FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Demo aplikace pro ověření funkčnosti ClientPortal API
Student:	Daniil Poletaev
Vedoucí:	Ing. Jan Trdlička, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2022/2023

Pokyny pro vypracování

1) Seznamte se s Client Portal API pro komunikaci uživatelské aplikace s Interactive Brokers.
2) Navrhněte a implementujte aplikaci pro iPadOS, která bude demonstrovat funkčnost tohoto API. Aplikace by měla umět:
   a) přihlásit se na účet u Interactive Brokers,
   b) zobrazit informace o účtu,
   c) načíst a zobrazit popis různých zadaných instrumentů (Forex páry, akcie opce, ...),
   c) načíst historická i reálná data těchto instrumentů a vhodným způsobem je zobrazit,
   d) zadávat obchodní příkazy a zobrazit jejich přehled.
3) Aplikaci otestujte a porovnejte vlastnosti Client Portal API s původní Trader Workstation (TWS) API.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. října 2021 v Praze.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Demo application for verifying the functionality of the ClientPortal API

Daniil Poletaev

Department of Software Engineering
Supervisor: Ing. Jan Trdlička, Ph.D.

May 9, 2022

Acknowledgements

I want to thank my supervisor Ing. Jan Trdlička, Ph.D. for supervising my work and giving beneficial advice, which helped immensely in my bachelor's thesis. I also really appreciate my family and friends for helping me, supporting me during my studies, and believing in me. Without their care, it would be tough for me to cope with all the hardships of studying. I also want to thank all the professors of Czech Technical University who taught me a lot of practical skills, which I use and will continue to use in my life.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 9, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Poletaev, Daniil. *Demo application for verifying the functionality of the Client-Portal API*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

This bachelor's thesis aims to implement a mobile demo application on iOS to show features of relatively new technology Interactive Brokers ClientPortal API. The demo application allows users to use Interactive Brokers' services, such as buying, selling, and analyzing investment instruments. I suggested the application architecture, designed interfaces, implemented functionality, and tested the final application. Based on this demo application, investors and traders can create their own, custom, application for iOS gadgets

Keywords Interactive Brokers API, Client Portal API, Interactive Brokers IOS, Client Portal API iOS, Mobilní aplikace pro iOS, iOS mobilní aplikace, demo iOS mobilní aplikace

Abstrakt

Cílem této bakalářské práce je vyvinutí mobilní demo aplikace pro iOS/iPadOS, díky které lze ukázat vlastnosti relativně nové technologie Interactive Brokers ClientPortal API. Demo aplikace umožňuje uživatelům využívat služby populárního amerického makléře Interactive Brokers, nakupovat, prodávat a analyzovat investiční nástroje. V rámci práce byla navržena architektura a rozhraní aplikace, které byly následně implementovány a otestovány. Na základě této demo aplikace mohou následně investoři a obchodníci vytvářet vlastní aplikace pro zařízení iOS/iPadOS.

Klíčová slova Interactive Brokers API, Client Portal API, Interactive Brokers IOS, Client Portal API IOS, Mobile app for iOS, iOS mobile app, demo iOS app

Contents

List of Figures

List of Tables

Introduction

Nowadays, more and more people have started investing in stocks, options, and other investment instruments worldwide. Investing 50 years ago and today is different. Past investors had to meet face–to–face with the broker each time they wanted to buy or sell their portfolio positions. Thanks to the technologies of today's world, all people have the opportunity to invest within a few clicks, not only on their computers but also on their mobile gadgets. Today a trader can buy an option while riding the subway to work. Moreover, even though there are a lot of mobile applications for brokers, more professional investors and traders often try to build custom applications suitable for their needs.

My motivation is to create a demo application of ClientPortal API provided by one of the most popular brokers in the United States and worldwide so that enthusiastic market players can build their custom applications referencing this one.

In this bachelor's thesis, I suggest an architecture, design, implement and test mobile iOS application and the ClientPortal API. This demo application communicates with real-world Interactive Brokers API - so with the help of this demo application, investors can buy, sell and analyze investment instruments provided by this broker.

In the first chapter, I research information about Interactive Brokers and describe and compare all of their existing API solutions. In the second chapter, I describe different technologies used for building iOS apps, point out their advantages and disadvantages, and choose solutions for making the application. In the third chapter, I analyze existing solutions, provide functional and non-functional requirements, and use case diagram. In the fourth chapter, I compare the two most popular programming architectures used for programming iOS applications, choose one of them and describe the application's presentation layer. In the fifth chapter, I describe the implementation of business logic in the app, applying the principles and technologies I described in previous chapters. In the final, sixth part, I describe the testing process of the application.

Thesis aim

The primary aim of this bachelor thesis is to create a demo iOS/iPadOS application demonstrating the functionality of relatively new Client Portal API technology provided by Interactive Brokers. Based on this demo project every trader will be able to create a custom trading application, which will satisfy his or her needs.

The research part aims to analyze existing solutions, compare different API technologies provided by Interactive Brokers, and describe the advantages and disadvantages of technologies used for developing iOS/iPadOS apps.

The practical part aims to propose an application, design functional and non-functional requirements and user interface, choose and compare architectures, implement a responsive mobile application and cover it with automated tests.

Chapter 1

Interactive Brokers API Overview

In this chapter, I describe ClientPortal API and compare it with other technologies provided by the Interactive Brokers to communicate with their services.

1.1 Interactive Brokers (IB)

Interactive brokers are one of the most popular brokers used worldwide. *"Interactive Brokers is ideal for institutional investors and sophisticated, active traders who want a robust trading platform and access to a long list of asset classes."* [5]

By the Investopedia, this broker is top-rated among lots of nominations such as: "Best Online Broker for Advanced Traders", "Best Online Broker for Day Trading", "Best Broker for Low Margin Rates", "Best Broker for Fractional Shares", "Best Online Broker for Non-U.S. Investors" and many more. [6]

The company's history starts in 1977, when the Chairman of the Interactive Brokers Group, Thomas Petterffy, bought a seat on the American Stock Exchange. Soon in 1995, Interactive brokers created their first platform and provided access for their customers to buy and sell their financial assets. [5]

As of nowadays, the primary mission of IB remains unchanged: *"Create technology to provide liquidity on better terms. Compete on price, speed, size, diversity of global products and advanced trading tools."* [5]

1.2 IB Trader Workstation (TWS)

TWS is the trading platform provided by Interactive Brokers for market enthusiasts, who want to take additional control of the features of trading and

investing. With the help of this powerful interface, traders can automate their trading strategies, and get market data and information about their account balance and portfolios. Moreover, with the help of this platform, investors can buy and sell over 150 different worldwide markets. [1]

Furthermore, TWS provides access to over 100 order types and trading algorithms and helps traders manage their risk.[1]

On the listing below you can take a look, at what the TWS interface looks like.

Figure 1.1: Trader Workstation Interface [1]

1.2.1 Trading Workstation API (TWS API)

TWS also allows traders to create custom applications and connect them to the workstation to take additional control and use advanced trading tools. It supports many different programming languages, such as C++, C#, Java, Python, ActiveX, RTD, DDE, and many more. With the help of TWS API developers can get up-to-minute market data, and trade different financial instruments worldwide including stocks, options, futures, currencies, bonds, and funds. [1]

Moreover, TWS API allows getting real-time news from leading services, world-class analyst research, financial information on thousands of companies, and event calendars. [1]

1.3 Financial Information eXchange (FIX)

"The Financial Information eXchange (FIX) is a vendor-neutral electronic communications protocol for the international real-time exchange of securities transaction information." [7]

FIX protocol is mostly used for B2B communication to improve business messages and transaction flow. It was created by a non-profit firm to ensure, that this protocol is in the public domain. [7]

Interactive Brokers also provides this protocol, but only for firms. FIX API technology allows creating trading systems with a high speed of processing transactions. [8]

1.4 Client Portal API

Client Portal API is the newest technology offered by Interactive Brokers nowadays. It provides a modern technologies for fast and easy communication with Interactive Brokers. [8]

This API provides more detailed information about user's account such as portfolio, balances, statements, transaction history and even notifications. Some of this information can not be fetched using other provided technologies. It also allows traders to analyse market and history data of stocks, options, futures, etc. [9]

The big advantage of this technology is that developers can fetch data from server not only with REST API, but also with sockets, which are crucially important for real-time information, which is being updated within seconds.

Via REST API traders can get account, contract, balances, ticket information and market and history data. Also via this interface, developers can buy and sell financial instruments and track orders' statuses.

Via WebSocket API developers can access real-time market and history data, track orders, get profit and loss information and watch after accounts' notifications. [10]

To start using ClientPortal API no additional libraries required. Only the gateway must be downloaded and started.

1.5 Overall Comparison

In the last section of this chapter, I will compare all the technologies offered by Interactive Brokers, and the best comparison is the table of all features of different APIs.

Table 1.1: API Feature Comparison [4]

API Feature Comparison			
Feature	TWS API	Client Portal API	FIX CTCI
Place Trades	+	+	+
User Authentication	+	+	
OAuth		+	
View Positions	+	+	
View Orders	+	+	+
Profit and Loss	+	+	
Market Data	+	+	
Tick by Tick Data	+		
Real-Time Drop Copy			+
News	+		
Account List	+	+	
Security Definition	+	+	
Aggregate User Support			+

In my bachelor's thesis, I will mainly describe the features of the Client Portal API and how it can be used to build custom mobile applications.

Chapter 2

User interface frameworks for iPadOS/iOS

There are some different technologies with which iOS applications can be built. Some of them are: Swift, React Native, Flutter, and Xamarin.

In this chapter, I am going to compare two of these technologies: cross-platform technology - React Native and Swift - technology to build native applications running on iOS.

2.1 React Native

React Native is a cross-platform technology that allows developers to write code once for different platforms such as Android, iOS, and Web. It was released by Facebook and gained popularity in the last few years.

Even if it seems that React Native is used to build non-complex applications, many big firms such as Instagram, Facebook, and Skype have some modules/screens implemented in this technology.

2.1.1 Advantages

Build once - use in Android, iOS, or Web. React Native can be built only once for different operational systems. Only one team is needed to build iOS and Android applications.

Big ecosystem. There is a vast ecosystem where developers can find almost any package they need. It helps speed up the development process because programmers won't need to write some module when they can import them.

Fast-reloading on any changes. While making some changes app automatically refreshes within a few seconds. There is no need to rebuild the application each time the text or some feature is being changed.

Less expensive. As only one team is needed, React Native apps can be less expensive than native applications, but this depends on the app's complexity.

It can be used in an existing project. There is no need to rewrite the whole application if it already exists. React Native can be imported and used in existing applications.

2.1.2 Disadvantages

Hard to debug. It is harder to debug applications because no debugger can show the change in real-time.

It may be dependent. React Native apps can become a nightmare if more and more packages are being added to the app. Some of those packages use other packages, which may lead to some problems while upgrading some packages. Furthermore, most of the modules are being developed by enthusiasts, which sometimes don't have time to update and fix existing packages, which can be full of bugs.

Performance problems. When an app grows more and more complex, React Native applications can show a significant difference in performance compared to native applications.

2.2 Swift

Swift is a relatively new programming language designed by Apple and released on June 14, 2014. Before Swift, most of the native applications for Apple gadgets were built using Objective-c language. As Apple states: *"Swift is the result of the latest research on programming languages, combined with decades of experience building Apple platforms."* [11] This programming language was built specifically for building iOS, iPadOS, macOS, tvOS, and watchOS native applications.

As of 2022, building apps for iOS on Swift is divided into two frameworks - SwiftUI and UIKit. Both of these frameworks offer lots of advantages as well as disadvantages. But the good thing about these frameworks is that they can complement each other. A developer can add some modules written in SwiftUI for an existing app built on UIKit and vice versa.

2.2.1 UIKit

UIKit is the predecessor to SwiftUI. UIKit was publicly released in 2008 and helped Swift gain popularity. The significant difference between UIKit is that this framework is more visual-based, meaning that programmers can build UIScreen using something like a constructor.

2.2.1.1 Advantages

Consistent experience. Building apps using UIKit offers a consistent experience among all gadgets.

Easy to find a solution As UIKit was publicly released in 2008, it already has some best practices formed, and developers can easily find a solution to their problems on the internet.

2.2.1.2 Disadvantages

Views are expensive. View in UIKit is much more expensive than views in SwiftUI. This leads to resuing views, which sometimes can be challenging.

It may soon be deprecated. As of now, SwiftUI is more supported for newer versions of iOS.

2.2.2 SwiftUI

SwiftUI is a new framework that Apple released in 2019 with iOS 13.

2.2.2.1 Advantages

Live previews. Live changes can be previewed while developing the UI screen. This speeds up the development process, as there is no need to restart the app after each change.

Less code needed. There is less code needed to write the SwiftUI app against of UIKit app.

Better supported. Starting from iOS 13, SwiftUI is better supported for the newer versions of iOS.

2.2.2.2 Disadvantages

Only starting from iOS 13. SwiftUI support only gadgets with iOS 13 or greater, which leaves phones with older iOS without support.

Small ecosystem. It can be tough to find a solution to a problem because SwiftUI is a new framework with some issues no one has faced before. Furthermore, best practices are not yet formed.

Some features are not available yet. Some features are not yet available in SwiftUI. So, probably, there will be a need to implement some modules written on UIKit. For example, WebView is not implemented in SwiftUI.

2.2.3 Why I've choosen SwiftUI

SwiftUI is a new technology that Apple is forcing. I think that shortly most of the iOS apps will be built on SwiftUI. Furthermore, SwiftIU is great for building demo applications because it allows to create UI screens using live

preview fastly. Also, some experience with the Reactive framework React Native will let me quickly get used to SwiftUI.

2.3 iOS, iPadOS programming difference

The task of my bachelor's thesis is to create an application for iPadOS. One thing I want to point out is that iOS and iPadOS applications are mostly being developed the same way.

Because I did not have any physical iPad available, I mostly tested the application on a physical iPhone and virtual iPhone and iPad, so the application is mentioned to be available for both platforms: iOS and iPadOS.

Live previews available in SwiftUI really helped to maintain responsive design as all the changes in code are being updated on the screen within seconds and a few screen previews with different sizes can be watched at the same time. I added iPad 4th Generation and iPhone 12 for live previews to maintain responsive design in my application. For more information, please, take a look at section 5.2.2.

Chapter 3

Analysis

In this chapter, I will analyze existing solutions and define the requirements and use cases of the application that will be implemented.

3.1 Existing solutions

With the help of the research, I found only one application for iOS that is built to communicate with Interactive Brokers Services. It is an official mobile app provided by IBKR. Please take a look at appendix figures B.1 and B.2.

3.2 Requirements

"Requirements Engineering is a requirement process carried out with an engineering level of rigor. Requirements Engineering includes all aspects of requirements gathering and maintenance." [12]

Requirements in software engineering are usually divided into two kinds functional and non-functional.

Some of the functional and non-functional requirements are based on the assignment, and some of them are based on the analysis of existing software - IBKR for iOS.

3.2.1 Functional requirements

Functional requirements are technical requirements, including features, capabilities, and security. Those requirements mainly describe the real behavior of software applications. [13]

3.2.1.1 Authentication

User can log in to the application to a production and paper(demo) account.

3.2.1.2 User information

User can look at information about account on the separate tab.

3.2.1.3 Real-time market data of instruments

User can access real-time market data of the financial instrument, which will be refreshed within 1 second.

3.2.1.4 Historical data of instruments

User can access historical data of instruments, such as graphs, 52 weeks low and high, average volume, etc.

3.2.1.5 Making orders

User can buy and sell financial instruments.

3.2.1.6 Viewing transaction

User can take a look at transactions made within a scope of 24 hours.

3.2.2 Nonfunctional requirements

Non-functional requirements are all other requirements that are not related to behavior. For example, does the project need documentation or what programming language software should be written on.

3.2.2.1 Operating system

The application supports operating systems iPadOS and iOS starting from iOS 13.

3.2.2.2 Programming language

The application is written on the newest Swift programming language version - Swift 5.

3.3 Use cases

In this demo application, there is only one role - user. Please take a look at the figure C.1 in appendix for use case diagram.

Chapter 4

Architecture and design

In this chapter, I will describe the app's architecture, compare the two most popular architectures for iOS mobile development, and show and describe the presentation layer of the application.

4.1 Architecture

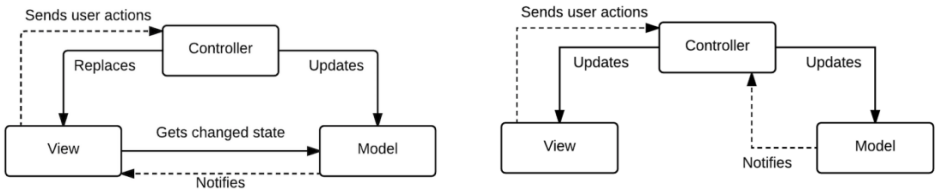
The exemplary architecture can significantly simplify the process of developing an application, testing it, and its further maintenance.

As of today, there are many different architectures that can be choosen for the project. Usually, choosen architecture depends on the programming language, operational system, and client requirements.

For iOS development in SwiftUI are primarily used 2 kinds of architecture: MVC and MVVM.

4.1.1 Model-View-Controller Architecture (MVC)

MVC was introduced in 1979 by Trygve Reenskaug during his work at Xerox. The main idea of that pattern is to separate applications into three different layers: Model layer - to store and manage data, View layer - to represent data to the user, and Controller layer - to handle user interactions. With the help of this separation, MVC allows developers to write maintainable and structured applications. The main drawback of this pattern was that each of the three layers is linked to the other two layers, reducing the reusability of objects. Because of that, Apple decided to change the traditional pattern just a little, as shown in Figure 4.1. This decision solved the problem of reusability.[2]



(a) The traditional MVC design pattern (b) Apple's MVC design pattern

Figure 4.1: MVC design patterns: traditional and Apple's [2]

Nevertheless, sometimes controllers become more and more massive, which becomes hard to test and maintain. This is called a Massive View Controller Problem. This problem states that if the controller is longer than 150 lines of code, there is a problem.

4.1.2 Model-View-ViewModel Architecture (MVVM)

This design pattern was introduced in the 1980s to solve the limitation problems of the MVC pattern. MVVM pattern reduces the complexity of the code and maximizes testability and reusability. MVVM introduced a new layer ViewModel, which is needed to prepare data for the View layer.

Basically, this pattern states that if some data from the Model needs to be presented on the View, View will ask ViewModel to prepare data for it, and ViewModel will access data from the Model layer. MVVM stands for Model-View-ViewModel.

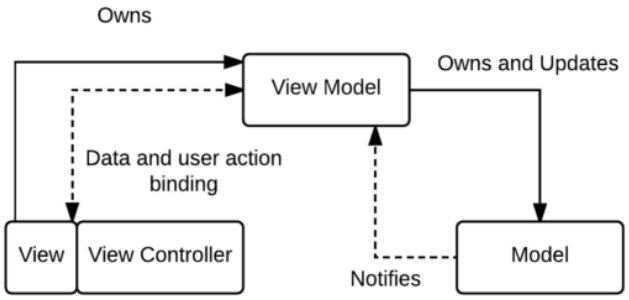


Figure 4.2: MVVM Model [2]

4.2 Design

In this section, I divide project in to separate layers according to MVVM pattern, as well as show and describe designed screens of the application.

4.2.1 Views

The view is the layer where all representative pages are stored. For each view, there is a ViewModel, which helps to communicate with the model layer.

- **Home View** - Initial screen, where the user gets to when he or she opens an app. Consists of overall data: account performance graph, top three portfolio positions, and daily gainers tickets are coming from the screener. For more information, look at subsection 4.3.2.
- **Account View** - Screen, which contains the primary information about the user's account. For more information, look at subsection 4.3.3.
- **Portfolio View** - Screen containing the user's portfolio, P&L ratio, cash balances, and cash power. For more information, look at the subsection 4.3.4.
- **Ticket View** - Screen containing ticket information, like history and market data. For more information, look at the subsection 4.3.5.
- **Search View** - Screen, where users can search for tickets. For more information, look at the subsection 4.3.6.
- **Trades View** - Screen, containing all user's orders. For more information, look at the subsection 4.3.7.
- **Transaction View** - Screen, where the user can buy or sell a financial instrument. For more information, look at the subsection 4.3.8.

Each view is stored under the Views folder, and almost for each view, there is another folder because most of the views are separated on components, as it is easier to maintain, and those components could be reused in the future as well. For example Portfolio View is divided in components:

- Portfolio View
- Cash Balances component
- Portfolio Header component
- Cash Balances Row component
- Portfolio List Item component

4.2.2 Model layer

In this section, I describe a Model layer of the app. First of all model layer is divided into a few pieces. The first one is data models, where all data models are stored.

- **Account Model** - Account models are used to encode POST bodies and decode responses in AccountApiService.
- **Home Model** - Models used are to encode POST bodies and decode responses in HomeApiService.
- **Portfolio Model** - Portfolio models are used to encode POST bodies and decode responses in PortfolioApiService.
- **Ticket Model** - Ticket models are used to encode POST bodies and decode responses in TicketApiService.
- **Transaction Model** - Transaction models are used to encode POST bodies and decode responses in TransactionApiService.
- **Trades Model** - Transaction models are used to encode POST bodies and decode responses in TradesApiService.

The second one is repositories. Repositories are used to process data coming from API.

- **Account Repository** - repository, that processes and returns data coming for AccountViewModel.
- **Home Repository** - repository, that processes and returns data coming for HomeViewModel.
- **Portfolio Repository** - repository that processes and returns data coming for PortfolioViewModel.
- **Ticket Repository** - repository that processes and returns data coming for TicketViewModel.
- **Search Repository** - repository that processes and returns data coming for SearchViewModel.
- **Transaction Repository** - repository that processes and returns data coming for TransactionViewModel.
- **Trades Repository** - repository that processes and returns data coming for TradesViewModel.

The third one is services. Services directly make an API call to different endpoints.

- **Account API Service** - Service, which calls account information endpoints and pass data down to repositories.
- **Home API Service** - Service, which calls endpoints to get data, which is being used on HomeView.
- **Portfolio API Service** - Service, which calls endpoints to get portfolio data of user.
- **Ticket API Service** - Service, which calls endpoints to get market and history data of ticket.
- **Search API Service** - Service, which calls endpoint for searching a ticket.
- **Transaction API Service** - Service, which calls endpoint for making an order (buying and selling financial instruments).
- **Trades API Service** - Service, which call endpoint to get all user's orders.
- **Web Socket Service** - Service, which connects to socket and recieves real-time socket information.

All of the services, besides WebSocketService, have their folder. There are three files in those folders - protocol, production API service, mock API service, which implement services protocol, and simulated models, where all testable mocked responses are located.

So-called protocols describe the API services. Protocols in SwiftUI are like interfaces in other programming languages. They describe methods without implementing them. Protocols help to write more reusable and testable code.

For mocked models, I decided to use real responses, which are being sent from the server, and store them in additional JSON files, which are being situated under the folder MockJSON. In mocked models, JSON responses are being decoded.

For example, the structure of the folder AccountApiService includes files:

- **AccountApiServiceProtocol.swift**
- **AccountApiService.swift**
- **MockAccountApiService.swift**
- **MockAccountModels.swift**

4.2.3 ViewModel layer

In this section, I describe the ViewModel layer of the application. ViewModels are extremely important in architecture. ViewModel is like an intermediary layer between the View and Model layers. Through ViewModels View layer interacts with the Model layer to get and send data. Almost every view, has it's own ViewModel.

- **Account ViewModel** - Intermediary layer between Model layer and AccountView.
- **Home ViewModel** - Intermediary layer between Model layer and HomeView.
- **Portfolio ViewModel** - Intermediary layer between Model layer and PortfolioView.
- **Ticket ViewModel** - Intermediary layer between Model layer and TicketView.
- **Search ViewModel** - Intermediary layer between Model layer and SearchView.
- **Transaction ViewModel** - Intermediary layer between Model layer and TransactionView.
- **Trades ViewModel** - Intermediary layer between Model layer and TradesView.
- **Environment ViewModel** - ViewModel, which is being used across few Views.

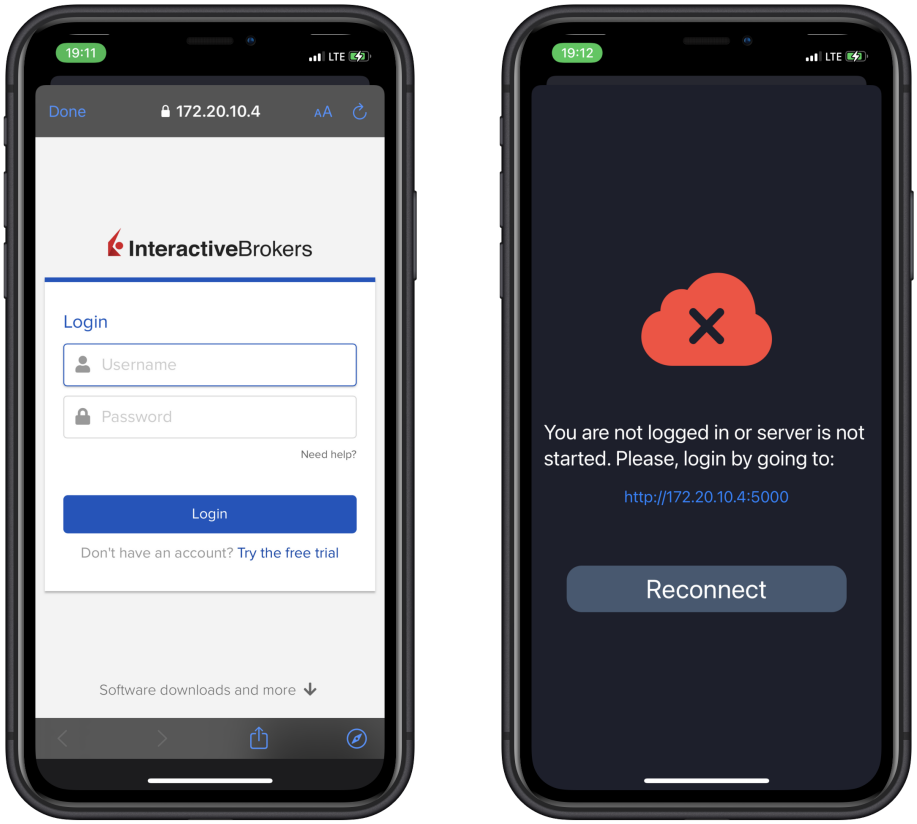
4.3 User interface

In this section, I will show user interfaces and describe their expected behavior.

4.3.1 Login screen

Since login is being made through the gateway, which needs to be started locally, I made a few variants of the login screen. The first one redirects the user to SFSafariView to the Interactive Brokers login page if the gateway is started through a secured HTTPS protocol. Look at figure 4.3 (a).

The second variant shows an alert, indicating that the user is not logged in and providing steps to authenticate. This screen will be displayed if a user is not logged in and the gateway is started through HTTP protocol. Look at figure 4.3 (b).



(a) Login SFSafariView through HTTPS (b) Login alert through HTTP

Figure 4.3: Two variants of login screens

4.3.1.1 Why are there two different variants of login?

Apple monitors the security of developed applications. It is tough to open unsecured web pages via SFSafariView or WebView. Of course, it can be opened, but Safari does not allow cookies for unsecured web pages. If an unsecured web page is being opened, Safari alerts that cookies must be enabled.

Unfortunately, Client Portal API uses cookies to authenticate. When an unsecured web page is being opened through SFSafariView or WebView, the fields for entering username and password are hidden, and the user can't log in. Look at figure 4.4.

The chrome browser works differently, and the user can see those fields, but we cannot guarantee that the user has downloaded the Chrome browser from App Store before.

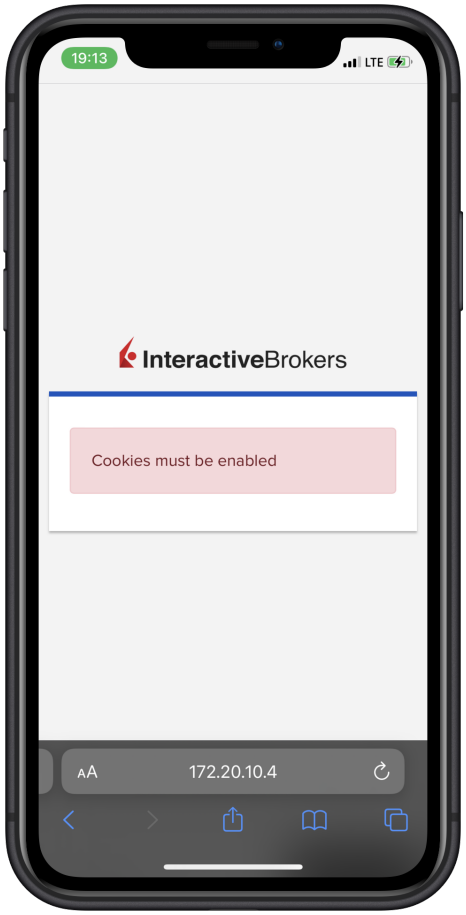


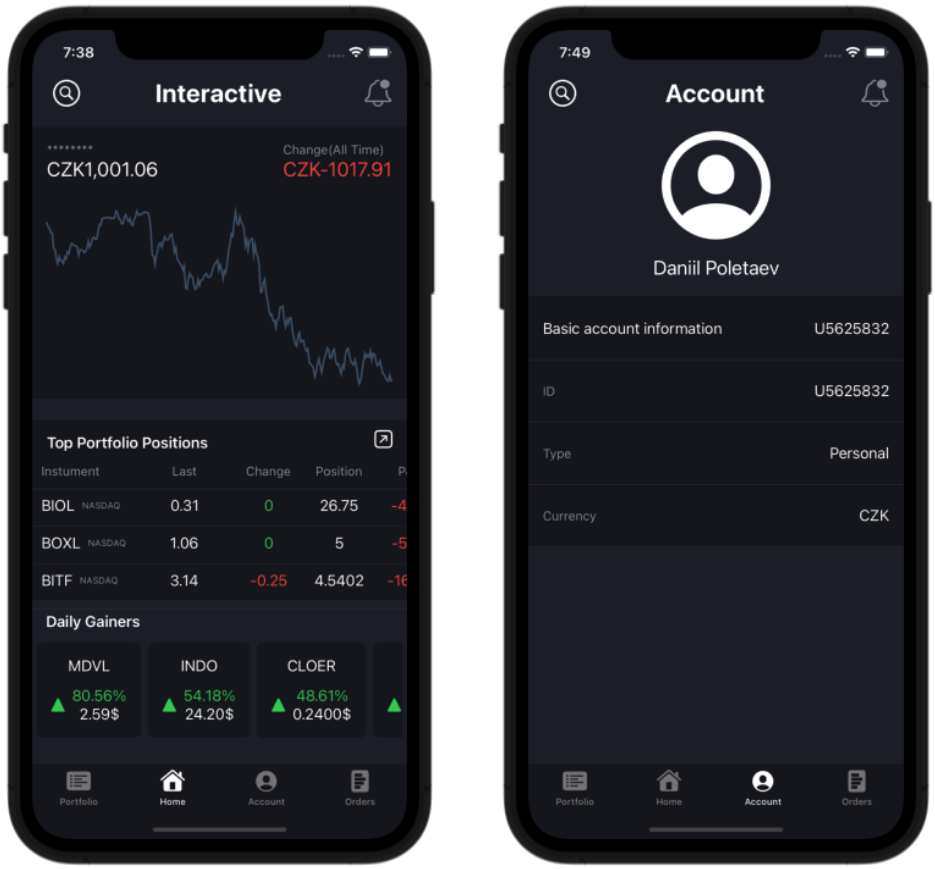
Figure 4.4: Safari cookies must be enabled

4.3.2 Home screen

This is an overall screen with basic account information. On that screen, users can access account performance, take a look at the graph representing their investment and get the top three portfolio positions and five daily gainers at the bottom. Please take a look at figure 4.5 (a).

4.3.3 Account screen

On the account screen, users can overview basic account information, such as account type, name, id, and base currency. Please take a look at figure 4.5 (b).



(a) Home Screen (b) Account Screen

Figure 4.5: Home and Account screens

4.3.4 Portfolio screen

On the portfolio screen, users can look at all of their currently opened positions, cash balances and primary account information, and daily P&L ratio. Please take a look at figure 4.6 (a).

4.3.5 Ticker screen

On the ticker screen, users get up-to-date information about the ticker they selected before on the ticker screen. Information on this screen is coming from WebSockets. Thus market data is being refreshed within seconds. Please take a look at figure 4.6 (b).

(a) Portfolio Screen (b) Ticker Screen

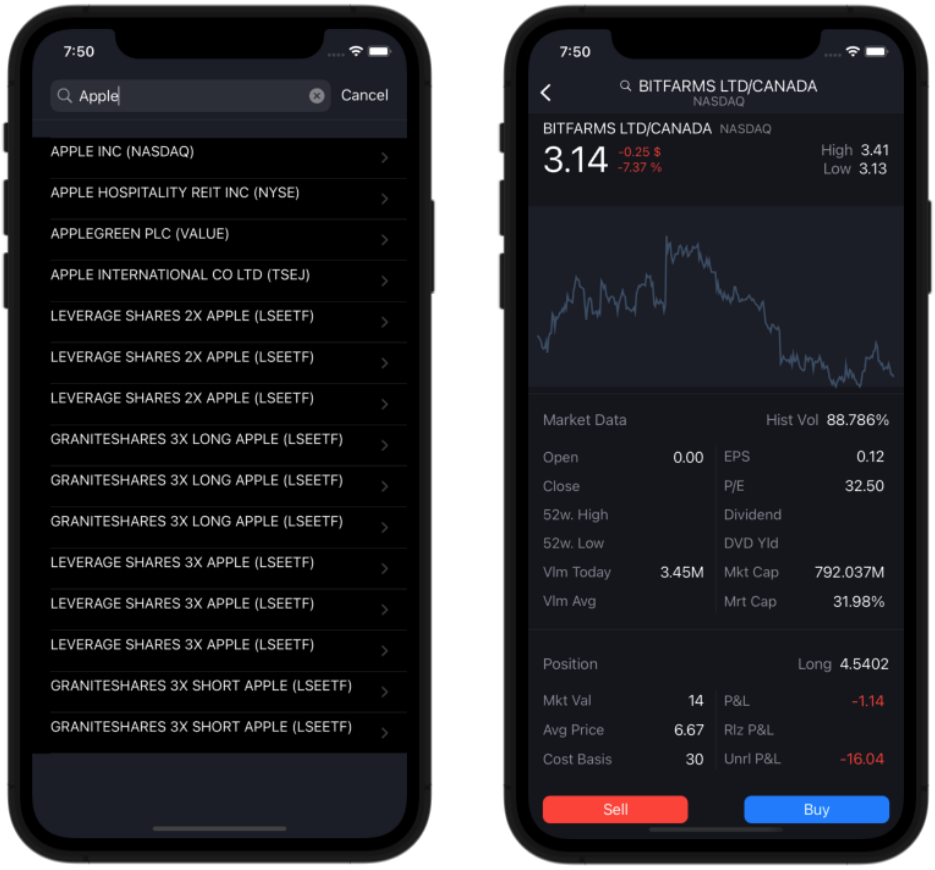
Figure 4.6: Portfolio and Ticker screens

4.3.6 Search screen

On the search screen, users can type and find stocks they are interested in. By clicking on found stock, the ticker screen will be opened. Please take a look at figure 4.7 (a).

4.3.7 Trades screen

Users can look at orders made during the day and check orders' statuses on the trades screen. Please take a look at figure 4.7 (b).



(a) Search Screen (b) Trades Screen

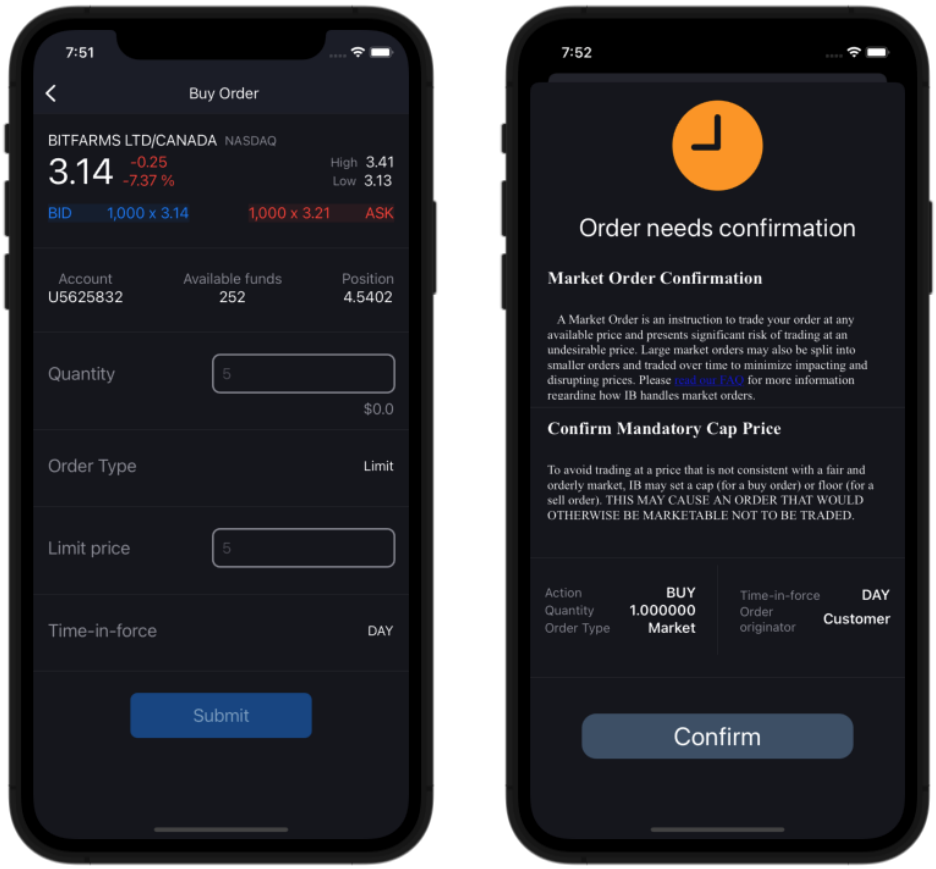
Figure 4.7: Search and Trades screens

4.3.8 Transaction screen

By clicking buy/sell at the ticker screen, users are redirected to the transaction screen. On the transaction screen, they can buy or sell financial instruments. Please take a look at figure 4.8 (a).

4.3.8.1 Confirmation screen

By clicking submit button on the transaction screen, the confirmation screen opens. On that screen, users need to confirm the transaction they made before. If the transaction is approved, they are redirected to the orders screen. Please take a look at figure 4.8 (b).



(a) Transaction Screen (b) Confirmation Screen

Figure 4.8: Transaction and Confirmation screens

Chapter 5

Implementation

This chapter will describe the technical implementation of the iOS demo mobile application. I will point out the most critical aspects of the development process, such as authorization, getting and processing data from the server, dealing with security nuances, and describing open source packages used for the project.

5.1 Setting up Client Portal API gateway

Before we can start using Interactive Brokers API, we need to set up a gateway. Here are the steps to set it up:

1. Download and unzip the gateway from the official Intractive Brokers web site

2. Take a look at gateway configuration file. You can adjust it by your needs. I will only describe basic configurations, which I used in the project.

    a) For HTTP protocol, change variable listenSsl to false. When listenSsl is true, it will start the gateway through HTTPS protocol.

    b) You can change the certificate. If you think about deploying a gateway to the server, it is a good idea. Do not forget to change the variable "sslCert" (certificate file name, which is stored under gateway /root folder) and "sslPwd" (certificate password used for the creation of the certificate).

    c) Do not forget to add allowed IP addresses under allow otherwise you will get the error: "Access denied".

3. In terminal navigate to the directory of unzipped gateway and run command:

a) To start gateway on **windows** run: ”bin\run.bat root\conf.yaml”.

b) To start gateway on **Mac/Unix** run: ”bin/run.sh root/conf.yaml”.

4. Authenticate with username and password to start getting data from endpoints. [9]

```yaml
ip2loc: "US"
proxyRemoteSsl: true
proxyRemoteHost: "https://api.ibkr.com"
listenPort: 5000
listenSsl: true
svcEnvironment: "v1"
sslCert: "vertx.jks"
sslPwd: "mywebapi"
authDelay: 3000
portalBaseURL: ""
serverOptions:
    blockedThreadCheckInterval: 1000000
    eventLoopPoolSize: 20
    workerPoolSize: 20
    maxWorkerExecuteTime: 100
    internalBlockingPoolSize: 20
cors:
    origin.allowed: "*"
    allowCredentials: false
webApps:
    - name: "demo"
      index: "index.html"
ips:
  allow:
    - 192.*
    - 131.216.*
    - 127.0.0.1
  deny:
    - 212.90.324.10
```

Listing 1: Gateway configuration

5.2 Configuration of project

In this section, I will describe practices I used for configuring projects to maximize the productivity of the developing process, such as mocking responses,

dependency injections and live previews of UI views, and telling a few nuances related to security settings.

5.2.1 Dependency injection & mocking responses

Fowler firstly introduced the term Dependency Injection in 2004. After that, DI containers spread all over popular frameworks, including Java Spring framework and Google Guice. [14]

In other words, dependency injection is a pattern we use while developing an application. It reduces coupling between components as well as saves us from repeating boilerplate factory creation code over and over again. [15]

Dependency injection moves the initialization of dependent objects away from the component. Thus those dependent objects are being initialized and passed to the component. This technique also helps in testing because we can pass mocked objects to the component. For example, we can pass mock API services and test the behavior of applications and units on those services instead of using productions services.

Based on the PortfolioView.swift, I show how I use DI in the application. Take a look at the listing below.

```
1  struct PortfolioView: View {
2      @StateObject var portfolioViewModel: PortfolioViewModel
3      @EnvironmentObject var environmentModel:
       ↪  EnvironmentViewModel
4
5      // Dependency injection through initializer
6      init(portfolioViewModel: PortfolioViewModel?) {
7          _portfolioViewModel = StateObject(wrappedValue:
           ↪  portfolioViewModel ?? PortfolioViewModel(repository:
           ↪  nil))
8      }
9      // ...
10 }
```

Listing 2: PortfolioView.swift

I moved the initialization of needed variables to the initializer in which I can pass either initialized variables or nil. If nil is passed, the initializer creates new default instances of required variables.

Mainly, I pass non-null instances to components only for testing and live previews.

5.2.2 Live previews

In subsection 2.2.2.1, I pointed out that one of the most significant advantages of SwiftUI is live previews. It means that Xcode can show up-to-date previews while developers create custom Views. Each time code changes, Xcode automatically reloads the preview and shows new changes up on the screen. There is no need to build an application and run it.

Furthermore, it allows customizing those previews by choosing a background and color scheme and permanently changing the device. In addition to this, multiple devices can be added for the preview simultaneously.

My assignment stated that I had to implement an application for iPad OS. Still, I had only an iOS mobile phone, so I decided to add simultaneously two devices - iPhone 12 and iPad Air 4th generation and watch every change on both of them simultaneously. Please look at figure 5.1 below to see what it looks like in a development environment, and take a look at the listing 3 to view the implementation of the preview in code.

Figure 5.1: Live preview in Xcode

```
struct AccountView_Previews: PreviewProvider {
    static var previews: some View {
        // Mock environment model
        let environmentModel =
        ↪  MockedAccountModels.mockedEvnironmentModel
        Group {
            AccountView().environmentObject(environmentModel)
                .onAppear(perform: {
                    environmentModel.fetchData()
                })
            .background(CustomColor.lightBg)
            .environment(\.colorScheme, .dark)
            // Set preview device
            .previewDevice(PreviewDevice(rawValue: "iPhone 12"))

            AccountView().environmentObject(environmentModel)
                .onAppear(perform: {
                    environmentModel.fetchData()
                })
            .background(CustomColor.lightBg)
            .environment(\.colorScheme, .dark)
            // Set preview device
            .previewDevice(PreviewDevice(rawValue: "iPad Air
            ↪  (4th generation)"))
        }
    }
}
```

Listing 3: Live Preview of Account View

5.2.3 Security limitations

Apple looks after the security of the application written for iOS and iPadOS. Default apple security does not allow fetching endpoints through HTTP or HTTPS protocols with an unverified or self-signed certificate.

As of nowadays, unfortunately, the Client Portal API gateway comes with an expired certificate. As it is stated on the official pages FAQ: *"Since the gateway is running on your premises the certificate needs to be created/self-signed by you, or officially signed by a 3rd party. The gateway is similar to another web server such as Tomcat which doesn't provide a certificate along with the release."* [16]

It does not make sense, as this gateway is mostly running on localhost, making it hard to get a signed certificate. Because of that, I implemented

CustomURLSession delegate, which inherits swift protocol URLSessionDelegate.

"URLSessionDelegate is a protocol that defines methods that URL session instances call on their delegates to handle session-level events, like session life cycle changes." [17]

Basically, it helps to override the default behavior of the session life cycle. I used it to change the behavior of interacting with HTTP or self-signed HTTPS servers, allowing such connection without throwing an error.

I created custom URLSessionDelegate class, which allows fetching from unsecure protocols. Take a look at the listing below.

```
1  // Custom URLSessionDelegate for handling
2  // unsecure API calls
3  class CustomUrlSessionDelegate: NSObject, URLSessionDelegate {
4      func urlSession(_ session: URLSession, didReceive challenge:
       ↪  URLAuthenticationChallenge, completionHandler: @escaping
       ↪  (URLSession.AuthChallengeDisposition, URLCredential?) ->
       ↪  Void) {
5          completionHandler(.useCredential, URLCredential(trust:
           ↪  challenge.protectionSpace.serverTrust!))
6      }
7  }
```

Listing 4: Custom URLSessionDelegate

After that, I created the class DataManager. All APIServices inherit this class and make API calls through URLSession instance, which is initialized in DataManager. Please take a look at the listing below.

```
1  class DataManager: NSObject {
2      let session: URLSession = URLSession(configuration:
       ↪  URLSessionConfiguration.default, delegate:
       ↪  CustomUrlSessionDelegate(), delegateQueue:
       ↪  OperationQueue.main)
3      //  ...
4  }
```

Listing 5: DataManager.swift

Disclaimer: Connecting to an unsecured server is only allowed while developing an application. Apple won't let publish an app to the App Store.

5.3 Authorization

In this section, I am going to describe the authorization process. The authorization process is a bit complicated because of Apple's security limitations. For more information about security limitations, please take a look at subsection 5.2.3.

In order to start getting responses from endpoints, the user needs to authenticate to Interactive Brokers through the started gateway. For that user needs to go to the web page. The web page address depends on the config file. For more information about config files, take a look at section 5.1. **The default address of local gateway is https://localhost:5000**.

By going to that web page in the browser, the user can log in to his or her account and start getting responses from the server. The web page is provided by the Interactive Brokers gateway. The content of the login web page is shown in the figure below.

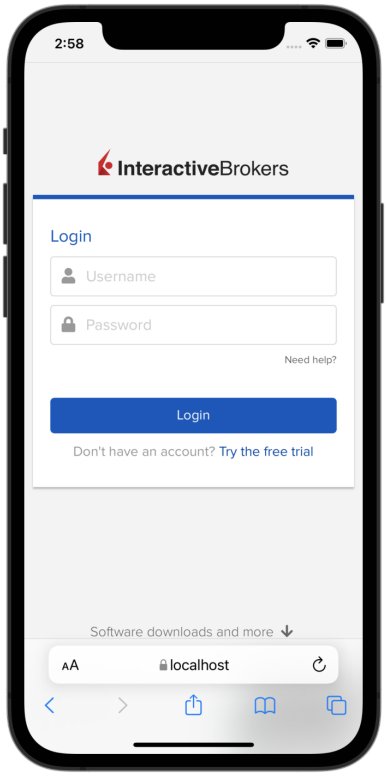


Figure 5.2: Gateway authorization page

5.3.1 Is there OAuth protocol?

On the comparison table 1.1, which was taken from the official Interactive Brokers' web page, it was said that Client Portal API supports OAuth.

OAuth is the protocol that was created by a small community of developers. They wanted to solve the common problem of enabling access to protected resources between various websites. The first version of that protocol was released in October of 2007. [18]

"OAuth introduces a third role to the traditional client-server authentication model: the resource owner." [18] The main idea of that protocol is that clients can share server resources with another third-part service without sharing their actual credentials. [18]

When I started to write implementation, I thought it would be possible to authorize Interactive Brokers with the help of the OAuth protocol. Still, when I contacted IB service, they said that OAuth works only for companies, such as financial advisors.

5.3.2 Authorization sheet

There are two different variants of login, which I already described in subsection 4.3.1. In this subsection, I will describe the technical nuances of the authorization process through the HTTPS protocol.

First of all, in ContentView, I check if the user is already authorized. It is done by calling the endpoint in the onAppear method. *"onAppear method adds an action to perform when this view appears."* [19] Take a look at the listing below.

```
1  .onAppear(perform: {
2              self.environmentModel.getIServerAccount{ (data,
               ↪  error) in
3                  if (error != nil) {
4                      self.showingAuthorizationSheet = true
5                  }
6              }
7          })
```

Listing 6: ContentView onAppear method

If getIServerAccount returns an error, showingAuthorizationSheet is being changed to true, and a sheet is being popped up above the application's content.

That sheet conditionally renders content based on whether the gateway is started through HTTP or HTTPS protocol.

If the gateway is started through HTTPS protocol, the Safari browser is opened inside that sheet. Only text appears if the gateway is being started through HTTP protocol, asking the user to authorize manually. Please, look at the listing below, which shows an implementation of the authorization sheet.

```swift
struct AuthorizationSheet: View {
    @Environment(\.dismiss) var dismiss
    @EnvironmentObject var environmentModel:
    ↪ EnvironmentViewModel
    @State var errorAgain = false

    var body: some View {
        // Conditional rendering based on protocol
        if DataManager().API_URL.starts(with: "https") {
            SFSafariViewWrapper(url: URL(string:
            ↪ "https://\(APIConstants.COMMON_BASE_URL)")!)
        } else {
            VStack {
                // ...

                Text("You are not logged in or server is not
                ↪ started. Please, login by going to:")
                    .font(.title2)
                    .padding(10)
                    .accessibility(identifier:
                    ↪ "loginErrorStaticText")
                // ...
            }
            // ...
        }
    }
}
```

Listing 7: Authorization sheet

Nowadays, there is no WebView or SafariView implemented in SwiftUI. I had to write a custom SafariViewWrapper component, which uses the UIKit SafariServices package and transforms it for use in SwiftUI. To use SFSafariViewWrapper, only the URL must be passed to an instance of this struct. Please take a look at the listing below.

Unfortunately, SFSafariViewController only opens the web page in the application and does not allow tracking. Because of this, whenever users log in to their accounts, they will need to manually close the authorization sheet, pressing ”Done” at the left-top corner. Whenever the ”Done” button is pressed, the application will try to get a response from getIServerAccount after 3 seconds of closing the sheet. If there is an error, the authorization sheet will be displayed again for the user.

```
struct SFSafariViewWrapper: UIViewControllerRepresentable {
    let url: URL
    func makeUIViewController(context:
    ↪ UIViewControllerRepresentableContext<Self>) ->
    ↪ SFSafariViewController {
        return SFSafariViewController(url: url)
    }
    func updateUIViewController(_ uiViewController:
    ↪ SFSafariViewController, context:
    ↪ UIViewControllerRepresentableContext<SFSafariViewWrapper>)
    ↪ {
        return
    }
}
```

Listing 8: SFSafariViewWrapper

5.4 Getting data

This section will describe the process of getting data from Interactive Brokers using REST API and WebSockets.

5.4.1 Getting data from REST API

Most of the information is being fetched in the application via REST API. It is the basic information, where the continuous update is not that important.

I have used REST API to fetch account information, get daily gainers, place orders, search for contracts and get portfolio information.

As was described in section security limitations, I created a custom URLSessionDelegate to allow requests from HTTP and unverified or self-signed HTTPS protocol. All API services inherit the class DataManager, where the session is initialized with a custom delegate. Inheriting from DataManager allows me to use URLSession instance in all API services without further initialization of this instance in all of the services.

For example, please take a look at the function fetchAccount listed below. This function gets account information, decodes it, and sends it down to the repository for processing. Before decoding the response, I always check if the it was successful.

```swift
func fetchAccount(completion: @escaping ([Account]) -> ()) {
        guard let url = URL(string:
        ↪  self.API_URL.appending("/portfolio/accounts")) else
        ↪  {
            // returned if URL is incorrect
            return
        }
        let task = self.session.dataTask(with: url) { data, _,
        ↪  error in
            guard let data = data, error == nil else {
                return
            }
            do {
                // Decodes JSON response to swuft object
                let accounts = try
                ↪  JSONDecoder().decode([Account].self, from:
                ↪  data)
                DispatchQueue.main.async {
                    completion(accounts)
                }
            } catch {
                // prints error if there was
                // decoding error
                print(error)
            }
        }
        task.resume()
    }
```

Listing 9: fetchAccount function

For decoding, I created data transfer objects used for serialization and deserialization of JSON. Basically, they describe expected responses with their properties and the types of those properties.

In swift, data transfer objects are mostly being created with the same property names as coming from the responses. Swift automatically knows how to decode such data.

For responses, where property names start with unusual characters or numbers, I had to additionally add CodingKeys for decoding.

For example, please, take a look at Account DTO in the listing below.

```
struct Account: Codable, Identifiable {
    let id: String
    let accountId: String
    let accountVan: String
    let accountTitle: String
    let displayName: String
    let accountAlias: String?
    let accountStatus: Decimal
    let currency: String
    let type: String
    let tradingType: String
    let ibEntity: String
    let faclient: Bool
    let clearingStatus: String
    let covestor: Bool
    let parent: AccountParent
    let desc: String
}
```

Listing 10: Account Data Transfer Object

5.4.2 Getting data from socket

Even though most of the data is being fetched through REST API, getting data from a socket is essential for an application with updated information every second or even faster. In application, sockets are used to get up-to-date information about stocks and other contracts. This information changes fast and should be reliable, so users can evaluate the price, volume, or further contract information and decide whether they would buy/sell this financial instrument.

Before we can start getting messages from the server via a socket connection, there is a need to authorize. To authorize, the session value needs to be sent to the server. This session value can be obtained from the endpoint /tickle.

To work with sockets, I wrote the class WebSocketService. The listing below shows the main methods used to realize socket connection between client and server. Function ”sendRepeatedly” sends a message with a 1-second delay, and function ”listenForMessages” waits for messages from the server.

```
class WebSocketService: AsyncSequence {
    private var stream: AsyncThrowingStream<Element, Error>?
    private var continuation: AsyncThrowingStream<Element,
    ↪  Error>.Continuation?
    private let socket: URLSessionWebSocketTask
    let session: URLSession = URLSession(configuration:
    ↪  URLSessionConfiguration.default, delegate:
    ↪  CustomUrlSessionDelegate(), delegateQueue:
    ↪  OperationQueue.main)
    // Initialization
    init(url: String, session: URLSession =
    ↪  URLSession(configuration:
    ↪  URLSessionConfiguration.default, delegate:
    ↪  CustomUrlSessionDelegate(), delegateQueue:
    ↪  OperationQueue.main)) {
        socket = session.webSocketTask(with: URL(string: url)!)
        stream = AsyncThrowingStream { continuation in
            self.continuation = continuation
            self.continuation?.onTermination = { @Sendable
            ↪  [socket] _ in
                socket.cancel()
            }
        }
    }
    // Waiting for messages
    private func listenForMessages() {
        socket.receive { [unowned self] result in
            switch result {
            case .success(let message):
                continuation?.yield(message)
                listenForMessages()
            case .failure(let error):
                continuation?.finish(throwing: error)
            }
        }
    }
    // Sending messages with a delay
    func sendRepeatedly(message: String) {
        DispatchQueue.global().asyncAfter(deadline: .now() + 1)
        ↪  { [self] in
            send(message: message)
        }
    }
}
```

Listing 11: WebSocketService

Using the WebSocket class the socket connection can be easily established and maintained in the ViewModels. Please take a look at the appendix for usage of WebSocket class. TickeViewModel establishes a connection with the socket.

First of all function loadTickerInfo is being called, which authorizes the user with session obtained from /tickle endpoint. After authorization, request containing contract id (conid) and required fields needs to be sent to start getting ticker information. There is a huge list of all available fields of ticker information. You can take a look at them at the swagger under the /iserver/-marketdata/snapshot endpoint.

Function retrieveMessages waits for messages from the server. Once the message is being recieved this method decodes the response and refreshes information about the ticket, which is then being presented on the View layer.

Whenever the user leaves the screen, the connection is closed.

Usually, the ticker information response returned from the socket connection does not have all the required fields, which were sent in the request. This can lead to overwriting some properties of the ticker information object to null. To avoid this problem, I created an extension that helps to combine ticker information. Basically, this extension gets decoded response and appends only properties with non-nullable values to the existing instance of ticker information. Please take a look at the listing below.

```
1 extension TickerInfo {
2     func combine(newTicket: TickerInfo) -> TickerInfo {
3         let high = newTicket.high ?? self.high
4         let low = newTicket.low ?? self.low
5         // All other properties the same way
6         // Return TickerInfo instance with appended properties
7     }
8 }
```

Listing 12: TickerInfo combination extension

5.5 Processing data

Once the API service fetches and decodes the response, data is passed to the repository for data processing.

Some of the repository methods only return decoded responses. For example, please take a look at the listing below. Method searchForNameSymbol is used to find contracts by name or symbol. It returns a list of found contracts with such a name. There was no need to process and reformat this data. Because of that, this function passes down to the ViewModel decoded response without further processing.

```
final class SearchRepository: SearchRepositoryProtocol {
    // ...
    func searchForNameSymbol(value: String, completion:
    ↪ @escaping ([SearchTicket]) -> ()) {
        self.apiService.searchForNameSymbol(value: value) {
        ↪ tickers in
            completion(tickers)
        }
    }
}
```

Listing 13: Search for name or symbol repository function

Even though some repositories' functions return decoded responses, most of the repositories' methods process data to get the correct format. For example, please take a look at the listing below. Function fetchTopPositions gets data about accounts' opened positions, filters it in ascending order, and returns the top three positions to present them on HomeView.

```
func fetchTopPositions(completion: @escaping ([Position]) ->
↪ Void) {
        self.accountApiService.fetchAccount { accounts in
            self.portfolioApiService.fetchPositions(accountID:
            ↪ accounts[0].accountId) { positions in

                let filtered = positions.sorted {
                    $0.position ?? 0 > $1.position ?? 0
                }
                var slicedPositions: [Position]
                if (filtered.count > 3) {
                    slicedPositions = Array(filtered[0...2])
                } else {
                    slicedPositions = positions
                }
                if (positions.count == 0) {
                    completion(positions)
                }
                completion(slicedPositions)
            }
        }
    }
```

Listing 14: fetchAccountPerformance function

5.6 Making graphs

Graphs are essential for such an application. Graphs are used for presenting the account's performance and financial instrument price history. Charts are helpful for traders to make correct decisions related to the specific contract. Whether it is the best time to buy/sell a financial instrument.

I decided to use SwiftUI Stock Charts [20] to present a price graph in my application. I added this open-source package to my project, and I was able to use it right after that.

For example, of SwiftUI Stock Chart package usage, please, take a look at the listing below. TicketGraph is a component that displays a price graph in TicketView.

```
1  import SwiftUI
2  import StockCharts
3
4  struct TicketGraph: View {
5      @Binding var tickerInfo: TickerInfo?
6      @Binding var graphData: [Double]
7      @Binding var dates: [String] // format: yy-MM-dd
8
9      var body: some View {
10         VStack {
11             LineChartView(
12                 lineChartController:
13                     LineChartController(
14                         prices: self.graphData, dates:
                           ↪ self.dates, downtrendLineColor:
                           ↪ CustomColor.graphBlue, dragGesture:
                           ↪ true
15                     )
16             )
17         }
18         .padding(.horizontal, 10)
19         .frame(width: UIScreen.screenWidth, height: 200,
           ↪ alignment: .center)
20         .background(CustomColor.lightBg)
21     }
22 }
```

Listing 15: TicketGraph component

5.7 Loading animation

To make the application more user-friendly, I added loaders while data is being fetched from the server.

I added the ActivityIndicatorView open-source package [21], which has about ten predefined types of loaders. Please, look at the usage example below.

```
1  struct HomeView: View {
2
3      // Initialization
4
5      var body: some View {
6          ScrollView(showsIndicators: false) {
7              ZStack {
8                  VStack(alignment: .center) {
9                      // Content on top of which
10                     // loading is displayed
11                 }
12                 ActivityIndicatorView(isVisible:
                   ↪  \$homeViewModel.isLoading, type:
                   ↪  .scalingDots)
13                     .foregroundColor(Color.white)
14                     .frame(width: 80, height: 50, alignment:
                       ↪  .center)
15             }
16         }
17         // ...
18     }
19 }
```

Listing 16: ActivityIndicatorView example

Chapter 6

Testing

Nowadays, software testing is essential as a part of the software development process, and tests are widely accepted in the industry. However, the size and complexity of the software application steadily grow, and the development time decreases. Sometimes developers do not have time for writing tests, which leads to bugs and legacy code, which is hard to maintain in the future. [22]

In this chapter, I describe the testing of the application and the testing techniques used for testing the application.

All tests are divided into two different tests - Unit tests and UI tests.

6.0.1 Unit tests

Unit tests are pieces of code that test specific, small areas of software. Unit tests are the most critical tests in software testing. Unit tests make the life of developers more manageable. They help avoid legacy code, build well-designed software and help maintain it in the future. [23]

In my application, I mainly focused on Unit tests because they test the application's business logic. I started by testing small pieces of code, usually methods, of a higher application level and steadily continued to the lowest layer - ViewModels. With the help of Swift protocol and dependency injection, I created mocks of API services, which implement the protocol of API service. In the example below, you can take a look at AccountAPIServiceProtocol.swift file.

```swift
protocol AccountApiServiceProtocol {
    func fetchAccount(completion: @escaping ([Account]) -> Void)

    func getAccountPerformance(accountIds: [String], freq:
    ↪ String, completion: @escaping ((PerformanceResponse?,
    ↪ NetworkError?)) -> ())

    func getAccountAllocation(accountId: String, completion:
    ↪ @escaping (AllocationResponse) -> ())

    func getAccountSummary(accountId: String, completion:
    ↪ @escaping (AccountSummary) -> ())

    func getPnL(completion: @escaping (PnLModelResponseModel) ->
    ↪ ())

    func getIServerAccount(completion:
    ↪ @escaping((IServerResponse?, NetworkError?)) -> ())

    func tickle(completion: @escaping (TickleResponse) -> ())

    func getCurrentBalance(acctIds: [String], completion:
    ↪ @escaping (PASummaryResponse) -> Void)
}
```

Listing 17: AccountApiProtocol.swift

Mostly those mocked API services return a real-world response, decoded from JSON files, which are stored under the folder MockJSON. For example, below, you can see mock account models.

```swift
enum MockedAccountModels {
    static let account: [Account] = Bundle.main.decode(type:
    ↪ [Account].self, from: "AccountResponse.json")

    static let performanceResponse: PerformanceResponse =
    ↪ Bundle.main.decode(type: PerformanceResponse.self, from:
    ↪ "PerformanceResponse.json")
    // ...
}
```

Listing 18: MockAccountModels.swift

The purpose of the Mock API service is to return mock models and pass them down to repository layers. For more natural behavior, I made a delay of 0.5 seconds before the response is returned because it can take up to a few seconds to fetch those responses from the endpoint.

```
class MockAccountApiService: AccountApiServiceProtocol {
    func fetchAccount(completion: @escaping ([Account]) -> Void)
    ↪ {
        // returns accountTestData after 0.5 seconds
        DispatchQueue.main.asyncAfter(deadline: .now() + 0.5) {
            completion(self.accountTestData)
        }
    }
    // ...
}
```

Listing 19: MockAccountApiService.swift

This mock technique helped me to test services and other layers. The unit test of service is straightforward. It tests that the method returns simulated data and that the dependency injection is working.

```
class AccountApiService_Tests: XCTestCase {
    func test_AccountApiService_fetchAccount_shouldReturnItems()
    ↪ {
        let testAccount = MockedAccountModels.account
        // Initialization of instance of
        ↪ AccountApiServiceProtocol

        // Checks if method fetchAccount returns testAccount
        // object, which was passed through dependency
        // injection
        accountApiService.fetchAccount { accounts in
            XCTAssertEqual(accounts.count, 1)
            XCTAssertEqual(accounts[0].accountId,
            ↪ testAccount[0].accountId)
        }
    }
    // ...
}
```

Listing 20: AccountApiService Unit tests

Once I tested methods in API services, I continued with the testing of repositories. I recall that the main task of repositories is to process responses coming from the server and return ready-to-use objects with data, which can be accessed in ViewModels.

```
1  class AccountRepository_Tests: XCTestCase {
2      func
       ↪ test_AccountRepository_fetchAccount_shouldReturnAccountObject()
       ↪ {
3          let mockedAccount = MockedAccountModels.account
4
5          // Initialization of instance accountRepository
6
7          // checks if accountRepository returns account object,
8          // not in array
9          accountRepository.fetchAccount { account in
10             XCTAssertEqual(account.accountId,
               ↪ mockedAccount[0].accountId)
11             XCTAssertEqual(account.type, mockedAccount[0].type)
12             XCTAssertEqual(account.accountTitle,
               ↪ mockedAccount[0].accountTitle)
13             XCTAssertEqual(account.accountStatus,
               ↪ mockedAccount[0].accountStatus)
14         }
15     }
16     // ...
17 }
```

Listing 21: AccountRepository Unit tests

Finally, I wrote unit tests for ViewModels. In ViewModels, I mainly tested that the value coming from endpoints is assigned to states to present it on the View layer. Expectations in Swift unit testing are used to test asynchronous functions. Once expectations are fulfilled, tests are executed. Sink observes values received by the published. [24] I use dropFirst(), because this sink is being called while first initialization, which may cause tests to fail because expectations will be fulfilled too early.

```swift
class AccountViewModel_Tests: XCTestCase {
    // ...
        func
        ↪ test_AccountViewModel_fetchAccount_shouldSetAccount()
        ↪ {
        let mockedAccount = MockedAccountModels.account
        // Initialization of ViewModel instance

        let expectation = XCTestExpectation(description: "Should
        ↪ return response after 0.5~1 seconds")

        // fullfill expectation once accountViewModel.account is
        ↪ being changed
        accountViewModel.$account
            .dropFirst()
            .sink { acc in
                expectation.fulfill()
            }
            .store(in: &cancellables)

        accountViewModel.fetchAccount()

        // Expectation fullfilled start the test
        wait(for: [expectation], timeout: 2)
        XCTAssertTrue(accountViewModel.account != nil)
        XCTAssertEqual(accountViewModel.account?.id,
        ↪ mockedAccount[0].id)
        XCTAssertEqual(accountViewModel.account?.accountTitle,
        ↪ mockedAccount[0].accountTitle)
    }
    // ...
}
```

Listing 22: AccountViewModel Unit tests

6.0.2 UI tests

The second part of the testing software was UI tests. UI tests stand for User Interface test. Those tests validate whether an app has desired functionality and all features of the app work as expected. They test views and how those views interact with business logic. Compared to unit tests, which test only small pieces of code (units), UI tests are supposed to test the whole user flow.

I wrote automated UI tests for a few user flows. I decided to run automated UI tests on the mock data because there is no need for a gateway to be started. I added three different environment variables to make it work, which can be passed to an app before the launch.

1. **-UITest_unauthorized** - When set to true, the app will behave as an unauthorized.

2. **-UITest_isHTTP** - When set to true, the app will connect to the gateway through HTTP protocol. If set to false, the app will connect to the gateway through the HTTPS protocol.

3. **-UITest_mockService** - When set to true, the app will use mock API services.

Those environment variables can be set through Product - Scheme - Edit Scheme in Xcode.

After adding and processing environment variables, I started to write tests for separate user flows.

There is about four separate user flow in automated UI tests. Those flows are described below.

1. **First user flow** mainly was related to authorization. Specifically, it tested how an app reacts to an unauthorized user. If a user is connected through HTTP protocol, it should show him an alert, asking a user to authenticate. If a user is authorized, the app should not show a login request to a user on the first start. I show an automated test in the listing 23, checking that an unauthorized alert was shown to the user.

2. **The second flow** was related to searching for tickets. If the user types in the search field, it should load found tickets on the screen.

3. **The third flow** was related to clicking on the icon near the Top portfolio, which should redirect the user to the portfolio screen.

4. **The final fourth flow** was related to placing an order and buying a financial instrument.

```
1       func
        ↪  test_ContentView_unauthorizedSheet_shouldShowUnauthorizedSheet_HTTP()
        ↪  {
2               let app = XCUIApplication()
3               // Pass required evnironment variables
4               app.launchEnvironment = [
5                   "-UITest_unauthorized": "true",
6                   "-UITest_mockService": "true",
7                   "-UITest_isHTTP": "true"
8               ]
9               // Launch app in simulator
10              app.launch()
11
12              // Wait one second for existence of the text,
13              // which appears on alert
14              let errorText =
                ↪  app.staticTexts["loginErrorStaticText"]
15              let sheetOpened =
                ↪  errorText.waitForExistence(timeout: 1)
16              XCTAssertTrue(sheetOpened)
17      }
```

Listing 23: Unauthorized alert UI test

6.0.3 Coverage

My testing strategy was to test as much as possible business logic with the help of unit tests, starting from mocking API services and continuing straight to the lower layer of business logic - ViewModels. Furthermore, I covered primary user flows with automated UI testing.

With the help of this testing strategy, I managed to achieve 84,2% total test coverage. In the table below, I describe the test result in more detail.

Table 6.1: Test coverage table

Test Unit	**Coverage in %**
UI View tests	93,8%
Unit tests ViewModels	91,33%
Unit tests Repositories	96,35%
Unit tests Services	50%
Total	84,2%

Of course, automated UI tests could not cover all code in the application, so I also tested it manually. Manual tests helped me to identify some bugs.

6.0.4 Continues Integration

To fully use the potential of automated tests, I decided to add Continues Integration to the project. The main idea of continuous integration (CI) is to constantly introduce minor changes, review the code and push it to the versioning system. Usually, continuous integration comes with automated tests, so any deviations from the norm and bugs could be easily spotted after the developer pushes new changes. [25]

For versioning control, I used GitLab and set up GitLab Runner, which is running locally on my computer. Each time I push changes, the runner compiles code, builds an application, and runs automated unit and UI tests to identify if my new changes broke any of the features.

Conclusion

This thesis aimed to analyze the functionality of new API technology provided by Interactive Brokers - Client Portal API, implement a demo application for iPadOS/iOS, and test the final version of the software.

First of all, I proceeded with analysis, where I compared Client Portal API with other solutions provided by Interactive Brokers, then I designed and built the application for iOS and iPadOS, where users can log in to their accounts, see account information, get historical and real-time stocks' market data, as well as buy and sell those financial instruments. I showed the usage not only of REST API but also of Web Sockets, with the help of which information can be retrieved each second. Integration with Client Portal API was sometimes challenging, not only because this technology is relatively new and many things are not working appropriately yet, but also because of business logic, which needs to be well thought out, and different security nuances of the programming for iOS. After the development, I described the testing process of the application using the automated unit and UI tests with setting up continuous integration.

In this work, I described the app's engineering process so that each market enthusiast, who wants to build a custom trade application for Interactive Brokers, could take this thesis as a template or reference, read it and implement an application that will fulfill their needs.

Bibliography

[1] Group, I. B. Global trading platform - IB trader workstation. *Interactive Brokers LLC [online]*, [cit. 2022-03-28]. Available from: https://www.interactivebrokers.com/en/index.php?f=14099#tws-software

[2] Aljamea, M.; Alkandari, M. MMVMi: A validation model for MVC and MVVM design patterns in iOS applications. *IAENG Int. J. Comput. Sci*, volume 45, no. 3, 2018: pp. 377–389, [cit. 2022-04-05].

[3] LLC, I. B. IBKR mobile - invest worldwide. *App Store [online]*, [cit. 2022-04-02]. Available from: https://apps.apple.com/cz/app/ibkr-mobile-invest-worldwide/id454558592?l=cs&platform=iphone

[4] Group, I. B. API Feature Comparison Table [online]. *Interactive Brokers LLC [online]*, [cit. 2022-03-28]. Available from: https://www.interactivebrokers.com/en/trading/ib-api.php#api-software

[5] Group, I. B. About the Interactive Brokers Group. *Interactive Brokers U.K. Limited [online]*, Jan 2022, [cit. 2022-03-27]. Available from: https://www.interactivebrokers.co.uk/en/index.php?f=41347

[6] Folger, J. Interactive brokers review. *Investopedia [online]*, Sep 2022, [cit. 2022-04-29]. Available from: https://www.investopedia.com/interactive-brokers-review-4587904

[7] Scott, G. Financial Information Exchange (FIX). *Investopedia [online]*, May 2021, [cit. 2022-03-28]. Available from: https://www.investopedia.com/terms/f/financial-information-exchange.asp

[8] Group, I. B. Trading API solutions. *Interactive Brokers LLC [online]*, [cit. 2022-03-28]. Available from: https://www.interactivebrokers.com/en/trading/ib-api.php#api-software

[9] Group, I. B. Clinet Portal API - Getting Started. *Client Portal Web API [online]*, [cit. 2022-04-20]. Available from: `https://interactivebrokers.github.io/cpwebapi/`

[10] Brokers, I. Streaming Websocket Data. *Global Trading Platform - IB Trader Workstation — Interactive Brokers LLC [online]*, [cit. 2022-03-29]. Available from: `https://interactivebrokers.github.io/cpwebapi/RealtimeSubscription.html`

[11] Inc., A. Swift. *Apple Developer [online]*, [cit. 2022-04-05]. Available from: `https://developer.apple.com/swift/`

[12] Gilb, T. *Competitive engineering: a handbook for systems engineering, requirements engineering, and software engineering using Planguage*. Elsevier, 2005, [cit. 2022-04-02].

[13] Larman, C. *Applying UML and patterns: an introduction to object oriented analysis and design and interative development*. Pearson Education India, 2012, [cit. 2022-04-03].

[14] Kocsis, Z. A.; Swan, J. Dependency injection for programming by optimization. *arXiv preprint arXiv:1707.04016*, 2017, [cit. 2022-04-25].

[15] Esbai, R.; Erramdani, M. Model-to-model transformation in approach by modeling: From UML model to Model-View-Presenter and Dependency Injection patterns. In *2015 5th World Congress on Information and Communication Technologies (WICT)*, IEEE, 2015, pp. 1–6, [cit. 2022-04-03].

[16] Brokers, I. Client Portal API FAQ. *Interactive Brokers FAQ [online]*, [cit. 2022-04-21]. Available from: `https://interactivebrokers.github.io/cpwebapi/faq.html`

[17] Inc., A. URLSessionDelegate — Apple Developer Documentation. *Apple Developer Documentation [online]*, [cit. 2022-04-22]. Available from: `https://developer.apple.com/documentation/foundation/urlsessiondelegate`

[18] Hammer-Lahav, E. The oauth 1.0 protocol. Technical report, 2010, [cit. 2022-04-25].

[19] Inc, A. onAppear(perform:) — Apple Developer Documentation. *Apple Developer Documentation [online]*, [cit. 2022-04-23]. Available from: `https://developer.apple.com/documentation/SwiftUI/AnyView/onAppear(perform:)`

[20] denniscm190, D. SwiftUI Stock Charts for iOS. *GitHub [online]*, [cit. 2022-04-24]. Available from: `https://github.com/denniscm190/stock-charts`

[21] Exyte, e. A number of preset loading indicators created with swiftui. *GitHub [online]*, [cit. 2022-04-25]. Available from: `https://github.com/exyte/ActivityIndicatorView`

[22] Klammer, C.; Kern, A. Writing unit tests: It's now or never! In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, 2015, pp. 1–4, [cit. 2022-04-28].

[23] Hunt, A.; Thomas, D. *Pragmatic unit testing in Java with JUnit*. The Pragmatic Bookshelf, 2003, [cit. 2022-04-28].

[24] Inc, A. sink(receiveValue:) — Apple Developer Documentation. *Apple Developer Documentation [online]*, [cit. 2022-04-21]. Available from: `https://developer.apple.com/documentation/combine/publisher/sink(receivevalue:)`

[25] Mohammad, S. M. Continuous Integration and Automation. *International Journal of Creative Research Thoughts (IJCRT), ISSN*, 2016: pp. 2320–2882, [cit. 2022-04-28].

Appendix A

Acronyms

IB Interactive Brokers

IBKR Interactive Brokers Group

API Application Programming Interface

MVC Model-View-Controller

MVVM Model-View-ViewModel

FIX The Financial Information eXchange

iOS iPhone Operating System

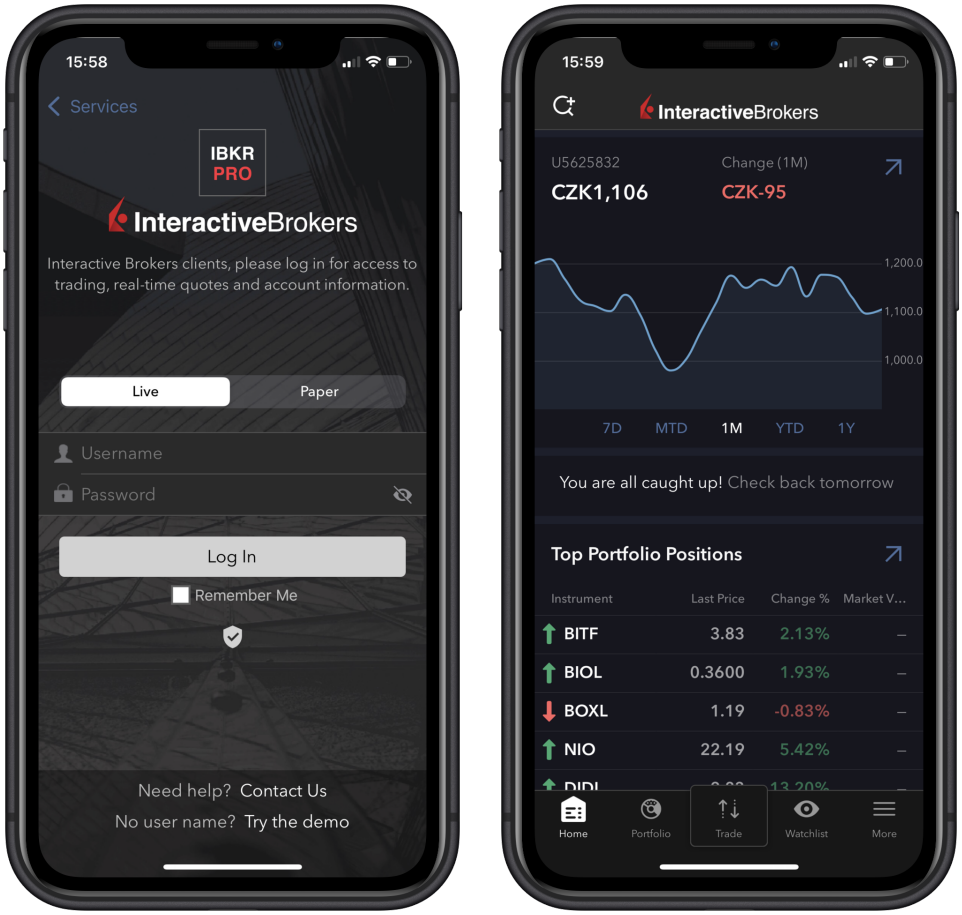
iPadOS iPad Operating System

DTO Data Transfer Object

TWS Trader Workstation

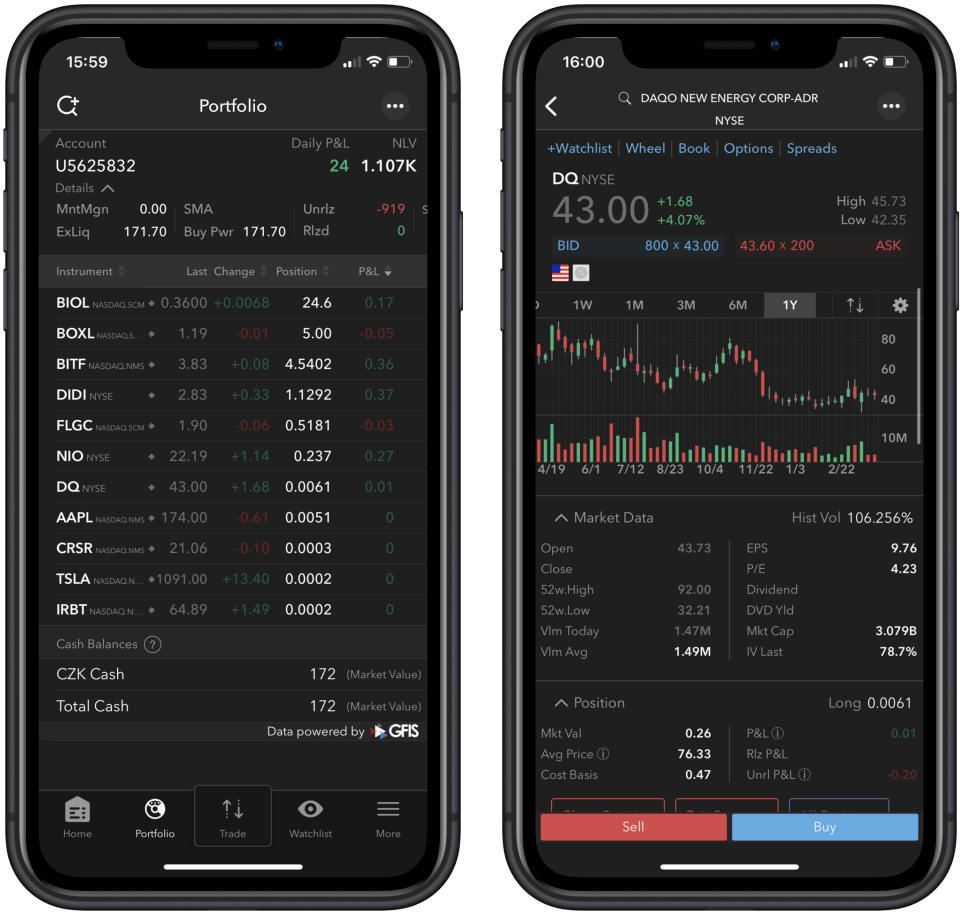
Appendix B

Existing iOS app



(a) Login screen (b) Home screen

Figure B.1: Official application from App Store [3]



(a) Portfolio screen (b) Ticket screen

Figure B.2: Official application from App Store [3]

Appendix C

Use case diagram

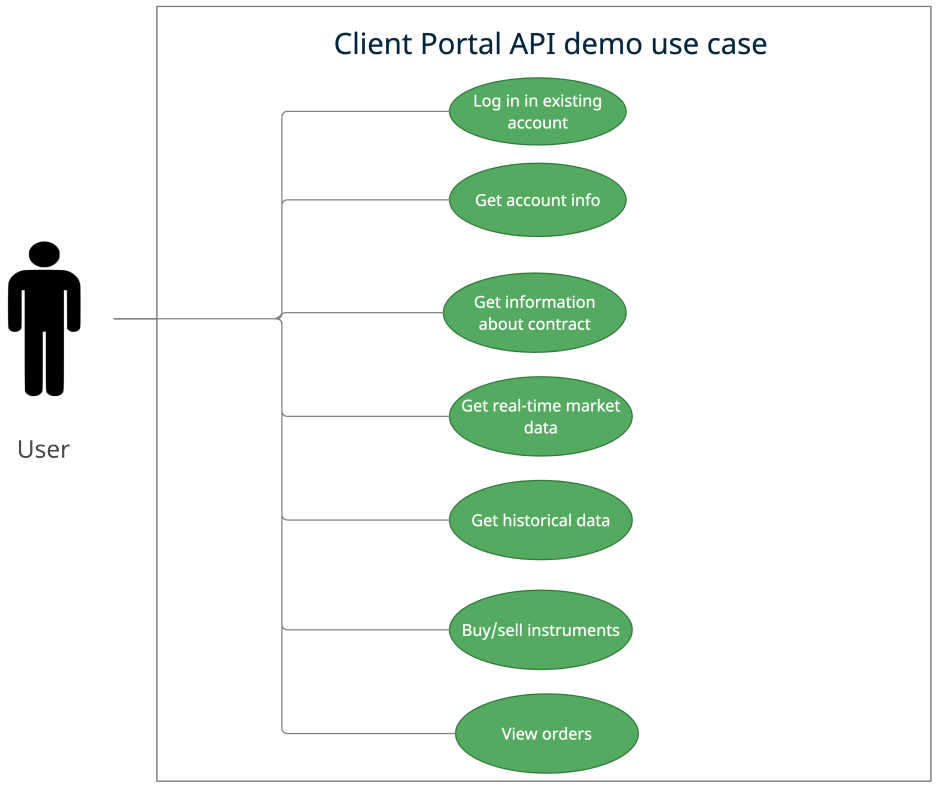


Figure C.1: Use case diagram for Client Portal API demo app

Appendix D

WebSocketService class usage

```
final class TicketViewModel: ObservableObject {
    // Initialization
    // ...
    func loadTickerInfoFromSocket(conid: Int) {
        self.repository.tickle { tickle in
            self.stream.authorize(token:authorizeMessageStr)
            self.stream.sendRepeatedly(message:
            ↪  "smd+\(conid)+{\"fields\":\(APIConstants.STRING_FIELDS)}")
        }
    }
    // Waits for messages
    func retrieveMessages() async {
        do {
            for try await message in stream {
                do {
                    // Decoding and appending data
                }
            }
        }
    }
    // Closes stream, if screen is unmounted
    func onDisappear() {
        self.stream.close()
    }
}
```

Listing 24: Search for name or symbol repository function

Appendix E

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `build` ..... the directory with executables
- `src` ..... the directory of source codes
  - `IBClientAPISwiftUI` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `demo` ..... the directory of demo video
  - `IBClientAPISwiftUI_demo.mp4` .... video demonstrating functionality
- `text` ..... the thesis text directory
  - `thesis.pdf` ..... the thesis text in PDF format
  - `thesis.ps` ..... the thesis text in PS format