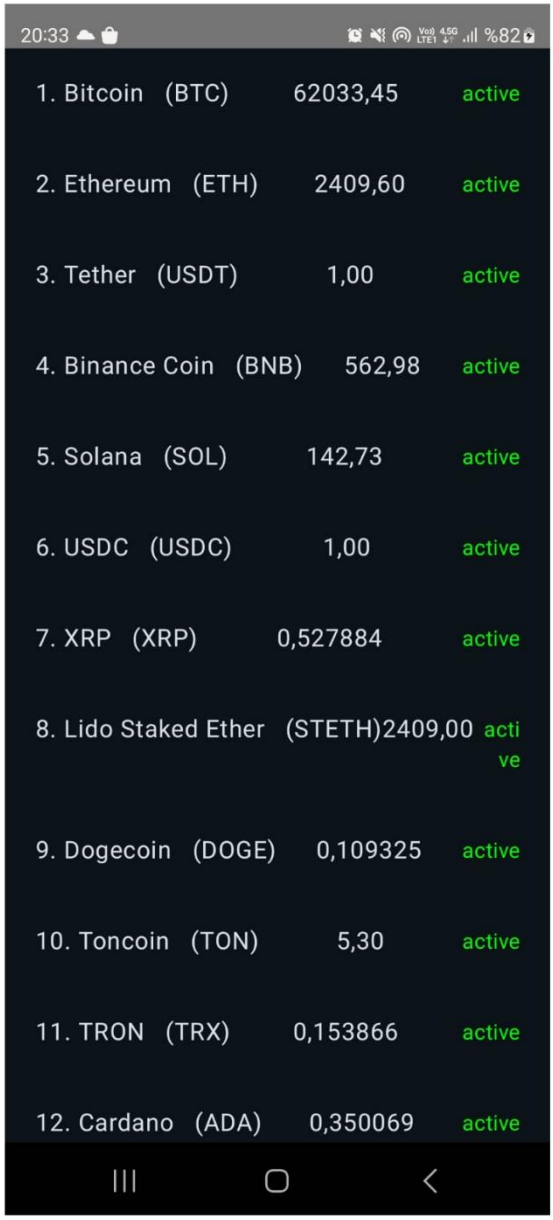
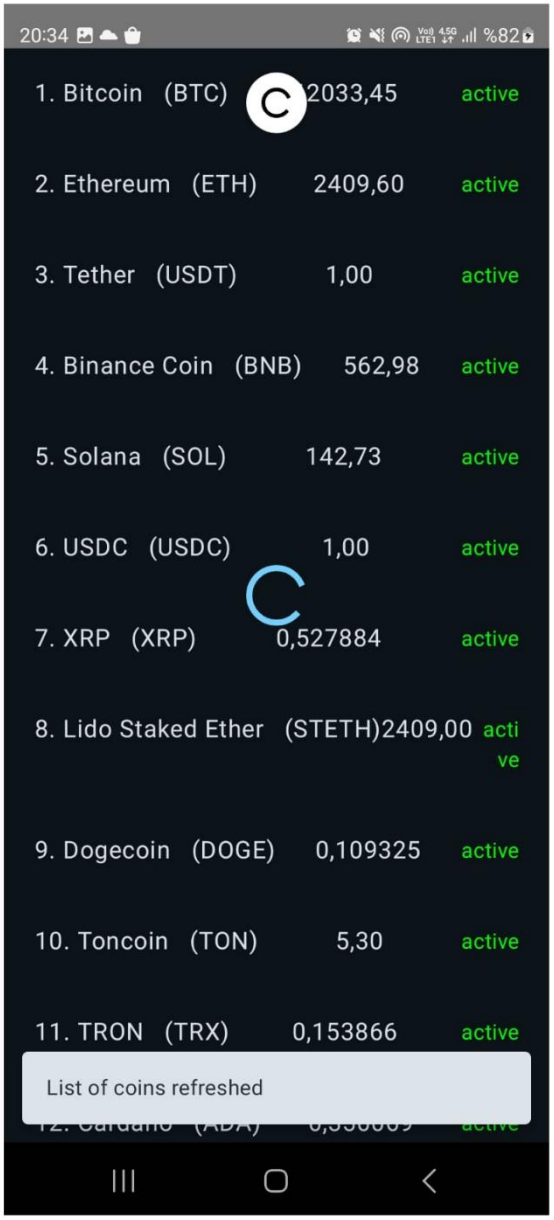
INTRODUCTION

This is the first Kotlin app I developed using the Jetpack Compose library. The app displays a list of cryptocurrencies on the main screen, showing each coin's name, symbol, and current price, along with whether the coin is active or not. Coins are sorted by their market capitalization on the main screen. Also, user can refresh the list and get the new prices if they are changed.

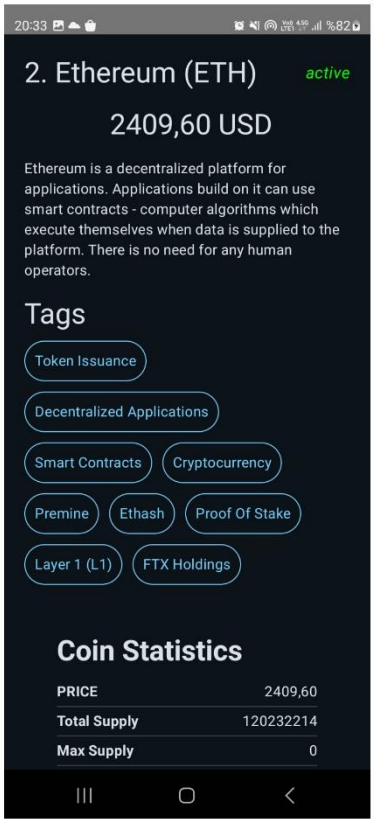
Main Screen:

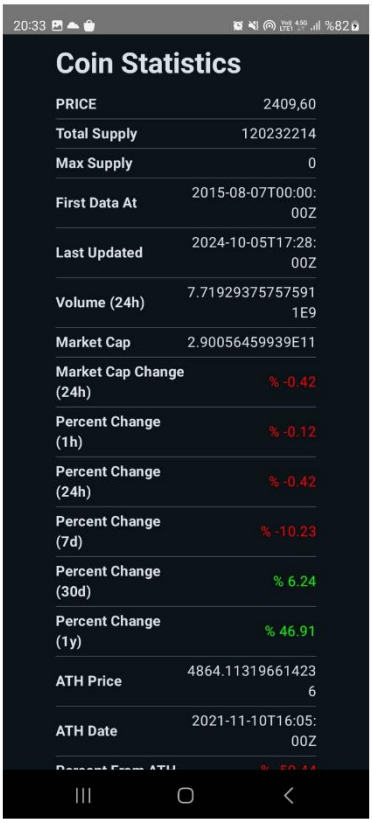


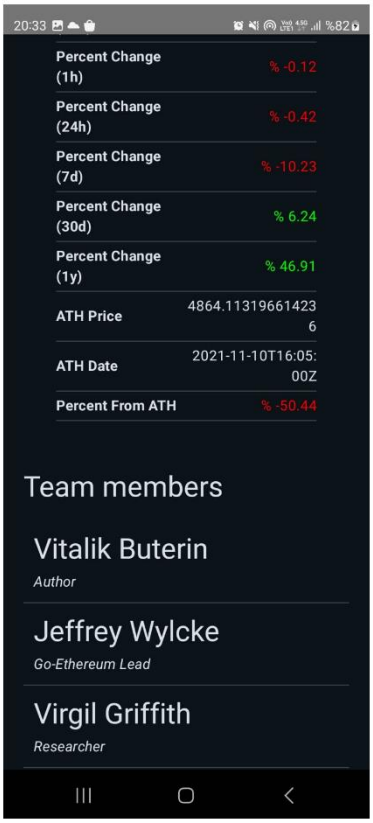


If a user wants to learn more about a coin, they can select it from the list to view a detailed screen with information about the coin's technology, price, tags, and, if available, the team members involved in its creation. The app also displays detailed statistics for each cryptocurrency, including total supply, max supply, first recorded data, last updated time, 24-hour trading volume, market capitalization, market cap change in the last 24 hours, percentage changes over 1 hour, 24 hours, 7 days, 30 days, and 1 year, all-time high (ATH) price, ATH date, percentage difference from the ATH price, and the current price of the coin.

Information Screen about a coin:







Jetpack Compose is used for the UI, Retrofit for API calls, and Dagger Hilt for dependency injection. When I developed the app, I followed clean architecture principles and implemented the MVVM (Model-View-ViewModel) architecture pattern. Additionally, several design patterns were used throughout the development process. Each concept has its own section, and I've tried to explain everything in a way that even those without Android or development experience can understand. I hope it's clear and accessible.

ARCHITECTURE OVERWİEW

First, let me explain what clean architecture is. Clean Architecture is a design pattern that organizes code into layers, promoting separation of concerns and making the app easier to maintain and test. The main layers are:

Presentation Layer: This layer contains UI components and handles user interactions. It communicates with the next layer to retrieve data.

Domain Layer: This layer contains the business logic and use cases. It defines the rules and processes for the application without being tied to the UI or data sources.

Data Layer: This layer handles data management and retrieval, including interactions with databases and APIs. It provides the necessary data to the domain layer.

By structuring the app in this way, each layer is independent and can be modified or replaced without affecting others, leading to cleaner, more manageable code.

In this app, I have used MVVM architecture pattern. Let’s look at it.

Model — View — ViewModel (MVVM) is a widely-used architecture pattern in the industry. MVVM separates the data presentation (UI) from the application's business logic, ensuring a clean structure.

Model: Handles the data sources and works with the ViewModel to manage data retrieval and saving.

View: Notifies the ViewModel of user actions and observes it for updates, without holding any application logic.

ViewModel: Acts as a bridge between the Model and View, providing the data needed by the UI.

DESIGN PATTERNS

I will not explain diverse design patterns such as creational, structural, and behavioural patterns because I didn’t use most of them. Here are the patterns I have used in this app:

Singleton: It is an object only a single instance of exists in our code. It can be reached out anywhere. Multiple instances can not be created. If we use it with Hilt, it will create our functions or class for the throught the app, then when we want to use them, same instance will be returned to us.

Facade: Basically, it is sweeping your code under the carpet. It is easier to explain it with an example:

```
interface api
{
        @GET(/v1/coins)
        suspend fun getCoins() : List<Coins>
}
```

Here @Get and retrofit handles the API request and the function, we do not provide any code more than that. Other developers who read your code only see the top of this iceberg, don’t care about the rest. This is facade design pattern.

Builder: You prepare your meals with ingredients which you want, right? Builder does it. Choosing the properties of your classes which you want to build is builder design pattern. It is often used in Retrofit and Room.

Dependency Injection (DI) is like comparing ready-made chocolate milk with regular milk. With regular milk, you can make banana milk because you can add bananas and mix it in a shaker. But you can't make banana milk with chocolate milk, right? Because chocolate milk is already flavoured with chocolate, and you can’t change its flavour. Dependency Injection is like choosing regular milk and mixing it with your favourite flavours. DI lets you decide what ingredients to use, keeping things flexible.

LIBRARIES AND TOOLS

Jetpack Compose: Jetpack Compose allows us to define UI components in a declarative way directly in Kotlin code. Normally, we would create UI components using XML files, but with Compose, we do it in our code using composable functions. It simplifies UI development and makes it much more enjoyable. Additionally, Compose makes it easy to manage state efficiently and eliminates the need for XML layouts, which is especially useful for handling dynamic UI changes.

Retrofit: Retrofit is used to make network requests. Retrofit is a type-safe HTTP client library that allows us to define API endpoints easily using Java/Kotlin annotations. It lets us specify query parameters, request bodies, headers, etc. It also automatically handles JSON serialization to ensure type safety. Plus, Retrofit supports different converters like Gson or Moshi and works well with Coroutines for making asynchronous requests.

Dagger Hilt: Dagger-Hilt is a dependency injection library for Android that reduces boilerplate code and allows us to inject our dependencies easily. It's built on top of Dagger and simplifies the setup of dependency injection. Hilt also helps manage lifecycle-aware components like ViewModel, Activity, and Fragment, making DI setup even more convenient.

Coroutines: Coroutines are also used. Let me explain how and what is a coroutine. A coroutine allows tasks to pause and resume without freezing the program. Unlike traditional threads, coroutines wait for things (like data) without stopping the app. This keeps the app responsive while background tasks continue, and once ready, the coroutine picks up from where it paused. For example: Imagine you're cooking. While waiting for water to boil, instead of just standing there, you chop vegetables. You're not wasting time, and both tasks happen efficiently. Coroutines in programming do the same—they let a program pause one task (like waiting for data) and work on another, without stopping

STATE MANAGEMENT

The app consists of two main screens: one that displays a list of cryptocurrencies and another that shows detailed information about a selected coin. For the coin list screen, a data class manages whether data is being loaded, holds the list of coins to be displayed, and provides an error message if something goes wrong. Similarly, the coin details screen has a data class that indicates if data is loading, holds the detailed information for the selected coin, and provides an error message if there's an issue retrieving that data. In all cases, the app ensures that the user is always informed. While data is being fetched, a spinner is displayed to indicate loading. Once data is successfully loaded, the relevant information is shown. If an error occurs, a message explains the problem. This approach guarantees that users always know what's happening, whether the data is loading, successfully retrieved, or an issue arises. The app is prepared to handle all situations smoothly.

ERROR HANDLING

If there is a problem with the internet connection or communication with the server, I catch these errors and display them on the screen to inform the user why they encountered the issue. For example, if the user loses their internet connection, a message will be shown that says, "Check your internet connection." If the server sends a bad response, which can happen if the user exceeds the API search limit or the server fails to send data, an error message will be shown again.

CONCLUSION

In this documentation, I have explained how my application is structured and designed, as well as the design choices I have made. I've also outlined the tools and libraries used, making it accessible for everyone, even those who aren't familiar with Android development. The app uses Jetpack Compose for the UI, Retrofit for API calls, Dagger Hilt for dependency injection, and Coroutines for handling background tasks. I've implemented MVVM (Model-View-ViewModel) architecture and Clean Architecture principles to keep the code organized and easier to work with.

I've learned a lot during this project and am excited to keep improving my skills in Kotlin and Android development. If you have any suggestions, feel free to reach out to me. Also, if you're interested in seeing the code, you can check it out on my GitHub page. Thanks for reading, and I hope you find this documentation helpful.

KEREM GÖBEKCİOĞLU