **Kotlin** programming language which we are already familiarized. We learned the fundamentals of Android apps development and the programming language behind it along this course so it was a unanimous decision to go with Android.

5.2 Jetpack Compose

Jetpack Compose is Android's recommended modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Just like React, Jetpack Compose allows us to define UI components as small, reusable pieces of code. These components are composed using function-like syntax, where properties are passed as parameters. Any modification to these properties triggers a re-composition of the affected component.

5.2.1 Recomposition and Reconfiguration

Compose is declarative and as such the only way to update it is by calling the same composable with new arguments. These arguments are representations of the UI state. Any

time a state is updated a **recomposition** takes place.

However, this isn't the only way a recomposition can happen. There can also happen what we call **reconfiguration**, which happens when a user changes some device configuration.

A reconfiguration triggers the recreation of an Activity, leading to the destruction of the current Activity instance and the creation of a new one. This process results in the reset of all non-persistent values to their initial content. Android developers need to keep in mind these events may happen at any time during the app's lifecycle.

A recomposition isn't always caused by a reconfiguration but a reconfiguration always causes a recomposition.

- **Recomposition**: re-running composables to update the UI Composition when data changes.
- **Reconfiguration**: a device configuration change that causes the Activity to be recreated and a new composition to be created.

This emphasizes the importance to have some way to protect our app from triggering **ViewModel** [19] functions that maybe do some API calls whenever the user rotates the screen for example, because it would overload the backend servers with unnecessary requests. To do this we create what we called **screen states**, this are classes that represent a state of a screen (e.g. loading, fetching, loaded), they will be discuss in detail later in the section 5.3.2.

5.3 Client Structure

We agreed to develop our Android app by adhering to best practices, implementing established design patterns, and maintaining a clean and organized architecture.

5.3.1 Architecture

The architecture of the client side of our service is also divided into different layers, each responsible for a certain purpose.

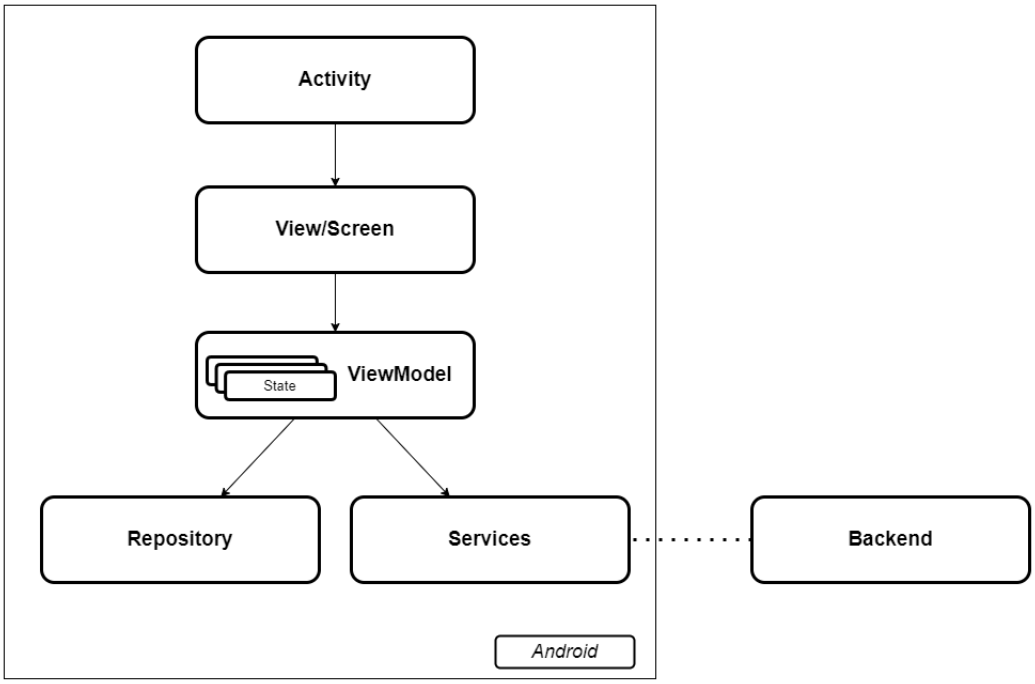

Figure 5.1: Client Architecture

The MainActivity is the entry point. Its where the NavGraph (where all main screens are hosted) is and the point from where we navigate to other activities too. Every screen has a viewmodel associated with it, to store a screen state (that can contain relevant screen data or simply represent a state with no data, e.g. loading state) and functions to manipulate it. A screen is then built by a lot of composables specifically designed for them.

5.3.2 Screen States

The screen states allow the ViewModel to verify if which phase the screen is currently, preventing it from being in an invalid state. This allows us to not load all products when the user causes a reconfiguration (triggering the ViewModel function) if they already loaded for example. We also have more control over the flow of data with a clear manipulation within certain states of the screen ensuring data is only available and displayed in certain stages.

```
sealed class ProductsScreenState {
    data object Idle : ProductsScreenState()

    data object Loading : ProductsScreenState()

    abstract class Loaded(
        open val productsOffers: List<ProductOffer>,
        open val hasMore: Boolean
    ) : ProductsScreenState()

    data class IdleLoaded(
        override val productsOffers: List<ProductOffer>,
        override val hasMore: Boolean
    ) : Loaded(productsOffers, hasMore)

    data class LoadingMore(
        override val productsOffers: List<ProductOffer>
    ) : Loaded(productsOffers, true)

    data class Failed(val error: Throwable) :
        ProductsScreenState()
}
```

Listing 5.1: ProductsScreenState.kt

We can also have extension functions only on certain states to perform the transition of states, ensuring only the intended flow between states occurs.

5.4 Dependency Management

Dependency injection in Android development is widely used and recommended by the platform's own documentation as it allows for greater code reuse, facilitates ease of modifying a particular element, and simplifies testing. However, as the size of the application grows, the number of dependencies to inject also tends to increase. Manually managing all dependencies makes this creation process difficult.

To continue benefiting from dependency injection without the need to manage them manually, the option of using a dependency management library was chosen. In the same section where dependency injection was recommended, a suggestion of the library to use was also presented, which is the **Hilt library** [20].

The Hilt library is based on the Dagger library and aims to bring, compared to Dagger, a simplified structure tailored for Android applications.

To declare a dependency as injectable, a class named AppModule was created (figure 5.3), which was annotated with @Module and @InstallIn. These annotations allow defining this

class as a module where Hilt should search for injectable dependencies and where to install this module. In this class, for each dependency, a function annotated with the @Provides annotation is created; the return type of this function indicates the type that should be constructed by the Hilt library and injected by it whenever an instance of this type is requested in a constructor or field annotated with @Inject.

```
@Module
@InstallIn(SingletonComponent::class)
class AppModule {
    private val Context.dataStore: DataStore<Preferences> by
        preferencesDataStore(
         name = MarketTrackerApplication.MT_DATASTORE
    )

    @Provides
    @Singleton
    fun provideDataStore(appContext: Application):
        DataStore<Preferences> {
         return appContext.dataStore
    }

    @Provides
    @Singleton
    fun provideAuthRepository(dataStore: DataStore<Preferences>):
        IAuthRepository {
         return AuthRepository(dataStore)
    }
```

Listing 5.2: AppModule.kt

5.5 Navigation

We decided to place a navigation bar at the bottom of the app, allowing users to easily navigate between different screens. This design choice enhances user experience by providing fast and intuitive access to all sections of our app. Each screen is a composable component hosted in a NavController and is associated with a specific route.

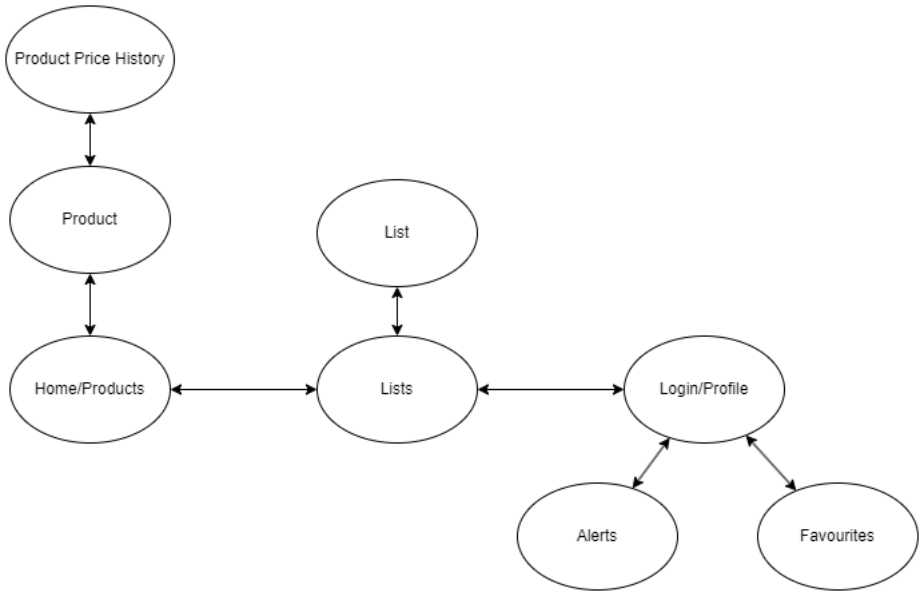

Figure 5.2: Navigation graph

We tried to follow the single-activity architecture because it simplifies navigation and state management, making the app more efficient and easier to maintain. However, some screens of our app warranted having a separate activity. This is particularly useful if we want to enable navigation to that activity from an external application. For example, if we support a feature in the future where a user can share a product via a link, the recipient could enter the ProductDetailsActivity directly.

```
NavHost(
        navController = navController,
        startDestination = Destination.HOME.route,
        modifier = Modifier.padding(paddingValues)
    ) {
        composable(Destination.HOME.route) {
            ProductsScreen(onProductClick, onBarcodeScanRequest, {
                changeDestination(Destination.PROFILE.route)
            }, authRepository, productsScreenViewModel)
        }

        composable(Destination.LIST.route) {
            ListScreen()
        }

        composable(Destination.PROFILE.route) {
            val authState by authRepository.authState.collectAsState()
            Log.v("User", "AuthState is $authState")
            if (authState.isLoggedIn()) {
                ProfileScreen(profileScreenViewModel)
            } else {
                LoginScreen(
                    onSignUpRequested = onSignUpRequested,
                    getGoogleLoginIntent = getGoogleLoginIntent,
                    loginScreenViewModel = loginScreenViewModel
                )
            }
        }
    }
}
```

Listing 5.3: NavGraph.kt

5.6 Authentication

Our Application provides 2 authentication interfaces.

- **Market Tracker** : Email and password combination need to be valid and correspond to an account in our application. They are sent to our backend service for verification.
- **Google** : The user will need to accept a consent screen of terms and conditions from Google, in order to proceed with the authentication.

In the first case, after the given credentials are verified in the backend server, the user will be logged in while on the latter the Google Identity API returns an IdToken that is sent to the backend and if valid, an account is created if none existed with that email and/or login token is sent in the response cookies.

After authentication, the user is redirected to their profile screen, if the authentication went successfully. If the authentication failed, a pop up model will appear indicating that authentication process failed.

5.7 Client Services

In our Android app we needed a way to communicate with the backend in order to obtain and submit data. The Service layer is responsible for this, by having a few sub-modules, each responsible for a handful of API requests.

To make http requests we use a library called **OkHttp** [21] and **Gson** [22] to deserealize the response bodies. Each service method needs to build the request in its own way, so we decided to create a class called *MarketTrackerService* to make it easier. This class holds singleton instances of OkHttp and Gson clients that are injected by Hilt. The function *requestHandler* is the on responsible for building the specified request and returning the result of the backend in a custom domain or model class.

```
abstract class MarketTrackerService {
    companion object {
        const val MT_API_URL = BuildConfig.API_URL
    }

    abstract val httpClient: OkHttpClient
    abstract val gson: Gson

    protected suspend inline fun <reified T> requestHandler(
        path: String,
        method: HttpMethod,
        body: Any? = null
    ): T {
        val url = URL(MT_API_URL + path)
        Log.v("requestHandler", "Request to API: $url")
        val request = Request.Builder().buildRequest(
            url,
            method,
            body
        )
```

Listing 5.4: MarketTrackerService.kt

Because of Kotlin reified keyword, we are able to check the return type of the generic function at runtime and map the response body to the specified class in our function as shown in figure 5.4. If the status code is larger or equal than 400 and if the response content-type is "application/json+problem" we map the body to Problem class that contains a detail message of why the error occured.

```
abstract class MarketTrackerService {
    companion object {
        const val MT_API_URL = BuildConfig.API_URL
    }

    abstract val httpClient: OkHttpClient
    abstract val gson: Gson

    protected suspend inline fun <reified T> requestHandler(
        path: String,
        method: HttpMethod,
        body: Any? = null
    ): T {
        val url = URL(MT_API_URL + path)
        Log.v("requestHandler", "Request to API: $url")
        val request = Request.Builder().buildRequest(
            url,
            method,
            body
        )
```

Listing 5.5: requestHandler function overriding callback's onResponse method

Now every module of the Service layer can extend MarketTrackerService and have access tho this *requestHandler* function to make HTTP requests and return results.

```
class AlertService(
    override val httpClient: OkHttpClient,
    override val gson: Gson
) : IAlertService, MarketTrackerService() {
    override suspend fun getAlerts(): List<PriceAlert> {
        return requestHandler<CollectionOutputModel<PriceAlert>>(
            path = alertsPath,
            method = HttpMethod.GET
        ).items
    }
```

Listing 5.6: AlertService.kt

5.8 Client Repositories

The repository layer in the client serves to save information locally, either being in the DataStore or in memory.

5.8.1 Memory

The AuthRepository utilizes memory to save the user's logged-in state. When a user is authenticated, we also store their lists and price alerts in memory. This approach allows us to efficiently access and display this data across various screens within the app without the need to repeatedly fetch it from the server. This reduces the load on our backend services by minimizing redundant requests, which can be particularly. This strategy not only enhances the user's interaction with the app but also optimizes resource utilization on both the client and server sides.

5.8.2 DataStore

The AuthRepository uses DataStore to store the token locally in the file system. When the user exits the app, MainActivity gets destroyed. Upon reopening the app, an initial request to the profile is made. If the token is still valid, it prevents the need for the user to log in again.

Chapter 6

Data Scripts

The following chapter explains the implementation and thought behind the process of inserting data in our database.

Our API can be used to insert information manually by human Operators through the React website portal or directly by making HTTP requests to the API. Alternately, Operators can be scripts that automate this process, keeping information up-to-date daily.

After admitting a human Operator into our system, part of our staff will contact them to determine how the information will be uploaded to the website. If the process is automated, our team will develop one of the following types of scripts to work with that operator's shops in his behalf:

- WebScrapping scripts - These scripts will extract product information, prices, and promotions from the operator's website and insert it into our API.
- CSV [23] files scripts - The operator provides an endpoint where a CSV file (column-based file) is available with all their products, prices, and promotions. The script will retrieve this file and insert the data into our API via HTTP.
- Operator's API - The script will make the necessary HTTP requests to the Operator's API and then transfer the data to our API.

These scripts can also make use of batch requests that our server supports, reducing the number of individual API calls and improving efficiency and performance.

6.1 Web Scraping

Web scraping is the process of extracting information from websites. Using **Puppeteer** [24], a library that provides a browser that can be programmatically controlled, with **Typescript** [25], which provides strong typing and modern JavaScript features, we were able to build

multiple supermarket scrapers. Puppeteer allows precise control over web browser interactions, including scraping HTML elements or listening for API requests. After getting all information of a Product and mapping specifications like categories and units, we make an HTTP request to our API to create or update an entry.

6.1.1 Getting the EAN from Supermarket Websites

Luckily, some websites have the EAN on the product page, even if its hidden from the user, so we were able to extract it when we scrape the products with our web scraping scripts.

Auchan has it displayed to the user:

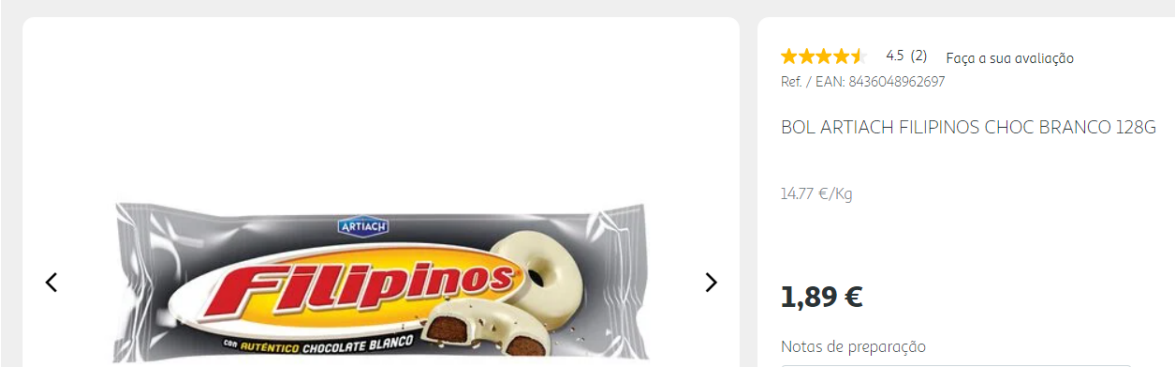

Figure 6.1: Auchan's website at June 2nd

Mercadão (Pingo Doce) makes an API call that has a list of them:

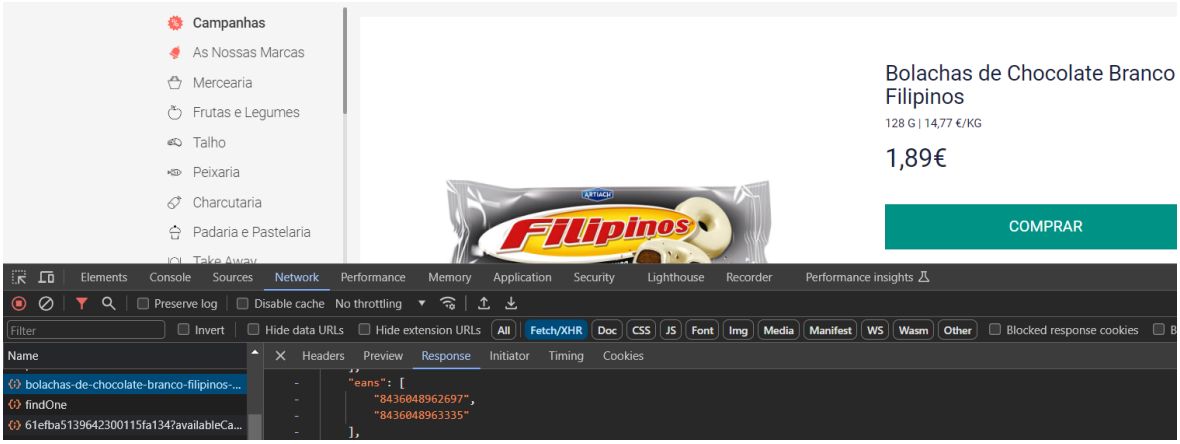

Figure 6.2: Mercadão's website at June 2nd

Continente has it embedded into a data url inside an element of the page:

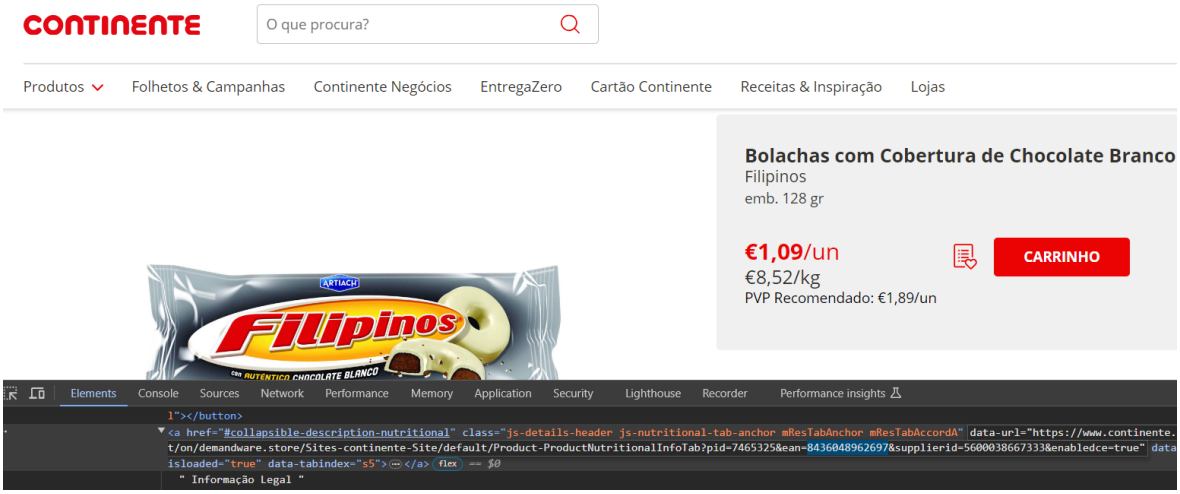

Figure 6.3: Continente's website at June 2nd

6.1.2 Matching Categories across supermarkets

Even tho categories are similar on different supermarket websites they aren't all the same. Each one always has a name and an id. Each category can then also have subcategories forming a tree structure. We decided to simplify the process and have just one layer of categories, in other words, a list of categories. We then map each website's top layer categories to ours, so every time a product is scraped we analyze its category in the external website and map to ours.

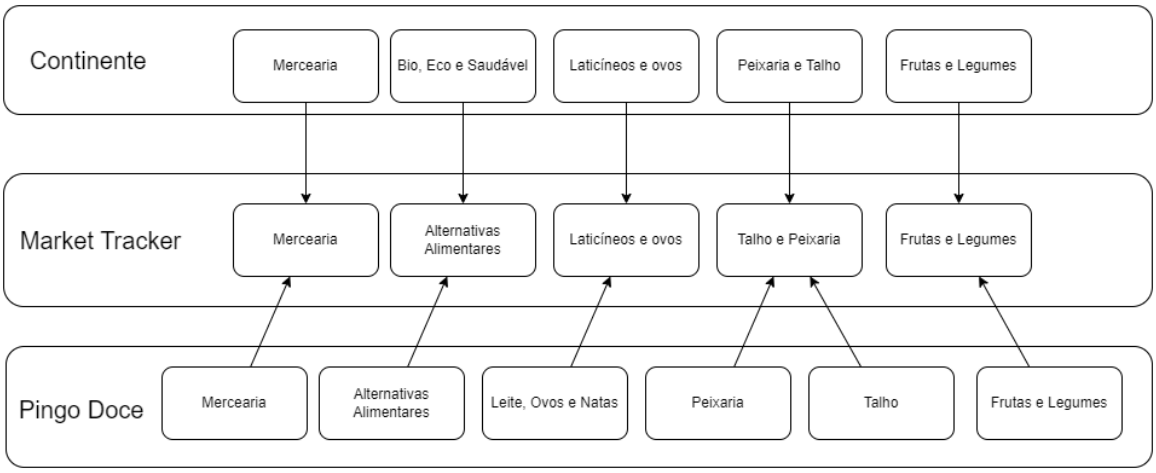

Figure 6.4: Categories mapping strategy

6.1.3 Retrieving data

There are 2 ways we get information from websites, we either scrap directly from their API (most efficient method) or from the HTML in their website. We discovered the website's endpoints from analysing the requests the browser made when we entered the site, we then simulate the same in our scripts and redirect the valuable information to our API.

In HTML scraping, we obtained several sitemap.xml URIs that index pages for search engines. This enabled us to gather numerous product page URLs, which we could then access via Puppeteer to search for the following elements:

- name
- image
- brand
- category it belongs
- quantity and unit

- price
- promotion (if available)

If our service was to be available 24/7, we would run these scripts at around midnight and send the information to our system under the corresponding store's name. If the product already existed in the creation POST request, the API is prepared to only update its price in that store and add it to the history.

```
class ContinenteScraper extends WebScraper {
  constructor(browser: Browser) {
  }

  parseUnitString(input: string): [ProductUnit, number] {
  }

  async scrapeProductPage(page: Page, url: string): Promise<Product> {
    if (!url.includes("continente.pt/produto")) {
      throw new Error(`Invalid url for ${this.constructor.name}:
          ${url}`);
    }

    await page.goto(url);
    await page.exposeFunction("mapUnit",
        this.parseUnitString.bind(this));
    await page.exposeFunction("mapCategory",
        this.mapCategory.bind(this));
    await page.exposeFunction("revertPercentage", revertPercentage);

    // HTML Selectors
    const ID_SELECTOR = "a.js-nutritional-tab-anchor";
    const NAME_SELECTOR = ".product-name";
    const UNIT_SELECTOR = ".ct-pdp--unit.col-pdp--unit";
    const BRAND_SELECTOR = "a.ct-pdp--brand.col-pdp--brand";
    const CATEGORY_SELECTOR =
      '.breadcrumbs .breadcrumbs-item:nth-child(3) a[itemprop="item"]';
    const IMAGE_URL_SELECTOR = ".pdp-img-container .ct-product-image";
    const PRICE_SELECTOR = ".js-product-price .value";
    const PROMOTION_PERCENTAGE_SELECTOR =
      ".product-name-details--wrapper
          .col-product-tile-badge-value--pvpr";

    await Promise.all([
      page.waitForSelector(ID_SELECTOR, { timeout: config.PAGE_TIMEOUT
          }),
      page.waitForSelector(NAME_SELECTOR, { timeout: config.PAGE_TIMEOUT
          }),
      page.waitForSelector(UNIT_SELECTOR, { timeout: config.PAGE_TIMEOUT
          }),
      page.waitForSelector(BRAND_SELECTOR, { timeout:
          config.PAGE_TIMEOUT }),
      page.waitForSelector(CATEGORY_SELECTOR, { timeout:
          config.PAGE_TIMEOUT }),
      page.waitForSelector(IMAGE_URL_SELECTOR, {
        timeout: config.PAGE_TIMEOUT,
      }),
      page.waitForSelector(PRICE_SELECTOR, { timeout:
          config.PAGE_TIMEOUT }),
    ]);
```

Listing 6.1: ContinentScrapper.ts

Chapter 7

Conclusion

Doing this project we learned how to work with .NET framework using C# programming language. We also wanted to know the limits and challenges of building a system that uses a relational database to do complex queries and learned the importance of optimizing the database schema.

We also knew and confirmed that the process is unstable and challenging to maintain. This instability comes from the frequent changes in website structures. Even a minor alteration in the HTML code can make our scraper scripts ineffective. Additionally, some supermarket websites do not provide the products' EAN numbers, making it impossible to include those products in our system. These factors collectively highlight the complexity and limitations of web scraping for this application.

7.1 Future Work

So far we have been able to implement and refactor the entire backend service and a first version of the Android application as well as integrating it with the API via HTTP. Although we are proud of our work so far, there is still room for significant additions, improvements and optimizations such as:

- Add the option to search for the best offer according to multiple filters for each product in a shopping list in our Android app as we already support that in our backend system.
- Optimizing the database schema and query to speed up the /api/products request, as it is taking a longer time to process as the number of products increases in our database.
- Using Server-Sent Events (SSE) in the list screen to allow the user to see products added by the other members in real time.
- Allowing Operators to create tokens with duration set by them to allow, for example, scripts to run using them.

- Authentication with refresh tokens to not force the user to login everyday.

Bibliography

[1] Kuantokusta - o comparador de preços nº1 em portugal. https://www.kuantokusta.pt/, last accessed on 09/07/24.

[2] Web api. https://www.w3schools.com/js/js_api_intro.asp, last accessed on 09/07/24.

[3] Http. https://developer.mozilla.org/en-US/docs/Web/HTTP, last accessed on 09/07/24.

[4] Android. https://www.android.com/intl/pt$_p$t/, *lastaccessedon*09/07/24.

[5] C programming language. https://learn.microsoft.com/en-us/dotnet/csharp/, last accessed on 09/07/24.

[6] .net. https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework last accessed on 09/07/24.

[7] Entity framework. https://github.com/dotnet/efcore, last accessed on 09/07/24.

[8] Asp.net mvc. https://learn.microsoft.com/en-us/aspnet/core/mvc/overview?view=as last accessed on 09/07/24.

[9] Postgresql. https://www.postgresql.org/, last accessed on 09/07/24.

[10] Kotlin. https://kotlinlang.org/, last accessed on 09/07/24.

[11] Google. https://www.google.com, last accessed on 09/07/24.

[12] Firebase cloud messaging. https://firebase.google.com/docs/cloud-messaging?hl=pt, last accessed on 09/07/24.

[13] React. https://react.dev/, last accessed on 09/07/24.

[14] Moq library. https://github.com/devlooped/moq, last accessed on 09/07/24.

[15] Rfc problem. https://datatracker.ietf.org/doc/html/rfc7807, last accessed on 09/07/24.

[16] Google oauth 2.0. https://developers.google.com/identity/protocols/oauth2, last accessed on 09/07/24.

[17] European article number. https://www.vendus.pt/blog/ean-codigo-barras/, last accessed on 09/07/24.

[18] Http batch requests. https://learn.microsoft.com/en-us/graph/json-batching, last accessed on 09/07/24.

[19] Viewmodel. https://developer.android.com/topic/libraries/architecture/viewmode last accessed on 09/07/24.

[20] Dagger hilt. https://developer.android.com/training/dependency-injection/hilt-ar last accessed on 09/07/24.

[21] Okhttp. https://square.github.io/okhttp/, last accessed on 10/07/24.

[22] Gson. https://github.com/google/gson, last accessed on 10/07/24.

[23] Csv file. https://flatfile.com/blog/what-is-a-csv-file-guide-to-uses-and-benefit last accessed on 09/07/24.

[24] Puppeteer. https://pptr.dev/, last accessed on 09/07/24.

[25] Typescript programming language. https://www.typescriptlang.org/, last accessed on 09/07/24.