Oleksandr Yuriichuk,

Repository with App: https://github.com/Bombonchik/MoneyManager

Documentation

1. Introduction

This document provides an overview of the Personal Finance Manager Application for Android. The primary objective of this application is to offer users a comprehensive platform to manage their personal finances effectively and conveniently. Developed using C# for Android devices, this application aims to address the prevalent challenges associated with personal finance management in today's fast-paced world.

1.1 Problem Statement

The complexities of managing personal finances have grown exponentially in the modern era. A significant portion of the population grapples with tracking expenses, adhering to budgets, and saving for future financial objectives. Such challenges often culminate in financial stress, hindering individuals from attaining financial stability. The Personal Finance Manager Application seeks to mitigate these issues, presenting users with a streamlined and efficient method to oversee their finances.

1.2 Features Overview

The Personal Finance Manager Application boasts an array of features tailored to enhance the user experience, including:

• **Account Management**: Users have the flexibility to create, edit, and delete accounts with a wide variety of types. When setting up an account, users can choose a name, select the account type, pick an icon for easy identification, and if desired, assign a unique identifier to the account.

• **Transaction Management**:

- Users can effortlessly add transactions of three distinct types: income, expense, and transfers between accounts.
- For income transactions, users can select an appropriate income category, and for expense transactions, an expense category can be chosen.
- Additionally, users have the option to add notes to transactions, providing context and clarity to each entry.

• **Transaction Tracking**: Users can view all their transactions, organized by each day, ensuring comprehensive financial tracking and a clear understanding of their daily financial activities.

• **Total Balances Overview**: At a glance, users can see the total balances across all their accounts, combined with their total income and total expenses, offering a holistic perspective of their financial standing.

• **User-Friendly Interface**: The application is designed with an intuitive user experience in mind, ensuring users can navigate and manage their finances with ease. The application also supports both light and dark themes, adapting to the user's phone theme preference.

1.3 Project Goals

The primary objectives of the Personal Finance Manager Application project encompass:

• Developing a dedicated application for Android devices using .NET MAUI in C#.

• Incorporating SQLite for local data storage and leveraging LINQ for data querying.

• Implementing the Model-View-ViewModel (MVVM) architectural pattern, a staple in .NET MAUI applications, to segregate the app's data (Model), UI (View), and logic (ViewModel), ensuring modularity and ease of maintenance.

2. User's Documentation

See the screens section for a visual representation

2.1 Main page

Navigation:

- The main page of the Application is designed with tabs at the bottom of the screen.

- These tabs allow you to navigate between different sections of the application.

- By default, you will see the Accounts page.

Switching Tabs:

- To switch between tabs, you can tap on the desired tab at the bottom of the screen.

- Each tab represents a specific section of the application, such as Accounts page, Transactions page, Add New Transaction, Overview and Settings.

2.2 Accounts Overview Page

Overview:
This space is dedicated to providing a comprehensive look at your financial situation. Here, you can view your accounts, check balances, and perform various account management tasks, all in one place.

Features and Functions:

1. **Financial Summary Section**:
    - **Total Balance**: Shows the combined balance of all your accounts.
    - **Income**: Displays the total income for the current month.
    - **Expenses**: Provides the total expenses you've incurred for the month.
    - **Average Expense**: Shows the average expenses you've had for the current month.
2. **Accounts Display**:
    - This section provides a visual grid representation of all your accounts.
    - Each account card showcases:
        - **Icon**: A representative symbol for the account type.
        - **Account Type**: Shows the nature of the account (e.g., Savings, Checking, Credit).
        - **Identifier**: A unique identifier for your account, such as the last four digits of an account number.
        - **Account Name**: A user-defined name for easy identification.
        - **Balance**: The current balance of the account.
    - Selecting an account card will highlight it, indicating it's ready for further operations.
3. **Account Management Controls**:
    - **New Account**: Clicking this button open Account Management Page where you can add a new financial account to your overview.
    - **Edit**: Once an account card is selected, you can use this button to open Account Management Page and modify account details.
    - **Delete**: Use this button to remove the selected account from your overview

2.3 Account Management Page

Overview:
This is your centralized hub for managing your financial accounts in Finance Manager application. Here, you can add new accounts, update the information of existing accounts, select account icons, and specify various account details.

Features and Functions:

1. **Account Details Section**:
    - **Account Name**: Enter the name of your account (e.g., "Savings", "Checking", etc.). Maximum length is 16 characters.
    - **Balance**: Specify the current balance of this account.
    - **Identifier**: This can be a unique identifier for your account. For example, it could be a credit card number. It's used to differentiate multiple accounts of the same type. Maximum length is 256 characters.
    - **Account Type**: Choose from the predefined account types available or enter your own type if your account type isn't listed. Maximum length for a custom type is 16 characters.
2. **Icon Selection Section**:
    - **Select the icon**: Browse through the available icons and pick one that best represents the nature of your account. This helps to quickly identify accounts in your dashboard or list.
3. **Save or Cancel**:
    - **Save**: After filling out all details and selecting an icon, click the save button to store this account information.
    - **Cancel**: If you change your mind or need to exit without saving, click the cancel button.

Tips:

- Ensure all required fields are filled before saving an account.
- For security reasons, double-check any sensitive data you enter, such as the account's identifier.

2.4 Transactions Overview Page

Overview:
This page serves as a centralized space for reviewing and managing your financial transactions. Here, you can see a comprehensive list of your daily transactions, including their categories, notes, accounts, and amounts.

Features and Functions:

1. **Transaction Groups by Date**:
    - The transactions are grouped by date to help you navigate through your financial history.
    - Each group includes the date at the top, providing a clear time reference.
2. **Individual Transactions Display**:
    - Under each date group, you'll find a collection of individual transactions made on that day.
    - Each transaction entry includes:
        - **Category**: Displays the category of the transaction (e.g., "Groceries", "Transportation").
        - **Note**: Shows any notes you've added to provide context for the transaction.
        - **Account**: Indicates the account associated with the transaction.
        - **Amount**: Displays the transaction amount, with positive amounts for income and negative amounts for expenses.
3. **Color Coding**:
    - Transactions are color-coded based on their type (income, expense or transfer), making it easier to differentiate between them.

Tips:

- Regularly review your transactions to ensure accuracy and maintain financial awareness.
- Keep your notes concise but informative to remember the context of each transaction.

2.5 Transaction Management Page

Overview:
This page empowers you to create and edit financial transactions with ease. It offers a structured environment to input transaction details, including the type of transaction, date, accounts involved, amounts, and additional notes.

Features and Functions:

1. **Transaction Type Selection**:
    - At the top of the page, you can select the type of transaction you wish to record. The available options include income, expense, and transfer.
2. **Transaction Details Input**:
    - The page provides fields to input essential transaction details based on the selected type.
    - For all transaction types:
        - **Date and Time**: Specify the date and time of the transaction.
        - **Amount**: Enter the transaction amount in the relevant currency.
        - **Note**: Add any relevant notes to describe the transaction.
    - For income and expense transactions:
        - **Source Account**: Select or enter the source account for the transaction.
        - **Category**: Assign a category.
    - For transfer transactions:
        - **Source Account**: Choose the source account from which you're transferring funds.
        - **Destination Account**: Indicate the account receiving the transferred funds.
3. **Account and Category Selection**:
    - Depending on the transaction type, you can select accounts and categories from user-friendly drop-down lists.
    - The selected accounts and categories are displayed for your reference.
4. **Save and Cancel Buttons**:
    - Use the "Save" button to finalize your transaction and save it to your financial records.
    - If you decide not to proceed, click the "Cancel" button to discard the transaction.

2.6 Overview

Overview:
Please note that this page is currently under development and not yet complete.

2.7 Settings

Overview:
Please note that this page is currently under development and not yet complete.

3. Programmer’s Documentation

3.1 Architecture

The Finance Manager application boasts an architecture that guarantees robustness and scalability. This design is crafted with the intention of offering a top-tier user experience while maintaining a codebase that is easy to manage, expand, and optimize. Several fundamental elements form the core of the Finance Manager architecture:

Embracing .NET MAUI for Development

The Finance Manager app is built on the foundation of .NET MAUI (Multi-platform App UI), a powerful development framework that drives its feature-rich capabilities. This strategic choice is guided by the framework's distinct advantages, paving the way for a robust financial management solution.

What is .NET MAUI?

.NET MAUI empowers developers to craft cross-platform applications from a unified codebase. It offers native user interfaces adaptable to Android, iOS, macOS, and Windows, replacing Xamarin.Forms while enhancing the development experience.

Why .NET MAUI?

The selection of .NET MAUI is based on several pivotal factors:

Unified UI: .NET MAUI creates native interfaces tailored to each platform, ensuring smooth interactions and adherence to platform aesthetics.

One Codebase: Developing with a single codebase across various platforms accelerates development and simplifies maintenance.

Component Reuse: The framework's comprehensive cross-platform controls and components enhance code reusability, fostering consistent functionality and adaptable UI.

Enhanced Performance: Native compilation in .NET MAUI optimizes performance, resulting in rapid loading and seamless user interactions.

Contemporary Design: Integration of modern UI patterns guarantees visually appealing and intuitive user experiences.

Robust Ecosystem: Aligned with the broader .NET ecosystem, .NET MAUI enjoys access to a wealth of tools, libraries, and resources.

Future-Proof: Designed to evolve with platform advancements, .NET MAUI ensures compatibility with future OS updates and devices.

Developer Efficiency: Leveraging familiar technologies like C# and XAML streamlines development, promoting rapid feature creation and iteration.

In summary, .NET MAUI's ability to deliver native experiences, optimize code reuse, ensure performance, embrace modern UI standards, and align with the .NET ecosystem drives its selection for the Finance Manager app. This choice guarantees a seamless, feature-rich, and forward-looking financial management solution catering to diverse devices and operating systems.

3.1.1 Model-View-ViewModel (MVVM) Design Pattern:

Finance Manager embraces the MVVM design pattern, a widely-adopted architectural approach that shines in the realm of cross-platform mobile app development. MVVM divides the application into three primary segments:

Model: This component encapsulates the data and business logic of the application. It encompasses data structures, data access mechanisms, and other pivotal business-oriented elements.

View: Responsible for the user interface (UI) components and layout, the view interfaces with the ViewModel to exhibit data and capture user interactions effectively.

ViewModel: Functioning as the bridge between the Model and the View, the ViewModel orchestrates the application's state. It skillfully processes user inputs and subsequently updates either the Model or the View as required.

Advantages of the MVVM Pattern:

The MVVM pattern orchestrates a clear separation of concerns, facilitating simultaneous development of both the UI and application logic. This partitioning ensures that parallel development remains feasible while guaranteeing that the app retains its high level of maintainability.

3.1.2 Messaging Pattern:

In the Finance Manager app, the Messaging Pattern serves as a sophisticated communication mechanism that enables seamless interaction and data sharing between different ViewModels. This approach empowers the application to offer a cohesive user experience by enabling efficient coordination between various components.

Understanding the Messaging Pattern

The Messaging Pattern is a strategic design approach that facilitates communication and data exchange between different parts of an application without tightly coupling them. Messages act as lightweight carriers of information, conveying commands, queries, and data payloads.

Benefits of the Messaging Pattern

1. **Decoupled Components**: By utilizing the Messaging Pattern, different parts of the application remain decoupled. Each ViewModel can focus on its responsibilities without being overly dependent on others. This enhances the maintainability and scalability of the codebase.
2. **Efficient Data Sharing**: A key advantage is the efficient sharing of data between ViewModels. Instead of fetching the same data repeatedly from the database, data can be fetched once and shared using messages. This approach reduces redundant database queries and improves performance.
3. **Modularity and Extensibility**: The Messaging Pattern promotes modularity in the application's architecture. Adding new features or modifying existing ones becomes easier, as the decoupled nature of this pattern allows specific parts of the app to be extended or modified without affecting others.

The Finance Manager app employs **CommunityToolkit.Mvvm.Messaging** as the preferred choice for seamless communication between different components. This decision is grounded in its reliability, efficiency, and compatibility with modern development practices.

Key Reasons for Choosing CommunityToolkit.Mvvm.Messaging:

- **Efficiency**: The framework offers a lightweight messaging structure, ensuring swift message transmission with minimal overhead.
- **Compatibility**: As part of the Microsoft Community Toolkit, it aligns well with contemporary development approaches, enhancing compatibility and future support.
- **Simplicity**: Setting up communication between components becomes straightforward, thanks to the framework's user-friendly syntax.

Benchmarks:

Method	Mean	Error	StdDev
MVVMToolkitStrong	**4.025 ms**	0.0177 ms	0.0147 ms
MVVMToolkitWeak	**7.549 ms**	0.0815 ms	0.0762 ms
MvvmCrossStrong	11.483 ms	0.0226 ms	0.0177 ms
MvvmCrossWeak	13.941 ms	0.1865 ms	0.1744 ms
MVVMLight	52.929 ms	0.1295 ms	0.1011 ms
Stylet	91.540 ms	0.6362 ms	0.4967 ms
MvvmGen	141.743 ms	2.7249 ms	2.7983 ms
Catel	148.867 ms	2.6825 ms	2.5093 ms
Prism	150.077 ms	0.5359 ms	0.4184 ms
CaliburnMicro	280.740 ms	3.7625 ms	3.1418 ms
MauiMessagingCenter	673.656 ms	1.7619 ms	1.3755 ms

Each benchmark run involves sending 4 different messages 1000 times, to 100 recipients. As you can see, WeakReferenceMessenger and StrongReferenceMessenger are both the fastest

3.1.3 Repository Pattern

The Finance Manager app employs the **Repository Pattern** to facilitate efficient communication with the database. This architectural choice not only streamlines database interactions but also enhances the codebase's organization and maintainability.

Repository Pattern: An Overview

The Repository Pattern acts as an intermediary between the application's business logic and the database. It encapsulates data access operations, allowing for a cleaner separation of concerns. The pattern offers several benefits, including:

- **Centralization**: All database-related operations are confined within dedicated repository classes. This centralized approach enhances code readability and minimizes redundancy.
- **Abstraction**: The repositories abstract away the complexities of data access, allowing the application to interact with data without being concerned about database-specific details.
- **Code Reusability**: By encapsulating database operations, the repositories can be reused across different parts of the application, reducing duplicated code and promoting consistency.

Advantages of the Repository Pattern:

- **Efficiency**: Repositories optimize data access, helping to avoid redundant database queries and promoting efficient use of resources.
- **Maintainability**: The pattern's structured approach makes the codebase easier to maintain and extend. Changes to database operations can be made within the repositories without affecting the rest of the application.
- **Flexibility**: Repository classes can be extended or customized to accommodate evolving data access requirements, ensuring adaptability over time.

Implementation Example:

In the Finance Manager app, repositories are implemented following the **IBaseRepository<T>** interface, which defines essential methods for data access. Each repository class, such as **BaseRepository<T>**, encapsulates database interactions for a specific data entity.

The repositories manage CRUD (Create, Read, Update, Delete) operations, retrieval of items, and more. By employing these repositories, the app benefits from efficient, structured, and organized data access, promoting high performance and maintainability.

Moreover, the use of a singleton pattern ensures that a single instance of each repository is available throughout the application's lifecycle, optimizing resource usage.

Through the implementation of the Repository Pattern, the Finance Manager app attains robust and efficient database interactions, contributing to a seamless and responsive user experience.

3.1.4 Services Pattern

Implementing the Services Pattern for Logic Separation and Flexibility

The Finance Manager app embraces the **Services Pattern** as a strategic approach to segregate specific logic that doesn't neatly align with ViewModels. This architecture not only keeps the codebase clean and well-organized but also facilitates executing operations that might not directly relate to specific models or require immediate access to them.

Services Pattern: A Closer Look

The Services Pattern introduces an additional layer of abstraction between the ViewModels and the business logic. It serves as a container for methods and functions that perform specialized tasks, which might not fit seamlessly within ViewModel boundaries. This pattern delivers several advantages:

- **Separation of Concerns**: The Services Pattern ensures that different types of logic, such as data manipulation, calculations, or external integrations, are kept separate from the core ViewModel functions. This separation fosters modularization and code readability.
- **Code Reusability**: The logic encapsulated within services can be reused across multiple ViewModels or other parts of the application, promoting consistency and minimizing code duplication.
- **Flexibility**: Services are designed to be decoupled from specific models or ViewModels. This flexibility allows the execution of logic that doesn't depend on a particular data entity or ViewModel, making it versatile and adaptable.

Benefits of the Services Pattern:

- **Clean ViewModel**: By offloading non-ViewModel-specific logic to services, the ViewModels remain focused on managing user interactions and maintaining the application's state.
- **Improved Testability**: The separation of services from ViewModels simplifies unit testing, as individual services can be tested independently, ensuring robustness and reliability.
- **Future-Proofing**: Services can hold logic that is required for application-wide operations, even if specific models or ViewModels are not yet instantiated. This future-proofing helps prevent delays in executing crucial tasks.

Implementation Example:

In the Finance Manager app, services are employed to handle various functions that might not be confined to a single ViewModel or model. These services can include data manipulation, validation, calculations.

3.1.5 Converter Pattern

The Finance Manager app employs the **Converter Pattern** to effectively transform and format data from C# code to XAML elements. This pattern plays a pivotal role in ensuring that data is correctly displayed and styled according to user interface requirements. The use of converters contributes to a more user-friendly and visually appealing app.

Converter Pattern: A Deeper Dive

The Converter Pattern introduces a layer of logic between the raw data stored in C# code and its presentation in the XAML user interface. Converters are classes that implement the **IValueConverter** interface and offer two primary methods: **Convert** and **ConvertBack**.

- **Convert**: This method is responsible for converting the source data (usually from C#) into a format suitable for display in the XAML interface. For instance, converting a color code string into a **Color** instance, or formatting a date for user-friendly presentation.
- **ConvertBack**: This method is used to reverse the process, converting user input or changes in the UI back into a format suitable for C# data manipulation.

Advantages of the Converter Pattern:

- **Data Transformation**: The Converter Pattern bridges the gap between data representation and user interface requirements. It enables the transformation of data into a format that best fits the presentation context.
- **Separation of Concerns**: By separating the data conversion logic into converters, the main application logic and ViewModel code remain focused on core functionality. This separation of concerns enhances code readability and maintainability.
- **Reusability**: Converters can be reused across different parts of the app, promoting code consistency and reducing duplication of logic.
- **XAML Agnosticism**: Converters enable developers to provide uniform data formatting and display across different parts of the app's user interface, irrespective of the specific XAML element being used.

Application in the Finance Manager:

The Finance Manager app demonstrates the use of converters to enhance the user interface. The provided converters, such as **HexToColorConverter**, **IdentifierConverter**, **TransactionTypeToColorConverter** and others transform data for specific display needs.

For example, the **TransactionTypeToColorConverter** converts a **TransactionType** enumeration into an appropriate **Color** value, allowing dynamic coloring of transaction amounts based on their type. This offers visual cues to users about different transaction types.

3.1.6 Database Management

Utilizing SQLite for Local Database Management

SQLite is a lightweight, embedded, serverless, and self-contained database engine. It is widely used for local database management in mobile and desktop applications due to its simplicity and efficiency. SQLite databases are stored as single files and do not require a separate database server to function.

Why SQLite is chosen?

- **Minimal Setup**: SQLite databases are simple to set up and require minimal configuration. This simplicity allows developers to focus on application logic rather than database administration.
- **Portability**: SQLite databases are self-contained files, making them highly portable across various platforms and devices.
- **Serverless**: Unlike traditional database systems, SQLite operates without the need for a separate server process. This eliminates the complexities of server setup and maintenance.
- **Performance**: SQLite is known for its excellent performance, especially in read-heavy scenarios. It provides efficient data storage and retrieval mechanisms.

Enhancing SQLite with Extensions and LINQ:

The Finance Manager app maximizes the benefits of SQLite by incorporating additional libraries and language features:

- **SQLiteNetExtensions**: This library extends SQLite capabilities by adding support for relationships between tables and simplifying complex data operations. It allows for easy management of parent-child relationships and navigation through related data.
- **LINQ**: Language Integrated Query (LINQ) is used to write queries in a more human-readable and type-safe manner. It provides a natural way to express database queries and operations directly in C# code.

Usage in the Finance Manager App:

In the Finance Manager app, SQLite serves as the local database engine, facilitating the storage of various data entities such as **Account, Category, Transaction** and others. The provided repository, **BaseRepository**, abstracts the underlying database operations and offers methods for CRUD operations, data retrieval, and more. The repository class utilizes SQLite and its extensions to perform these operations efficiently.

The combination of SQLite, SQLiteNetExtensions, and LINQ empowers the Finance Manager app with a robust and efficient local database solution. It ensures smooth data management, simplifies database operations, and promotes a cleaner, more maintainable codebase by abstracting complex database interactions.

3.2 Project structure

The project structure is designed in a neat and organized way, making it easy for developers to find their way around and manage the code. It encourages breaking down tasks into separate parts and using components that can be reused, which helps create a smooth and efficient process for developing the application.

- **MVVM:** This is the core of your application's architecture. It's organized into subfolders:
    - **Models:** This folder contains essential data model classes that represent the core entities within your application. These classes, such as **Account**, **Transaction**, and others, play a pivotal role in storing and managing data. Every instance of these classes is closely tied to the application's underlying SQLite database, allowing for seamless data storage and retrieval. They hold the crucial information that drives your application's functionality, and their interactions form the foundation of your data-driven operations. All models share a common inheritance from the *TableData* base class. This base class incorporates *Id*, which automatically increments and serves as the primary key.
    - **ViewModels:** View models drive the presentation logic, and they communicate between the models and the views. These view models handle data manipulation, validation, and interaction.
    - **Views:** The Views directory houses XAML files representing the app's user interfaces. Each page or app section has a corresponding View file. Views are responsible for displaying data and user interface elements, adhering to the structure defined in the ViewModel. In the context of .NET MAUI, each View is accompanied by a corresponding C# (.cs) file with the same name. These files primarily serve to set the BindingContext to the relevant ViewModel associated with the respective View.
- **DataTemplates:** In this folder, you define data template classes that dictate how data is presented in the user interface. Unlike models, instances of these classes aren't directly saved to the database. Instead, they are created dynamically at runtime to control the visual representation of data within your views. These templates define how data from models should be displayed to the user, ensuring a consistent and visually appealing presentation. Data templates contribute to an engaging user experience and facilitate effective communication of information while keeping your database storage optimized and uncluttered.For example, **AccountDisplay** and **TransactionDisplay** might define how accounts and transactions are displayed.
- **Abstractions:** This folder contains abstract classes and interfaces that define common behaviors and contracts used throughout the application. The **BaseViewModel** provides a foundation for view models, while **IBaseRepository** and **ISelectable** define interfaces for repository operations and selectable items.

- **Constants:** In this folder, you store various constant values used in the application. **CategoryConstants**, **DatabaseConstants**, and **DisplayConstants** likely hold constants related to categories, database configurations, and display elements.
- **Converters:** Converters are responsible for transforming data between different formats, often used to display information in XAML views. The converters you've provided handle tasks like color conversions, identifier formatting, and transaction type to color mapping.
- **Messages:** Message classes in this folder are part of the messaging pattern, allowing communication between different parts of the application. These messages facilitate the exchange of information without strong coupling between components.
- **Platforms:** This folder houses platform-specific implementations for each target platform (Android, iOS, MacCatalyst, Tizen, Windows). It contains platform-specific UI and code adaptations to ensure the app works seamlessly on different devices and operating systems.
- **Properties:** This folder contains settings related to the properties of the project, including launch settings.
- **Repositories:** The **BaseRepository.cs** file defines a repository pattern to interact with the SQLite database. This separation of concerns helps manage database operations more effectively.
- **Resources:** This folder contains various resources such as images, fonts, styles, and other assets used throughout the app. The **Styles** subfolder contains XAML files defining styles for consistent UI presentation.
- **Services:** Service classes encapsulate specific functionalities and logic that don't fit well within view models. **CachedAccountsDataService**, **ColorService**, and **RecurringTransactionsService** could manage cached data, color handling, and recurring transactions, respectively.
- I*n App.xaml.cs*, the app's core setup happens. It creates instances of repositories and services like AccountsRepo and CachedAccountsDataService. The AppShell is designated as the main screen.
- In *MauiProgram.cs*, the MauiApp is created. It configures fonts, registers repositories and services, and integrates the Maui Community Toolkit. This file ensures smooth app initialization.

3.3 ViewModels Overview

3.3.1 AccountManagementViewModel

This ViewModel handles the management and creation of financial accounts. It encapsulates various properties and commands to facilitate the account creation process. The properties include the selected account icon, whether the ViewModel is in edit mode, the account being created or edited, view header, operation name (Save or Add), available account types, selected account type, and a collection of available icon glyphs. The ViewModel offers two main commands: **SaveAccountCommand** for saving account data and **CancelCommand** for canceling the operation.

Properties:

- **IsEditMode**: Indicates whether the ViewModel is in edit mode.
- **NewAccountDisplay**: Represents the account being created or edited.
- **ViewHeader**: Displays the header text for the view.
- **OperationName**: Displays the text for the operation button.
- **AccountTypes**: List of available account types.
- **SelectedAccountType**: Currently selected account type.
- **IconGlyphs**: Collection of available icon glyphs.
- **SelectedIcon**: Selected account icon.
- **AccountSavedCallback**: Callback function for account saved action.
- **OperationCanceledCallback**: Callback function for canceled operation.

Commands:

- **SaveAccountCommand**: Command to save the account data. It validates the account details and performs the saving process. If successful, it triggers the appropriate callback and closes the page.
- **CancelCommand**: Command to cancel the operation and close the page.

Constructor:

- **AccountManagementViewModel**: Initializes the ViewModel with either an existing account (in edit mode) or in normal mode for account creation. It populates the available icon glyphs and initializes the necessary properties based on the mode.

Private Methods:

- **ClosePage**: Handles the page closing, invoking the appropriate callbacks.
- **ProcessSavingInNormalMode**: Processes account saving in normal mode, ensuring proper assignment of account IDs and other properties.
- **InitializeEditMode**: Initializes the ViewModel in edit mode, populating data for an existing account.
- **InitializeNormalMode**: Initializes the ViewModel in normal mode for new account creation.

3.3.2 AccountsViewModel

The **AccountsViewModel** class serves as the intermediary between the user interface and the application's core data entities related to financial accounts. It orchestrates the retrieval, display, creation, editing, and deletion of accounts while maintaining synchronization between the view and the data. This ViewModel manages various commands, properties, and message handlers to facilitate these interactions seamlessly.

Properties

- **CurrentAccountDisplay**: Represents the current account display being processed.
- **AccountSelectedCommand**: Command executed when an account is selected.
- **SelectedAccountDisplay**: Holds the selected account display.
- **AccountLookup** and **DeletedAccountLookup**: Dictionaries for efficiently managing accounts and deleted accounts.
- **CachedAccountsData**: Holds cached account data.
- **AccountDisplays**: ObservableCollection of **AccountDisplay** instances for rendering in the view.

Commands

- **DeleteAccountCommand**: Command for deleting an account, including handling associated recurring transactions.
- **OpenAddNewAccountPageCommand**: Command to open the account management page for adding new accounts.
- **OpenEditAccountPageCommand**: Command to open the account management page for editing existing accounts.

Message Handlers

- **HandleAccountsRequest** and **HandleDeletedAccountsRequest**: Handlers for responding to requests for accounts and deleted accounts data.
- **OnTransactionAdded** and **OnTransactionUpdated**: Handlers for processing added and updated transactions.

Methods

- **InitializeAsync**: Initializes the ViewModel by retrieving, processing, and registering relevant data.
- **ProcessAccounts** and **ProcessDeletedAccounts**: Processes existing accounts and deleted accounts data.
- **RecalculateTotalBalance**: Recalculates the total balance based on account balances.

- **HandleTransactionAddedAsync** and **HandleTransactionUpdatedAsync**: Handlers for processing transaction changes.
- **ChangeTotalBalanceAsync**: Updates the cached total balance.

Database Interaction

- Methods for saving, updating, and deleting account and account view data.
- Methods for retrieving accounts and deleted accounts data from the database.

Data Generation

- **FillDataAsync**: Generates and saves test account data if required.
- **GenerateNewAccount** and **GenerateNewAccountView**: Generates new account and account view instances.

CollectionExtensions

- Helper extensions for converting ObservableCollection and List to readable string representations.

This ViewModel plays a crucial role in managing the presentation, creation, and manipulation of financial account data in the application, ensuring a smooth user experience and consistent data synchronization.

3.3.3 TransactionManagementViewModel

The **TransactionManagementViewModel** class facilitates the management of financial transactions, allowing users to create or edit transactions with associated accounts, types, and categories. It communicates with the UI, orchestrates data retrieval, and handles user interactions related to transaction management.

Properties

- **currentAccount**: Holds the current account being processed.
- **selectedTransactionType**: Represents the selected transaction type (Income, Expense, Transfer).
- **IsTransactionCreating**: Indicates whether a new transaction is being created.
- **IsEditMode**: Indicates whether the ViewModel is in edit mode.
- **Messenger**: Provides messaging functionality for inter-component communication.
- **NewTransactionDisplay** and **OldTransactionDisplay**: Objects representing new and existing transaction displays.

- **IncomeCategories** and **ExpenseCategories**: Collections of categories for income and expense transactions.
- **CurrentCategories**: Holds the current categories based on the selected transaction type.
- **SelectedTime**: Represents the selected time for the transaction.
- **AccountLookup**: A dictionary for efficiently managing accounts.
- **Accounts**: ObservableCollection of available accounts.
- **TransactionTypes**: ObservableCollection of available transaction types.
- **IsTransfer**: Indicates whether the transaction is of type Transfer.
- **SelectedSourceAccount** and **SelectedDestinationAccount**: Selected source and destination accounts.
- **SelectedIncomeCategory** and **SelectedExpenseCategory**: Selected income and expense categories.
- Various placeholder properties for UI labels.

Commands

- **SelectCategoryCommand**: Command for selecting a transaction category.
- **SelectSourceAccountCommand** and **SelectDestinationAccountCommand**: Commands for selecting source and destination accounts.
- **SaveTransactionCommand**: Command for saving a transaction.
- **CancelTransactionCommand**: Command for canceling the transaction creation or edit process.

Methods

- **SelectCategory**: Method for selecting a transaction category.
- **UpdateUIForTransactionType**: Updates the UI based on the selected transaction type.
- **ClosePage**: Closes the transaction management page and resets data.
- **ProcessTransaction**: Processes and saves the transaction data.
- **SaveTransactionAsync** and **UpdateTransactionAsync**: Methods for saving and updating transactions.
- **FillInTransactionData**: Fills in transaction data based on user selections.
- **ResetPageData**: Resets page data after transaction processing.
- **GetCategories**: Retrieves income and expense categories.

Event Handlers

- **OnAccountDeleted**, **HandleAccountResponse**, and **OnAccountAdded**: Handlers for account-related messages.

Initializing

- **InitializeTransaction**: Initializes the ViewModel for transaction creation or edit.
- **InitializeEditMode** and **InitializeNormalMode**: Initializes ViewModel for edit or normal mode.
- **OnAppearing**: Handles ViewModel behavior when the associated view is displayed.

This ViewModel manages the creation, editing, and processing of financial transactions, ensuring data accuracy and user-friendly interactions. It communicates with the UI, handles user inputs, and interacts with the data model to provide a seamless transaction management experience.

3.3.4 TransactionsViewModel

The **TransactionsViewModel** class manages the presentation of financial transactions grouped by day. It facilitates data retrieval, formatting, and UI interaction for displaying transactions with associated details. This ViewModel ensures proper communication between the UI and data sources, providing a user-friendly and organized display of transactions.

Properties

- **DayTransactionGroups**: ObservableCollection of **DayTransactionGroup**, representing transactions grouped by day.
- **Messenger**: Provides messaging functionality for inter-component communication.
- **TransactionLookup**: A dictionary for efficiently managing transactions.
- **TransactionDisplays**: ObservableCollection of **TransactionDisplay** objects, representing transaction data.
- **AccountLookup**, **AccountNameExtractorLookup**, **CategoryNameExtractorLookup**, and **DeletedAccountLookup**: Dictionaries for efficiently managing account and category data.
- **CategoryLookup**: Dictionary for efficiently managing category data.
- **accountsReceivedTcs** and **deletedAccountsReceivedTcs**: **TaskCompletionSource** objects for handling asynchronous account data retrieval.

Constructor

- **TransactionsViewModel**: Initializes the ViewModel and triggers asynchronous initialization.

Event Handlers

- **OnAccountDeleted**, **OnTransactionAdded**, **OnAccountUpdated**: Handlers for account and transaction-related messages.
- **HandleAccountResponse**, **HandleDeletedAccountResponse**: Handlers for account-related response messages.

Private Methods

- **ProcessTransactions**: Processes transactions after ensuring account and deleted account data is received.
- **InitializeAsync**: Initiates ViewModel initialization, messaging, and data processing.
- **GetDayTransactionGroupsAsync**: Asynchronously retrieves and groups transactions by day.
- **AddTransactionDisplayToGroup**: Adds a transaction display to the appropriate day's group.
- **GetTransactionDisplaysAsync**: Asynchronously retrieves transaction displays.
- **GetTransactionDisplay**: Constructs a **TransactionDisplay** object with associated details.
- **ExtractAccountName** and **ExtractCategoryName**: Extracts account and category names using lookup dictionaries.
- **GetCategories**: Retrieves category data.
- **GetTransactionsAsync**: Asynchronously retrieves transactions from the data repository.

Initializing

The ViewModel initializes by registering event handlers and sending messages to request account and deleted account data. After receiving the required data, it processes transactions, groups them by day, and constructs the necessary display structures for presenting transaction data.

Presentation and Organization

The ViewModel primarily focuses on organizing and presenting transaction data. It ensures that transactions are properly grouped by day and displayed in a user-friendly manner. The ViewModel leverages dictionaries and messaging to efficiently manage and retrieve data, promoting seamless and organized interactions with the UI.

Overall, the **TransactionsViewModel** serves as a central hub for managing and presenting financial transactions, ensuring data accuracy, proper grouping, and user-friendly interaction in the application.

3.4 Main solutions used in Personal Finance Manager

Fody

The Finance Manager app employs the **Fody package** to automatically implement the **INotifyPropertyChanged** interface. Through Fody, the PropertyChanged events for properties are automatically generated, guaranteeing synchronized UI updates with data modifications. This integration eliminates the necessity for manual event invocation, streamlining property change notifications and optimizing the data binding process.

Model, ModelView, ModelDisplay

In the architecture of Finance Manager, a design pattern is employed that leverages the concepts of models, model views, and model displays. This approach fosters a well-structured organization, seamless interaction with data, and efficient data binding for the user interface.

1. **Models:** Within the Finance Manager application, models serve as the foundational representations of essential data entities. Each model encapsulates the crucial attributes and functionalities associated with its respective entity.
2. **Model Views:** Finance Manager's model views extend the capabilities of models by introducing additional attributes tailored for visual presentation and user interaction. Attributes such as icons and background colors contribute to an enhanced visual representation of the data.
3. **Model Displays:** Acting as intermediaries between core data models and the user interface, model displays play a pivotal role in Finance Manager. They harmoniously combine data from models with presentation-related attributes from model views. This amalgamation is particularly beneficial for data binding in the XAML interface. The resultant unified representation offers a fluid user experience, effectively integrating essential data with visual attributes.

Overall, the adoption of the models, model views, and model displays paradigm in Finance Manager cultivates a structured and modular approach to data management and presentation. This segregation of core data and presentation-specific attributes enhances the application's flexibility and maintainability.

Commands

In the development of the Finance Manager project, an integral aspect of the architecture involves the strategic incorporation of commands within view models. Commands serve as essential mechanisms for encapsulating user interactions and enabling seamless communication between the user interface and the underlying application logic.

Each command within the view models of Finance Manager represents a specific action that users can initiate through controls in the user interface. This action-oriented approach promotes a well-structured separation of concerns, allowing the view models to manage and execute tasks based on user input. The commands facilitate the execution of operations such as saving data, triggering specific functionalities, or navigating between different sections of the app.

Binding these commands to controls in the user interface further streamlines the interaction between users and the application. By binding commands to user interface elements like buttons, menu items, or entries, Finance Manager ensures that user-initiated actions seamlessly trigger the corresponding commands in the view models. This cohesive integration empowers the application to respond promptly and accurately to user inputs.

4. Conclusion

In conclusion, the Personal Finance Manager Application offers a robust solution to the intricate challenges associated with personal finance management. By seamlessly combining technological prowess with user-centric design, the application empowers users to take control of their finances, fostering financial literacy and stability. The strategic utilization of the Model-View-ViewModel (MVVM) architectural pattern ensures a modular and organized codebase, facilitating efficient maintenance and future scalability.

The application's comprehensive feature set, ranging from account management to transaction tracking, equips users with the tools needed to make informed financial decisions. The user-friendly interface, accommodating both light and dark themes, enhances usability and accessibility across diverse preferences.

Built upon the solid foundation of .NET MAUI and C#, the Personal Finance Manager Application underscores not just technological progress but also a commitment to elevating users' financial well-being. In an era where effective financial management is paramount, this app exemplifies the potential of technology-driven solutions, empowering users to confidently navigate their financial journeys.

5. References:

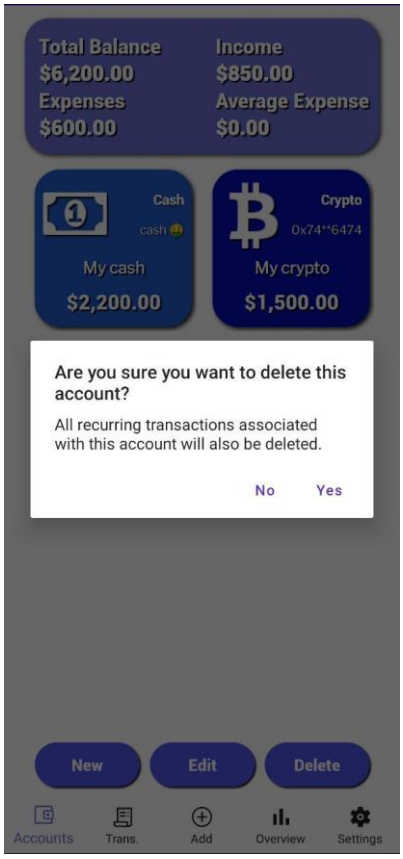
1. .NET MAUI Documentation: Microsoft's official documentation for .NET Multi-platform App UI (MAUI), providing comprehensive insights into building cross-platform applications using C# and XAML. Website: https://docs.microsoft.com/en-us/dotnet/maui/
2. PropertyChanged.Fody: A library that automates the implementation of INotifyPropertyChanged, GitHub Repository: https://github.com/Fody/PropertyChanged
3. SQLite Documentation: Website: https://docs.microsoft.com/en-us/dotnet/standard/data/sqlite/
4. .NET MAUI Data Binding: A comprehensive guide to data binding in .NET MAUI applications, detailing how to establish connections between UI elements and data sources. Article: https://learn.microsoft.com/en-us/dotnet/maui/fundamentals/data-binding/basic-bindings/
5. .NET MAUI Community Toolkit: An open-source collection of essential utilities and controls for .NET MAUI applications, facilitating common tasks and UI components. GitHub Repository: https://github.com/CommunityToolkit/Maui/

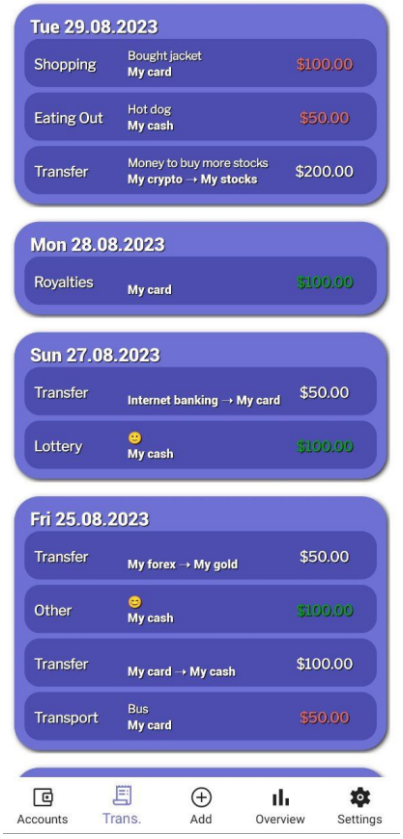
6. Screens


Total Balance
$6,200.00
Income
$850.00
Expenses
$600.00
Average Expense
$0.00
Cash
cash
My cash
$2,200.00
Crypto
0x74**6474
My crypto
$1,500.00
Debit Card
gh
My card
$2,500.00
New
Edit
Delete
Accounts
Trans.
Add
Overview
Settings


Fill in your account details
Name
Print Account Name
Balance
0
Identifier
(For example, a credit card number)
Print Identifier
Type
Card
Credit Card
Debit Card
Print your own type
Select the icon
Add
Cancel
Accounts
Trans.
Add
Overview
Settings


Edit your account details
Name
My card
Balance
2500
Identifier
(For example, a credit card number)
gh
Type
Card
Credit Card
Debit Card
Debit Card
Select the icon
VISA
PayPal
Save
Cancel
Accounts
Trans.
Add
Overview
Settings


Total Balance
$6,200.00
Income
$850.00
Expenses
$600.00
Average Expense
$0.00
Cash
cash
My cash
$2,200.00
Crypto
0x74**6474
My crypto
$1,500.00
Are you sure you want to delete this account?
All recurring transactions associated with this account will also be deleted.
No
Yes
New
Edit
Delete
Accounts
Trans.
Add
Overview
Settings


Tue 29.08.2023
Shopping
Bought jacket
My card
$100.00
Eating Out
Hot dog
My cash
$50.00
Transfer
Money to buy more stocks
My crypto → My stocks
$200.00
Mon 28.08.2023
Royalties
My card
$100.00
Sun 27.08.2023
Transfer
Internet banking → My card
$50.00
Lottery
My cash
$100.00
Fri 25.08.2023
Transfer
My forex → My gold
$50.00
Other
My cash
$100.00
Transfer
My card → My cash
$100.00
Transport
Bus
My card
$50.00
Accounts
Trans.
Add
Overview
Settings


Income
Expense
Transfer
Date
8/29/2023 11:26 PM
One-time
Account
Select an Account
Category
Amount
0
Note
Accounts
My cash
My crypto
My card
Save
Cancel
Accounts
Trans.
Add
Overview
Settings

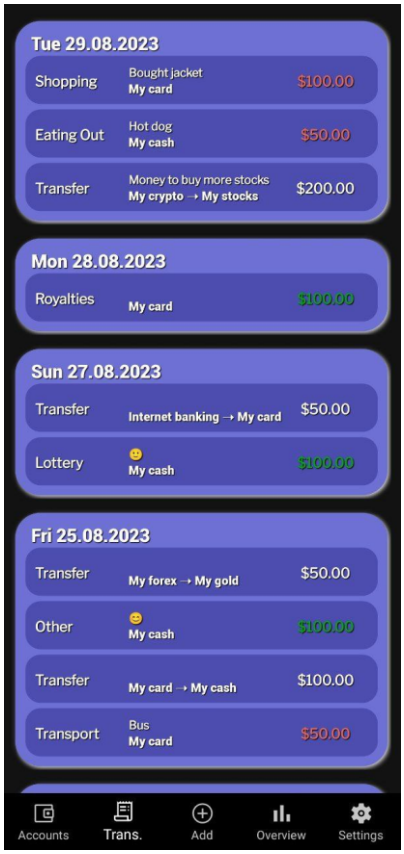

Income
Expense
Transfer
Date
8/29/2023 11:26 PM
One-time
Account
My cash
Category
Health
Amount
0
Note
Categories
Food
Housing
Transport
Eating Out
Health
Fun
Sports
Shopping
Education
Savings
Save
Cancel
Accounts
Trans.
Add
Overview
Settings


Income
Expense
Transfer
Date
8/29/2023 11:26 PM
One-time
From
My cash
To
My card
Amount
68
Note
Save
Cancel
Accounts
Trans.
Add
Overview
Settings


Total Balance
$6,200.00
Income
$850.00
Expenses
$600.00
Average Expense
$0.00
Cash
cash
My cash
$2,200.00
Crypto
0x74**6474
My crypto
$1,500.00
Debit Card
gh
My card
$2,500.00
New
Edit
Delete
Accounts
Trans.
Add
Overview
Settings


Income
Expense
Transfer
Date
8/29/2023 11:28 PM
One-time
Account
Select an Account
Category
Amount
0
Note
Accounts
My cash
My crypto
My card
Save
Cancel
Accounts
Trans.
Add
Overview
Settings


Income
Expense
Transfer
Date
8/29/2023 11:28 PM
One-time
Account
My card
Category
Select a Category
Amount
0
Note
Categories
Salary
Business
Investments
Rental
Dividends
Pension
Interest
Freelance
Sale of Assets
Social Security
Save
Cancel
Accounts
Trans.
Add
Overview
Settings


Tue 29.08.2023
Shopping
Bought jacket
My card
$100.00
Eating Out
Hot dog
My cash
$50.00
Transfer
Money to buy more stocks
My crypto → My stocks
$200.00
Mon 28.08.2023
Royalties
My card
$100.00
Sun 27.08.2023
Transfer
Internet banking → My card
$50.00
Lottery
My cash
$100.00
Fri 25.08.2023
Transfer
My forex → My gold
$50.00
Other
My cash
$100.00
Transfer
My card → My cash
$100.00
Transport
Bus
My card
$50.00
Accounts
Trans.
Add
Overview
Settings