Mobile Devices Programming
- **MoneyMind** -

xxxxxx xxxxxx
student ID xxxxxx
Coldani Andrea
student ID xxxxxx
xxxxxx xxxxxx
student ID xxxxxx

A.Y. 2024/2025

September 26, 2025

Contents

1 Introduction

The project aims to develop a mobile application for Android, designed to provide comprehensive support in managing **personal finances**.
The app offers a set of features designed to meet all the needs of a mobile application, including the offline-first principle, adaptability to multiple device "forms," and flexibility in language switching.
Among the main features offered, the user will be able to:

- Add **income** and **expenses** quickly and easily, with the option to categorize each transaction.
- Set a **budget** for each category, to always keep expenses under control and respect preset spending limits.
- Define **savings goals** for each category, helping the user plan and reach their financial milestones.
- Change **currency** easily, allowing the user to manage their finances in different currencies, making the app versatile for those who travel or manage foreign currency income.

In addition, the app allows the creation of a personal account, so that each user can save and sync their data, access personalized services, and track their financial habits securely and safely.

2 Development Tools

2.1 Technical Tools

- **Android Studio:** used as IDE
- **GitHub:** used for collaborative project management. Our repository is structured as follows:
  - **Main branch:** the main branch where stable and working versions of the application are pushed.
  - **Develop branch:** the branch dedicated to integrating changes from various developers. It is the reference point for new features under development. This branch allows the whole group to stay aligned without modifying individual branches.
  - **Andrea/xxxxxx/xxxxxx-develop branch:** a separate branch for each developer. Each time a new feature was implemented, it was merged into the `andrea/xxxxxx/xxxxxx-develop` branch.
- **Firebase/Firestore:** Google services used in this application to manage login/registration and as a remote database. Particularly useful for synchronization across devices.
- **Various libraries:** for example `Retrofit` to interact with the API, `Room` to implement the local database.
- **ECB API:** service that provides exchange rates between currencies. The exchange rate with the euro is always returned, as it is the default currency. More details in paragraph 3.

2.2 UI Design Tools

The tool used to design the graphical part of the application is `Figma`. This tool makes it easy to add and modify graphic elements that are easily replicable according to `MaterialDesign3 guidelines`.

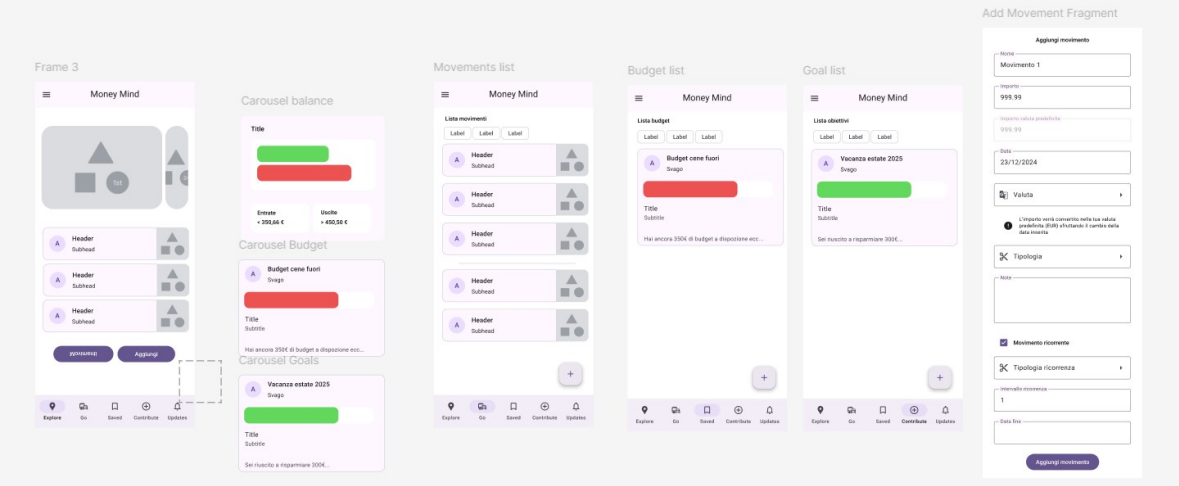


Figure 1: UI draft during design phase

3 API

The API used in this application is intended to retrieve **exchange rates** of currencies against the **euro**, which is the default currency of the application.

The **endpoint** used to obtain the data is the following:
`"https://data-api.ecb.europa.eu/service/data/EXR/D..EUR.SP00.A"`

To specify the days for which we need the exchange rates, the **parameters** `startPeriod`, `endPeriod` and `format` are used:
`"?startPeriod=YYYY-MM-dd&endPeriod=YYYY-MM-dd&format=jsonData"`.

To **interact** efficiently and easily with the API, the `Retrofit` library was used for managing `HTTP` communications.
`Retrofit` greatly simplifies the process of sending requests and handling responses, making the code much more **readable** and **maintainable** compared to direct use of low-level classes.

After the query, the API will return a **JSON file** which is interpreted by a dedicated class.

3.1 Practical Uses of the API

- The user can **record income** or **expenses** in currencies other than the euro, even with a date **prior** to today.
- The API provides updated exchange rates for as many as **32 different currencies**. The user will be able to easily select the desired currency via a drop-down menu.

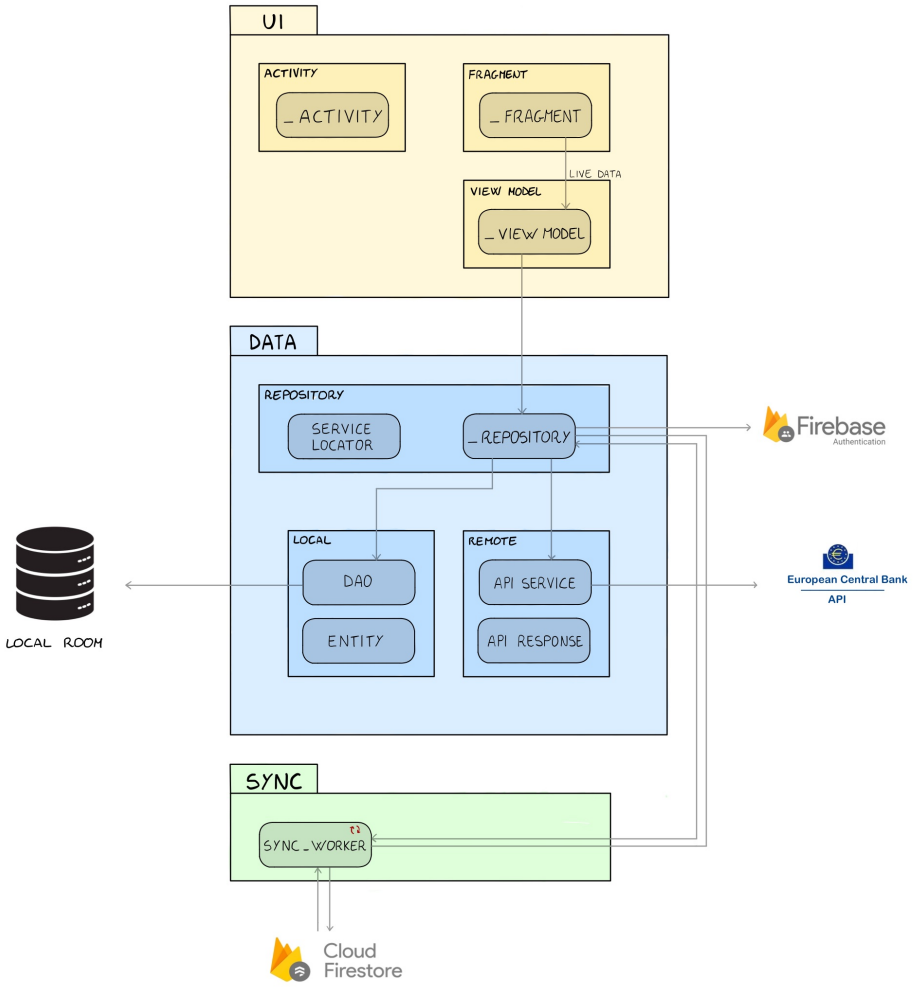
3.2 Availability and Limitations

- Exchange rate data is available from **1999** to **today**.
- The **update** of rates takes place daily at **4:00 PM**.
- Since data for **Saturday**, **Sunday** and **holidays** is not available, the application uses the exchange rates of the last available **business day**.

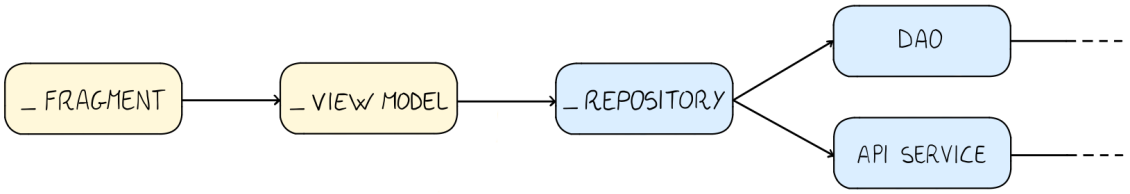
This management ensures that the user can obtain accurate and updated data to record transactions in different currencies over time.

4 Architecture

The **architectural pattern** followed by the application is MVVM (Model-View-ViewModel), which aims to separate **presentation** logic (UI) from **business** logic. The **architecture** of the application is as follows:



The general **flow** by which **data** is obtained is:



4.1 UI Layer

This layer handles data coming from the **Data Layer**.
During the development of the application, a clear separation between the **UI** and the **Data Layer** was maintained, ensuring that the UI **does not** handle **data management**, but only **displays** it. In particular:

4.1.1 Activity

There are 2 to ensure the "Single activity - multiple fragments": `MainActivity` which manages login/registration and `MainNavigationActivity` which manages the rest of the UI.

4.1.2 Fragment and ViewModel

They communicate mainly through **LiveData**, which, using the **Observer** pattern, allows them to exchange data reactively and without **direct coupling**.
Each Fragment is associated with a ViewModel that manages the related data, ensuring a clear separation between data management logic and the UI.
The Fragments and ViewModels used in the application are:

- `AddBudgetFragment` and `AddBudgetViewModel`
- `BudgetFragment` and `BudgetViewModel`
- `AddGoalFragment` and `AddGoalViewModel`
- `GoalFragment` and `GoalViewModel`
- `AddTransactionFragment` and `AddTransactionViewModel`
- `TransactionFragment` and `TransactionViewModel`
- `HomeFragment` and `HomeViewModel`
- `SettingFragment` and `SettingViewModel`
- `MainActivityViewModel`: used at **startup** of the app to check whether the user is already **logged in** in order to **sync** their data and determine which **screen** to show next.

4.2 Data Layer

First of all, we distinguish between two types of classes: those responsible for retrieving data from the correct source, namely the **Repositories**, and those that directly interface with data resources.
In this second category there are:

- **Local:** contains the classes to interact with `Room` that model entities and define the **Queries**.
- **Remote:** contains the classes needed to interact with the **API**.

4.2.1 Repository

The **Repositories** are responsible for retrieving data from the correct **source** and are the following:

- `BudgetRepository`
- `CategoryRepository`
- `GoalRepository`
- `ExchangeRepository`
- `TransactionRepository`
- `RecurringTransactionRepository`
- `UserRepository`
- `GenericRepository`

To promote the **Offline-First** approach, Repositories fetch data from the **local Database** (Room). These data are later synced to the **remote Database** (see paragraph 4.3).
The only **exception** is for **Login/Registration**: in this case the repository communicates directly with **Firebase Authentication** both **online** and **offline**, provided that the user has logged in at least once.

4.3 Sync

This component is essential to ensure **persistence** of the user's account data, regardless of the device used. In particular, the **Sync_Worker** handles synchronization of data between the **local database** (Room) and **Firebase**, transferring data in both directions.
By default, this process runs every **30 minutes**, at each **startup** of the app or at each **login**, although Android may adjust the timing based on system resource management.
Each Repository stores in the **Shared Preferences** a `TIMESTAMP` value, which indicates the last synchronization performed.
Specifically, the process works as follows:

1. **Synchronization** of local resources **to Firebase**: a query is made where `SYNC = false` and then the results are added to Firebase.
2. **Download** of data **from Firebase** not yet synced: this occurs by downloading the data and determining the maximum date among the transactions already synced on **Firebase**. Then we compare this date with the current `timestamp`. We use the more recent date between the two and retrieve from **Firestore** all records with a `last_sync` value greater than the most recent date, thus ensuring that only updated data are synced.
3. **Updating** the **timestamp**: once the synchronization is complete, the `timestamp` value in `shared preferences` is updated.

4.4 Patterns Used

At the structural level, to ensure clean and efficient code and pursue **Low Coupling** between classes, the following **Patterns** were mainly adopted:

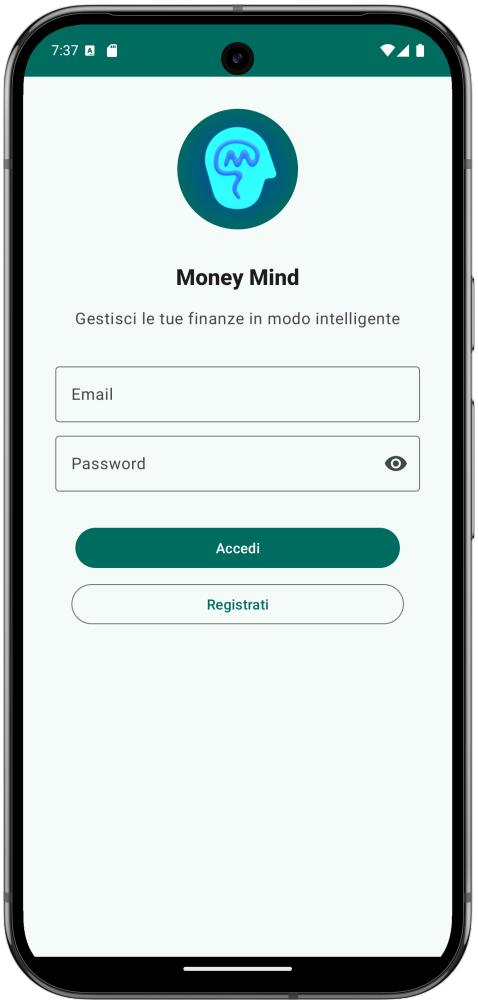
- **Observer**: to allow fragments to observe `LiveData`
- **Service Locator**: delegates the **responsibility** of Repository instantiation to a central component, preventing individual classes from having to handle it
- **Adapter**: allows **adapting incompatible interfaces** so they can work together. It was used, for example, in the home `carousel` and in the `drop-down menus`.
- **Singleton**: ensures that a class has **only one instance** and provides a global access point to that instance. For example, it was used to instantiate the `Retrofit Client`

- **Factory**: used to instantiate ViewModels that required a specific **Adapter**. In this way, the Factory centralizes ViewModel instantiation, **managing** the adapter dependency and improving code modularity and maintainability
- **Helper**: classes that contain only **static** methods and provide **support functions** for a specific domain without the need to create instances. These classes were placed in the `"Utils"` package
- **Static Utility Class**: a class with **static** methods that provides **general support** functions for the application, making tools **reusable** in different parts of the code. It includes common operations such as **date conversion** or creation of **SnackBars**, avoiding code duplication

5 Features

5.1 Login

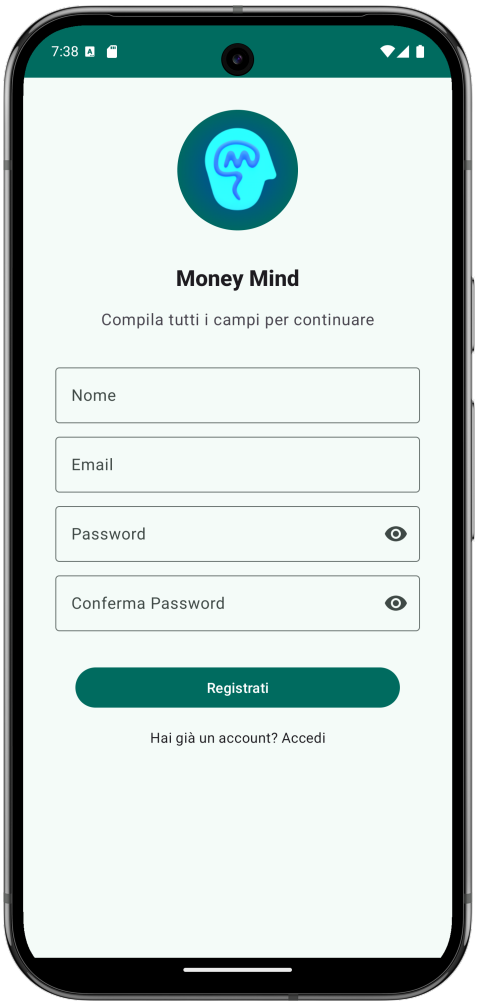
The user can access their account by entering `email` and `password`. If not yet registered, they have the option to switch to the **registration screen** to create a new account. Below, the **login** screen:



5.2 Registration

The app manages user registration by requiring a `name`, an `email`, and a `password`.
The email must be unique, and this is guaranteed by **Firebase**, which prevents the creation of a new account if the address is already associated with another user.
In addition, the **password** must have a minimum length of 8 characters and contain at least one

uppercase letter, one lowercase letter, one number, and one special character to ensure a basic level of security. Below, the **registration** screen:



5.3 Home

The Home provides an overview of the **expenses** and **income** of the current month, accompanied by a **graphical** representation. In addition, an interactive **carousel** allows you to scroll and also view the balances of previous months (dynamically loaded in groups of three up to the first month of activity). Further down, the screen shows the **last three transactions** made, with the possibility of adding a new one through a dedicated button.
A **navigation bar** allows navigation between the various screens. Below, the **home** screen:

5.4 Transactions

This screen allows you to view all the **transactions** made by the user, divided into two sections: "`Transactions`" and "`Recurring Transactions`". Recurring transactions can be entered by the user for expenses with a **fixed frequency** (daily, weekly, monthly, or yearly) and are checked and possibly recorded by a **worker** every 4 hours, at app startup, or when a new one is registered. Currency **conversion** is also performed if necessary.
In addition, from this screen it is possible to **delete** and **edit** transactions from the list, and to easily access the **add transaction** screen. From the `add transaction` screen, it is possible to **convert** transactions and view their value in the default currency (Euro). Below, the **Transactions** and **Add transaction** screens:

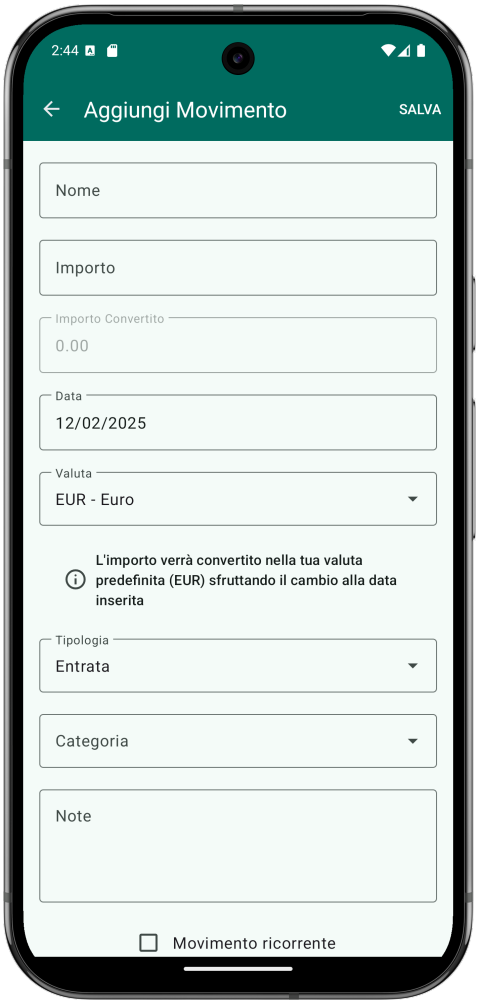


Figure 2: Adding a normal transaction

Figure 3: Adding a recurring transaction

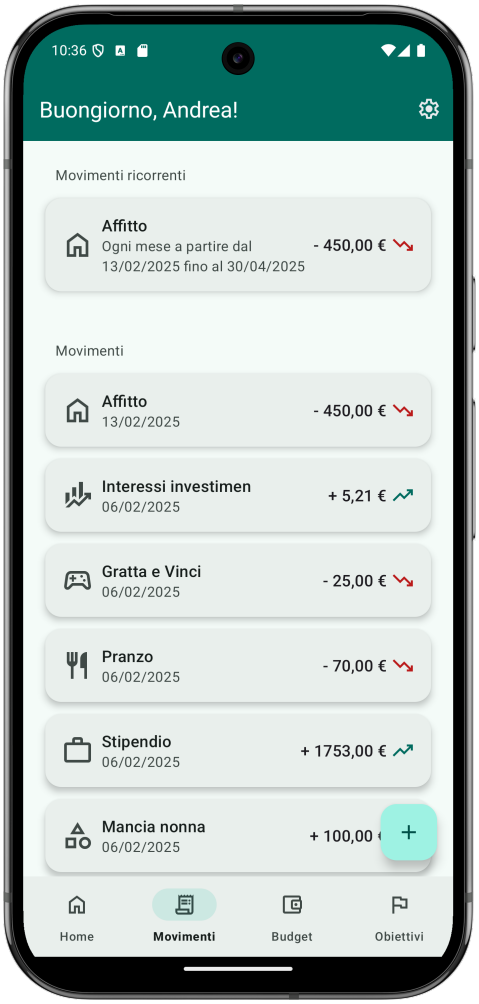


Figure 4: Transaction list

Figure 5: Deleting transactions from the list

5.5 Budget

Similar to the `transactions` screen, the **budgets** added by the user are displayed, with the possibility to **delete** and **edit** budgets from the list and, through another associated screen, to **add** new ones. The progress of each budget is accompanied by a **graphical representation** to make it easier to view. Below, the **Budget** and **Add budget** screens:

Figure 6: Adding a budget

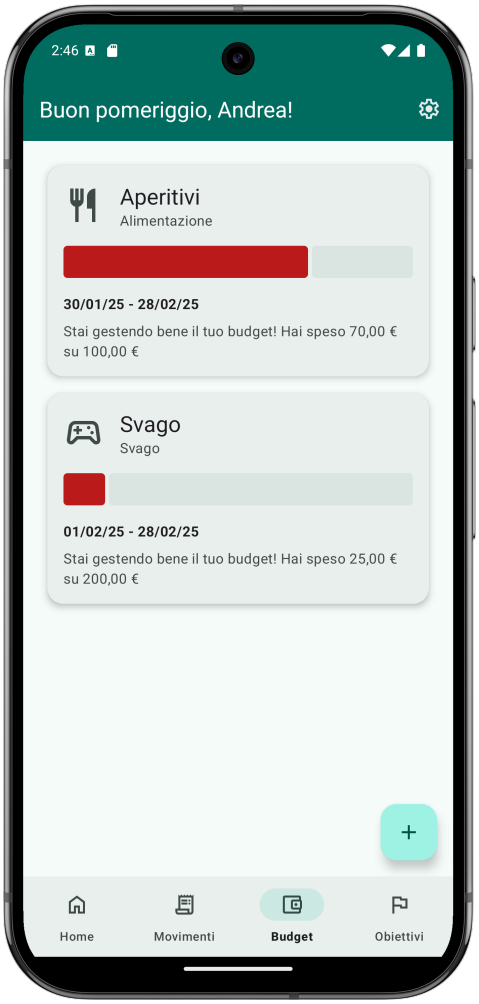


Figure 7: Budget list

5.6 Goals

As in the two previous sections, the **goals** added by the user also have two dedicated screens. Unlike budgets, in goals the user must **manually enter** the saved amounts by modifying the value in the edit screen (accessible by clicking on the goal of interest). Below, the **Goals** and **Add goal** screens:

Figure 8: Adding a goal

Figure 9: Goals list

5.7 Settings

In the **settings** section accessible from the home screen, it is possible to set the **theme** (Light/Dark) and the available **languages** (Italian/English). It is also possible to perform **Logout**. Below, the **Settings** screen:

6 Future Developments

During the development of the application, we had to make strategic choices to meet project deadlines while ensuring a **stable** and **working** version.
However, some additional features that would have further **improved** the user experience were not implemented due to their **technical complexity** and therefore the limited time to develop them.
Here are the main features we would have liked to add:

- **Goals:** allow setting aside part of an income to achieve a goal.
- **Notification System:** personalized alerts for exceeded spending thresholds or reminders for scheduled savings.
- **Login with Google Account:** allow an unregistered user to log in with their Google credentials.
- **Password Recovery:** a password reset system via email, with the sending of a reset link.

- **Ability to Change the Default Currency:** currently, the base currency is fixed, but it would have been useful to allow users to select a default currency.
- **Currency Conversion Page:** a dedicated interface to quickly convert amounts between different currencies using the European Central Bank API.
- **Charts and Detailed Analysis:** implementation of visual reports to monitor spending trends over time.
- **Application Languages:** translation of the app into languages other than Italian and English.