iOS Stock Management Application - Paper Trading and Queue Purchase Implementation

Po-Chu Chen
Master of Science in Computer Science
University of California, Riverside
CA, USA
pchen243@ucr.edu

Advisor: Dr. Zizhong Chen
Department of Computer Science and Engineering
University of California, Riverside
CA, USA
chen@cs.ucr.edu

***Abstract*—In this project, we develop a prototype iOS application designed for stock paper trading in the Chinese A-share market. The application integrates multiple key features commonly required in a stock trading system, including portfolio tracking, real-time stock monitoring, mock trading, and algorithmic trading. To achieve this, we adopted modern development practices encompassing front-end design using SwiftUI, backend integration through APIs, and algorithmic implementation tailored for trading strategies. With the support of Sina and Mairui APIs, the app seamlessly retrieves and processes real-time and historical stock data, providing users with an efficient and intuitive trading experience. Considering the significance of the Chinese stock market in the global economy, this project serves as both a practical tool and a research foundation for similar applications. The final prototype demonstrates the feasibility of combining advanced software engineering techniques and financial data analysis into a mobile platform. This work can be extended to various financial domains, including cryptocurrencies, bonds, and other investment markets, offering valuable insights and applications for financial technology (FinTech) innovation.**

Keywords—SwiftUI, iOS Development, Core Data, Paper Trading, Portfolio Management, Real-Time Data Processing, API Integration, Algorithmic Trading, Stock Price Monitoring, User Experience Design (UX)

I. Introduction

In the era of information and technology, mobile devices have become indispensable tools, enabling people to handle both daily tasks and complex activities with unprecedented convenience. Stock trading, traditionally confined to desktop platforms, has evolved into a more accessible and user-friendly experience on mobile devices. As mobile trading applications grow in popularity, developing intuitive and feature-rich platforms for investment management has become an essential focus for modern financial technology (FinTech).

When developing this project, we evaluated multiple frameworks for iOS development, including Objective-C and SwiftUI, both of which have unique advantages and trade-offs. Objective-C, being the older framework, offers extensive backward compatibility and flexibility for integrating legacy systems, making it a reliable choice for certain long-established applications. However, SwiftUI, as Apple's modern declarative framework, provides a more streamlined and intuitive syntax, which significantly reduces development time. Moreover, SwiftUI is deeply integrated with Apple's ecosystem, allowing for seamless creation of adaptive and responsive UIs across all Apple devices. After weighing these trade-offs, we chose SwiftUI for its simplicity, strong community support, and alignment with current industry trends, ensuring a robust and maintainable codebase for the project.

In selecting the mobile platform, we opted for iOS over Android for several reasons. First, iOS is recognized for its user-friendly interface and elegant design, which makes it particularly appealing for financial applications where usability and visual appeal are paramount. Second, iOS users tend to have higher engagement and spending behavior, making it an ideal platform for exploring potential financial technology use cases. While many earlier studies and applications favored Android due to its rapid prototyping and flexibility, the maturity of iOS frameworks like SwiftUI and the growing importance of iOS as a market cannot be overlooked. By focusing on iOS, we aim to leverage its strengths in design and user experience to create an impactful application.

At the heart of SwiftUI is the concept of "View", which represents a single building block of the user interface. A View in SwiftUI can be thought of as an individual "screen" or "page" that the user interacts with, such as the Main Page, Portfolio View, or Order View in this project. Each View is highly customizable and can be composed of smaller sub-views to create complex layouts and functionalities. For example, the Main Page combines multiple subviews like the Portfolio Summary and Watchlist components, all of which are managed independently but seamlessly integrated into the overall interface. In the context of this project, the term View refers to the individual interface frameworks or screens that the user sees and interacts with during navigation. For instance:

- The Main Page View displays the dashboard overview of the user's portfolio and market data.
- The Portfolio View provides details about owned stocks and their performance.

- The Order View facilitates mock trading functionalities, including stock purchases and queue purchases.

This modular design approach not only makes the development process more efficient but also ensures a clear separation of concerns. Each View handles a specific aspect of the application's functionality, making the codebase easier to manage and scale. Moreover, SwiftUI's state management tools allow Views to react instantly to changes in data, such as updating the portfolio value or displaying new stock information, ensuring a responsive and dynamic user experience.

The prototype we developed for this project implements a comprehensive stock paper trading system, providing users with the following key functionalities:

1. *Stock Price Information Retrieval:* By integrating APIs (such as those from Sina and Mairui), the application allows users to query real-time stock prices and historical data from the Chinese A-share market. This real-time data processing capability ensures users can make informed decisions while monitoring stock trends.

2. *Portfolio Management:* To manage users' assets effectively, we implemented a storage mechanism that persists user data. This ensures that users' portfolio information, such as stock holdings and account balances, is not lost even if the app is restarted or interrupted.

3. *Mock Trading:* Our application supports simulated trading by enabling users to execute buy and sell transactions in a risk-free environment. To achieve this, we developed functionality to track transaction history, check account balances, deduct or add funds, and update portfolio information in real-time.

4. *Queue Purchase:* One of the innovative features of the app is its Queue Purchase functionality, which introduces algorithmic trading capabilities. This feature allows users to automate stock purchases by monitoring real-time market data and executing trades when predefined conditions are met. The implementation relies heavily on real-time data processing to ensure accuracy and reliability.

By combining these features, our application serves as a practical tool for exploring stock trading strategies in a simulated environment. Additionally, this project contributes to the broader FinTech landscape by offering a scalable and adaptable prototype for financial applications. The framework and methodologies used in this project can be extended to various financial scenarios, such as cryptocurrency trading, quantitative trading, asset management, and insurance claims processing.

In summary, this project demonstrates the potential of modern iOS development frameworks and real-time data integration to enhance the mobile trading experience. It not only showcases the feasibility of building a comprehensive stock trading system on iOS but also provides a foundation for future research and applications in FinTech innovation.

II. Stock APP Main Function

In this part, we will introduce the core features of our paper trading system. In this part, we will introduce the core features of our paper trading system. The application is divided into several key components to provide users with a seamless and comprehensive trading experience. Each component is carefully designed to simulate real-world trading processes while ensuring an intuitive and user-friendly interface.

A. Main Page

The Main Page of the application in Figure 1. acts as the central dashboard, offering a comprehensive overview of the user's trading activities and portfolio performance. It is designed to present crucial information in a visually appealing and intuitive manner, ensuring a seamless user experience.

At the top of the Main Page, users can find a Market Overview section, which displays real-time updates for major stock indices such as the Shanghai Composite Index, NASDAQ, and Dow Jones. Each index is accompanied by its current value and percentage change, providing users with an instant snapshot of the market's overall performance. This feature ensures that users stay informed about market trends without the need for external sources.

The centerpiece of the page is the Portfolio Summary, prominently displayed in a purple card. This section showcases the user's Total Portfolio Value, along with the percentage change compared to the previous period. This at-a-glance summary allows users to quickly assess their overall financial performance. Additionally, a View All button below the summary leads to detailed portfolio analytics, offering deeper insights into the user's financial trends and historical data.

Below the portfolio summary is the Your Portfolio section, where users can view individual stocks they currently own. Each stock entry includes the stock's name, ticker symbol, the user's holdings (e.g., Inventory: 5), and the current profit or loss percentage. Gains are displayed in green, while losses appear in red, enabling users to quickly assess the performance of each stock. A View All link provides access to a detailed breakdown of the user's entire portfolio, including stock-specific performance metrics over time.

The next section, Your Watchlist, is dedicated to stocks that the user is monitoring but has not yet purchased. This feature displays the stock's name, ticker symbol, current price, and percentage change. To assist users in analyzing potential investments, a Graph button is included, which allows them to view detailed price trends for each stock in their watchlist.

Finally, the Navigation Bar at the bottom of the Main Page provides seamless access to other parts of the application, including the Home page, Search function, Account management, and Order placement. This tab-based design ensures that users can navigate the app effortlessly and access essential features without disrupting their workflow.

Figure 1. Main Page View

1) Portfolio: The Portfolio section is designed to provide users with a clear and interactive overview of the stocks they own. When a user purchases a stock, the stock automatically appears in the Portfolio Card along with its corresponding inventory amount. This section adopts a horizontally scrollable layout, allowing users to browse through their stock holdings in a seamless and intuitive manner. The design prioritizes usability, enabling users to quickly access essential information about their portfolio.

At the top of the portfolio section, a Total Portfolio Value Card is prominently displayed like Figure 2. This card dynamically calculates and displays the total value of the user's portfolio, which is computed by summing the product of the inventory amount and the current stock prices for all owned stocks. To ensure user privacy and security, a toggleable eye icon is provided, allowing users to hide or reveal the portfolio value as needed. This feature caters to scenarios where users may want to prevent others from viewing their financial data in shared or public environments.

The portfolio value is kept up-to-date through a backend system that queries the stock API every 5 seconds to fetch real-time price updates. In addition, the system is designed to trigger a refresh whenever the user performs a pull-to-refresh gesture by scrolling to the top of the screen. This dual mechanism ensures that users always have access to accurate and timely information, enhancing the overall reliability and user experience of the app.

Each Portfolio Card serves as a compact summary of a specific stock owned by the user. The card provides an intuitive preview of key stock details, including:

1. Percentage change: Displayed in green for positive changes and red for negative changes to visually indicate stock performance.
2. Current Price: Shows the latest stock price retrieved from the API.
3. Inventory Amount: Displays the number of shares held by the user.

To facilitate deeper exploration of individual stocks, users can tap on any Portfolio Card to open the Stock Detail View, which provides comprehensive information about the selected stock, including its historical performance, trading volumes, and additional market insights. This interactive design makes it easier for users to retrieve detailed stock information without navigating through multiple menus, creating a smooth and engaging user experience.

Our development approach emphasizes user-friendliness and interactivity, ensuring a seamless and engaging user experience. The horizontal scrolling mechanism for portfolio cards aligns with common iOS gesture-based interactions, making navigation intuitive and natural. Real-time updates, powered by periodic API queries and the pull-to-refresh gesture, provide users with the most accurate and up-to-date portfolio information. Additionally, the ability to tap on a Portfolio Card to access detailed stock information in the Stock Detail View minimizes navigation complexity and enhances accessibility. These features are carefully designed to keep users engaged and connected to the app, making it an indispensable tool for monitoring and managing their investments.

Figure 2. Portfolio Card

2) *Watchlist:* The Watchlist feature in Figure 3 is designed to help users efficiently track specific stocks they are interested in. It provides an intuitive interface for managing and monitoring these stocks, ensuring users have quick access to relevant market information.

To add stocks to the watchlist, users have two primary options: they can either use the Search Page View, where they can search and select specific stocks to add, or they can tap the "View All" button within the watchlist section on the Main Page to directly edit their watchlist. Once a stock is added, it appears as a Watchlist Card within the watchlist section. These cards display essential stock information, such as the current price and percentage change in value, similar to the portfolio cards. The information is visually organized to enable users to quickly gauge the performance of their tracked stocks.

The watchlist adopts a vertical scrolling layout, allowing users to browse through their tracked stocks with ease. Additionally, users can tap the "View All" button to access the Edit Watchlist View, where they have full control over the contents and order of their watchlist.

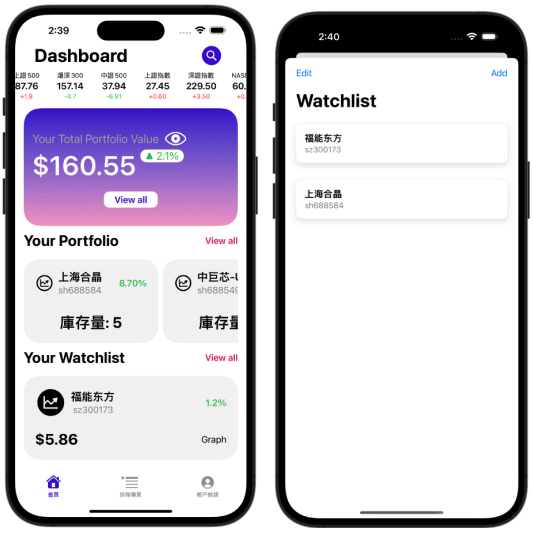


Figure 3. Watchlist

In the Edit Watchlist View shown in Figure 4, users can add, remove, or rearrange the items in their watchlist through a highly interactive design:

- Adding Stocks: By clicking the "Add" button, the Search View is triggered, allowing users to search for and add stocks directly to their watchlist. The Search View integrates with stock APIs to fetch real-time data, ensuring users can select from the most up-to-date stock options.
- Removing Stocks: Users can swipe left on any stock card to delete it from the watchlist. Alternatively, they can use the Edit mode to remove multiple stocks in one action.
- Reordering Stocks: The watchlist supports drag-and-drop functionality, enabling users to rearrange the order of their tracked stocks with ease. This allows users to prioritize certain stocks based on their personal preferences or market importance.

These features are implemented with a focus on user-friendliness and interactivity. For example, the drag-and-drop feature provides a tactile and intuitive way to organize the watchlist, mimicking physical interactions that users find natural on mobile devices. Additionally, the vertical scroll and real-time stock data updates ensure users can effortlessly browse and access the most current market information without navigating away from the watchlist.

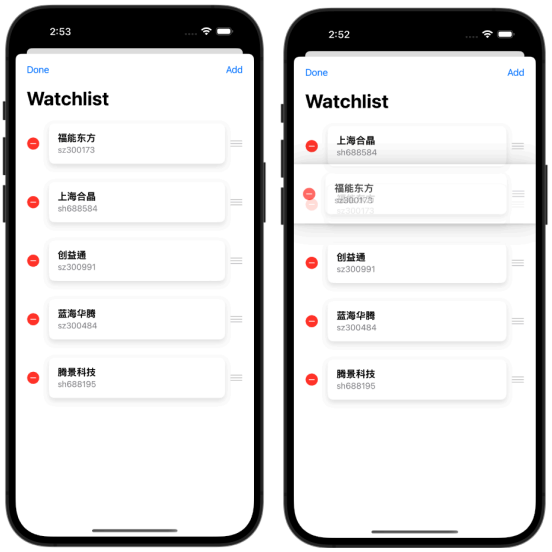


Figure 4. Editable Watchlist

B. *Account*

The Account section serves as the financial hub of the application, allowing users to manage their simulated funds and track their transaction history. This section is designed to mimic real-world financial operations, providing a simplified but functional framework for managing deposits and withdrawals.

At the core of the Account section is the Balance Display in Figure 5, which prominently shows the user's current account balance. For security and privacy, users can toggle the visibility of their balance using the eye icon, ensuring their financial information remains private in public or shared environments.

To streamline the development process, this project does not connect to real-world bank accounts, as such functionality exceeds the scope of the simulation. Instead, we implement a virtual bank account within the app, which simulates the process of depositing and withdrawing money. This approach simplifies the user experience while laying the groundwork for potential future enhancements. Integrating real bank accounts through APIs is feasible in

the future, should the app be extended for real-world financial applications.

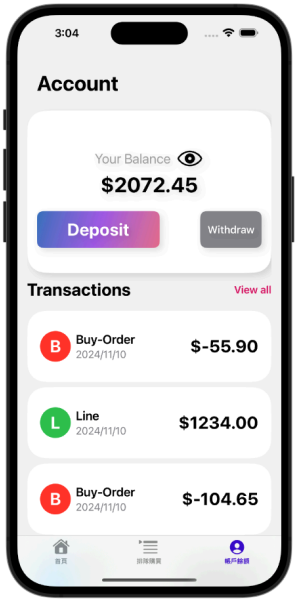


Figure 5. Account Balance

1) *Deposit:* The Deposit feature enables users to add funds to their simulated account and manage their balance. When users tap the "Deposit" button, the NewDepositView appears (Figure 6), providing an intuitive interface to input deposit details. This interface includes three deposit options: ApplePay, simulating credit card transactions; LinePay, representing debit card transactions; and Bank, mimicking bank account transfers. To ensure clarity and functionality, the deposit type is a mandatory field, while an optional description field allows users to add notes about the transaction. For instance, a user could enter "This is the charging from credit card" as additional context. Once the deposit details are completed and submitted, the application updates the user's balance accordingly and records the transaction in the Transaction History section (Figure XX). Each transaction card displays relevant information, such as the deposit type, amount, and timestamp, ensuring users can easily track their financial activities.

A critical aspect of this feature is the persistent nature of the balance. In a simulated trading environment, it is essential that the user's balance remains consistent across app sessions. Without such persistence, restarting the app and finding the balance reset to zero would disrupt the user experience. To achieve this, the project leverages **Core Data**, Apple's built-in persistence framework, which stores and manages data locally on the device.

Core Data plays a key role in ensuring data consistency and reliability. Within this framework, entities such as Account and Transaction are defined to structure the data. The Account entity tracks the user's current balance, while the Transaction entity stores detailed records of every deposit, withdrawal, or trade. Each transaction includes attributes such as type, amount, date, and optional description. When a user submits a deposit, the app creates a new Transaction object and updates the corresponding balance in the Account entity. These updates are automatically saved to the device's local storage, ensuring that the data persists even if the app is closed or the device is restarted.

The integration of Core Data also enables seamless interaction between the data layer and the user interface. When the Account View is loaded, the app fetches the latest balance and transaction history from Core Data. SwiftUI's declarative syntax ensures that any changes to the data are immediately reflected in the interface. For example, when a deposit is added, the updated balance and transaction card appear in real time without requiring additional user actions.

While this implementation focuses on local data persistence, it also provides a foundation for future enhancements. For instance, Core Data can integrate with Apple's CloudKit to enable synchronization across multiple devices, allowing users to access their account data seamlessly. Additionally, the existing infrastructure can support advanced features such as multi-currency accounts or in-depth financial analytics, further enhancing the app's functionality.

By combining a user-friendly interface with the robust capabilities of Core Data, the Deposit feature ensures that users can confidently manage their account balance. The design prioritizes both simplicity and reliability, making it an integral part of the app's simulated trading environment while laying the groundwork for future scalability.

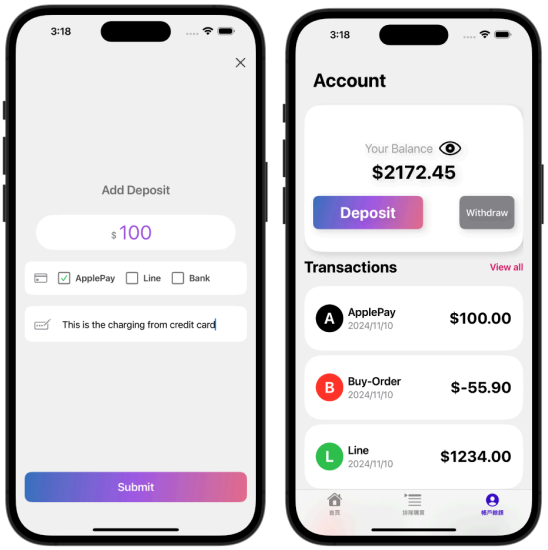


Figure 6. Deposit

2) *Withdraw:* The Withdraw feature mirrors much of the functionality and logic of the Deposit feature, offering users a simple and intuitive interface for deducting funds from their simulated account. Unlike the deposit process, the Withdraw

View does not include multiple options, as the act of withdrawing funds is typically a straightforward deduction from the user's current balance.

When a user taps the "Withdraw" button, the Withdraw View appears (Figure 7), allowing them to enter the withdrawal amount and an optional description. After submitting the withdrawal, the entered amount is deducted from the user's current balance. To ensure clarity in transaction history, withdrawal amounts are displayed as negative values on the Transaction Card, visually reflecting the deduction from the account.

The updated balance is immediately visible on the Account View, demonstrating a real-time reflection of the user's available funds. This synchronization between the withdrawal action and the balance update is achieved using Core Data, which ensures that all changes to the balance and transaction history are stored persistently on the device. Just like deposits, withdrawal records include attributes such as the amount, date, and optional description, providing users with a clear and accessible log of their financial activities.

From a design perspective, the Withdraw View prioritizes simplicity and user-friendliness. The absence of additional options streamlines the process, ensuring users can quickly and easily remove funds from their account without unnecessary complexity. While the feature is simulated in the context of this app, it provides a foundation for future extensions. For example:

- In a real-world application, withdrawals could be linked to external financial systems or payment gateways, such as a user's bank account or debit card. This functionality could be achieved through secure financial APIs, ensuring seamless integration with external systems.
- The current implementation also allows for potential scalability. For instance, integration with a financial agent interface could enable users to interact with brokers or automated trading systems for more advanced financial operations.

The Withdraw feature underscores the importance of maintaining simplicity while ensuring functionality in financial simulations. By combining clear transaction records, persistent balance updates, and a straightforward interface, this feature provides users with a realistic and engaging way to manage their funds in a simulated trading environment. Additionally, it ensures that each withdrawal is accurately reflected in both the account balance and transaction history, reinforcing transparency and user trust. By incorporating a user-friendly design and responsive feedback mechanisms, the system enhances accessibility while maintaining the integrity of financial tracking. This feature not only supports core trading functions but also lays the groundwork for potential enhancements, such as automated withdrawal rules or integration with real banking systems in future iterations.

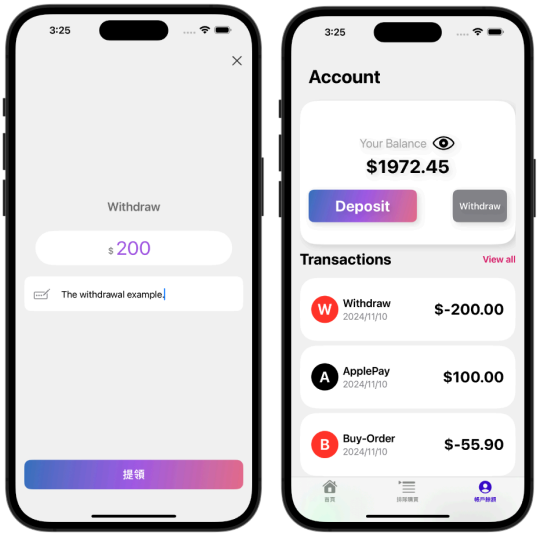


Figure 7. Withdraw

C. *Search*

The Search feature is a fundamental component of this stock trading app, allowing users to efficiently locate and retrieve information about stocks they are interested in. Developing this feature requires establishing a method to retrieve the complete list of A-share stocks. Leveraging the Mairui API, we were able to obtain the entire A-share stock list without incurring additional costs. However, the free API key account comes with a strict request limit, necessitating a more sustainable solution to ensure the availability of data for users at all times.

To address this, we utilize Core Data within SwiftUI to locally store the stock list retrieved from the API. This solution offers several advantages. First, data stored in Core Data persists on the user's device even if the app is closed or the background program is terminated. It will only be deleted if the user explicitly removes the app or resets the app's cache, which we assume will occur infrequently. This persistence ensures a seamless user experience by eliminating the need for repeated API requests.

To address this, we utilize Core Data within SwiftUI to locally store the stock list retrieved from the API. This solution offers several advantages. First, data stored in Core Data persists on the user's device even if the app is closed or the background program is terminated. It will only be deleted if the user explicitly removes the app or resets the app's cache, which we assume will occur infrequently. This persistence ensures a seamless user experience by eliminating the need for repeated API requests.

1. *Searching Bar:* The Search Bar in Figure 9 is accessible via the magnifier icon on the Main Page

View, which opens a comprehensive stock list interface. Users can search for specific stocks using either the stock ID (numerical) or the stock name (textual). As the user types into the search bar, the underlying stock list dynamically updates to display matching results in real time. This functionality enhances usability by allowing users to quickly narrow down their options without needing to navigate through the entire stock list.

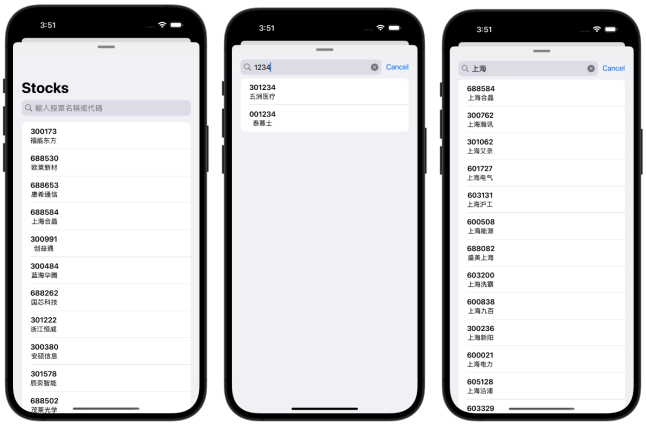


Figure 8. Search Bar

2. *Stock Detail View:* Once users locate their desired stock, they can tap on the stock name to open the Stock Detail View. This view provides comprehensive information about the selected stock, including:
    - Price Details: Open Price, Bid Price, Volume, Turnover, and other key metrics.
    - Candlestick Charts: Daily, weekly, and monthly candlestick charts are displayed, providing visual insights into the stock's historical performance. These charts are retrieved directly from the Sina API, ensuring accuracy and reliability.

The Stock Detail View (Figure 9) also includes interactive features to enhance user engagement:

1. Add to Watchlist: By clicking the heart icon, users can add the stock to their watchlist for easy monitoring. This action is reflected immediately in the watchlist section of the app.
2. Buy and Sell Buttons: The Buy and Sell buttons, located below the charts, are integral to the app's paper trading functionality. These buttons allow users to execute mock trades directly from the detail view. The implementation of these features is detailed further in the Order section.

The Stock Detail View also includes interactive features to enhance user engagement:

1. Add to Watchlist: By clicking the heart icon, users can add the stock to their watchlist for easy monitoring. This action is reflected immediately in the watchlist section of the app.

2. Buy and Sell Buttons: The Buy and Sell buttons, located below the charts, are integral to the app's paper trading functionality. These buttons allow users to execute mock trades directly from the detail view. The implementation of these features is detailed further in the Order section.

Figure 9. Stock Detail View

D. Order

The Order feature is the centerpiece of this paper trading application, designed to provide a seamless and realistic trading experience for users. To ensure convenience and accessibility, users can place orders from two primary entry points: the Stock Detail View and the Watchlist View. This dual-entry system allows users to locate and trade stocks efficiently, whether they are actively searching for new opportunities or monitoring their pre-selected watchlist.

When initiating a trade, users are presented with an Order Form, where they can specify details such as transaction type (buy or sell) and quantity. For buy orders, the system dynamically checks the user's current balance to ensure sufficient funds, while for sell orders, it verifies the availability of shares in the user's portfolio. The app fetches the latest stock price in real time to provide users with accurate market data, ensuring that their trading decisions are informed by up-to-date information. Once the user confirms the order, the app processes the transaction by updating the portfolio and adjusting the balance accordingly. Each completed trade is logged in the Transaction History, capturing essential details like stock name, transaction type, quantity, price, and timestamp, providing users with a transparent and detailed record of their activities.

Beyond the basic trading functionalities, the Order feature also serves as a valuable educational tool. By allowing users to simulate trades in a risk-free environment, the app provides a platform for exploring different investment strategies and understanding the mechanics of trading. Features like the Queue Purchase system further enhance this learning experience, offering users a glimpse into automated trading processes.

1) *Buy:* The Buy functionality is a key component of the Order system, enabling users to simulate the purchase of stocks within the app. After clicking the "Buy" button on either the Stock Detail View or Watchlist View, a number pad interface is presented to the user (Figure XX). This interface allows the user to input the number of shares they wish to purchase. To ensure accuracy and transparency, the total cost of the transaction is displayed dynamically based on the number of shares entered and the current stock price.

One critical backend validation occurs during this process: the app checks whether the user has sufficient funds in their account to complete the purchase. If the user's balance is insufficient, the "Place Order" button is disabled, preventing the transaction from proceeding. Additionally, a placeholder message appears, notifying the user that their available funds are not enough to cover the transaction (Figure 10). This validation step ensures that users can only execute orders that their balance can support, maintaining the integrity of the simulated trading system.

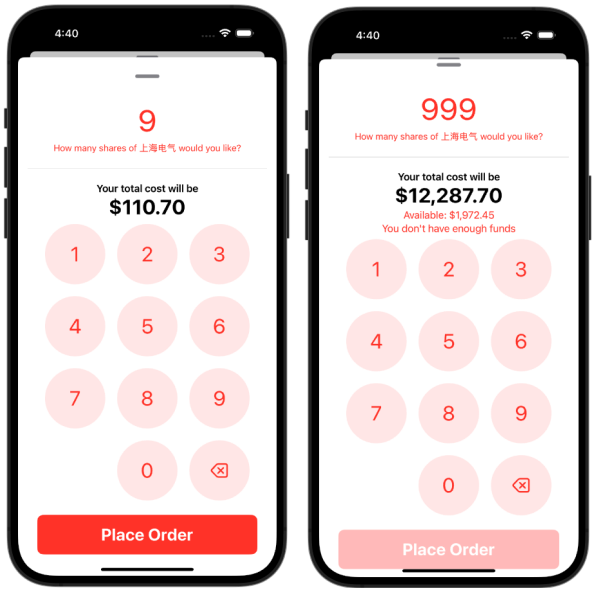


Figure 10. Buy Place Order

If the user has sufficient funds and successfully clicks the "Place Order" button, the transaction is processed as part of the paper trading system. Since this is a simulation, no real monetary exchange occurs; instead, the backend system records the transaction, updates the user's portfolio, and adjusts their account balance accordingly.

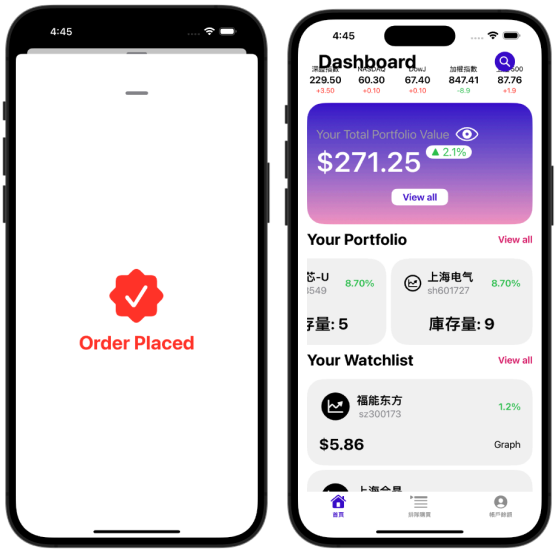


Figure 11. Completion of Buy Place Order

After a successful purchase, several updates occur across the app to reflect the new transaction:

1. *Portfolio Update:* The purchased stock is added to the user's portfolio, or, if the stock already exists in the portfolio, the number of shares is increased accordingly. This update is immediately visible on the Portfolio View in the Main Page View (Figure 11), which dynamically recalculates and displays the updated total portfolio value.
2. *Account Transaction Log:* In the AccountBalanceView, a new transaction card is added to the transaction history, as shown in Figure 12. This card is labeled with the type "Buy-Order", showing the negative amount corresponding to the transaction (i.e., stock price multiplied by the number of shares purchased). The account balance is also adjusted in real time, deducting the total cost of the purchase. For simplicity, transaction fees are not included in the calculation.

The entire process is designed to mimic the flow of real-world stock trading while maintaining the simplicity and flexibility required for a paper trading environment. The use of real-time updates ensures that users have a seamless and transparent experience, with all relevant data synchronized across different views within the app.

By combining user-friendly interface design with robust backend validations, the Buy feature provides users with a realistic yet risk-free way to practice stock trading. This functionality not only supports the educational purpose of the app but also lays the groundwork for potential extensions, such as the inclusion of transaction fees, advanced order types, or integration with real financial systems.

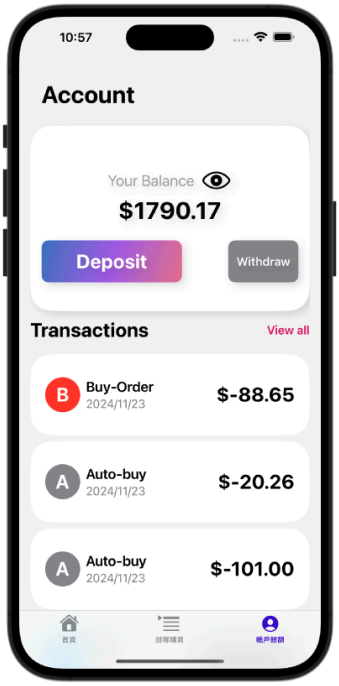


Figure 12. Account Post Buy Transaction

2) *Sell:* The Sell functionality complements the Buy feature, allowing users to simulate the process of liquidating their stock holdings in a seamless and realistic manner. Similar to the Buy feature, the user initiates the process by tapping the "Sell" button on either the Stock Detail View or Watchlist View. Upon doing so, a number pad interface is presented, enabling the user to input the number of shares they wish to sell (Figure 13). As the user enters the quantity, the app dynamically calculates and displays the total value of the transaction based on the current stock price.

The Sell functionality complements the Buy feature, allowing users to simulate the process of liquidating their stock holdings in a seamless and realistic manner. Similar to the Buy feature, the user initiates the process by tapping the "Sell" button on either the Stock Detail View or Watchlist View. Upon doing so, a number pad interface is presented, enabling the user to input the number of shares they wish to sell (Figure XX). As the user enters the quantity, the app dynamically calculates and displays the total value of the transaction based on the current stock price.

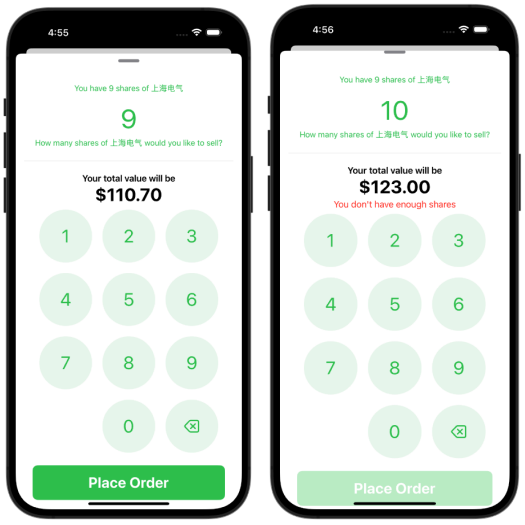


Figure 13. Sell Place Order

Once the user enters a valid quantity and clicks the "Place Order" button, the transaction is processed as follows:

1. *Portfolio Update:* The app deducts the sold shares from the user's inventory. If the user sells all shares of a stock, the corresponding Portfolio Card is removed from the Portfolio View. For partial sales, the card dynamically updates to reflect the remaining quantity of shares. These changes are immediately visible on the Main Page View (Figure 14), ensuring the user's portfolio remains accurate and up to date.
2. *Account Balance Update:* The proceeds from the sale are added to the user's balance, calculated as the product of the stock price and the number of shares sold. This updated balance is displayed in the Account View and logged as a "Sell-Order" in the Transaction History. The transaction card includes details such as the stock name, number of shares sold, total value of the transaction, and timestamp, providing users with a clear record of their financial activities.

To maintain consistency and prevent data loss, all updates to the user's portfolio and transaction history are stored locally using Core Data. When a sell order is executed, Core Data is updated to reflect the changes in the user's inventory and account balance. This ensures that the portfolio and transaction records persist across app sessions and are instantly retrievable upon reopening the app. The integration of Core Data with SwiftUI ensures that all updates are synchronized in real time with the app's user interface, providing a seamless and transparent experience for the user.

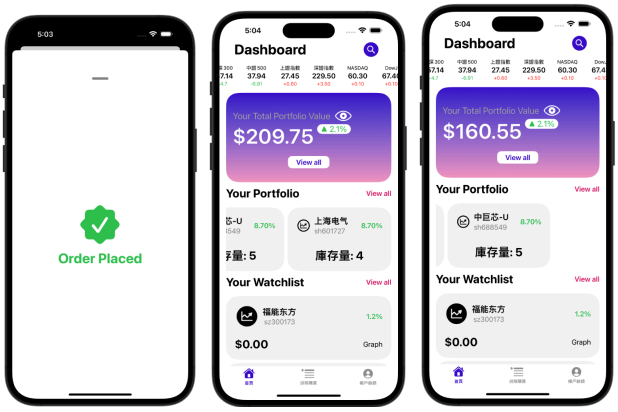


Figure 14. Sell Order Placed

When a sell order is successfully placed, the transaction is recorded in the AccountBalanceView as a Sell-Order entry within the Transaction History (Figure 15). Each entry includes the transaction type, the total value of the sale, and the date of execution, providing users with a transparent log of their trading activities. Simultaneously, the Portfolio View is updated to reflect the remaining shares of the sold stock. If a user sells all shares of a particular stock, the corresponding Portfolio Card is removed from the view after the portfolio refreshes. This ensures that the portfolio accurately represents only the stocks still held by the user, maintaining a clear and consistent overview of their investments.

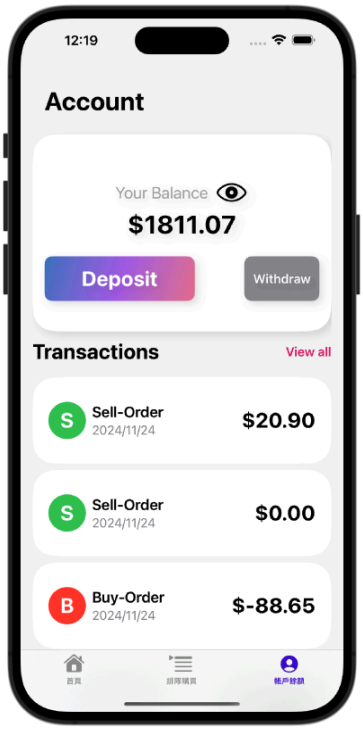


Figure 15. Account Post Sell Transaction

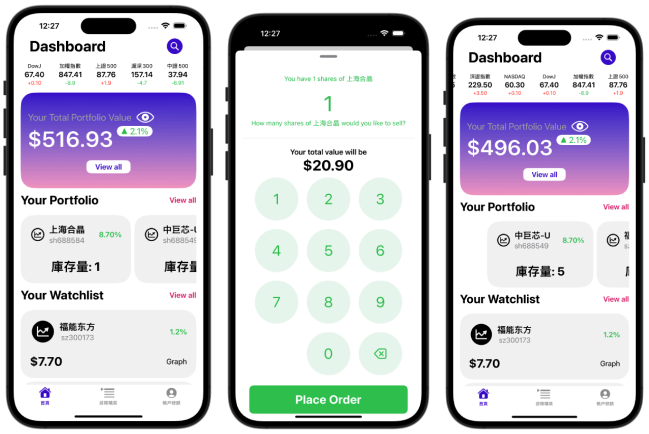


Figure 16. Portfolio Post Sell Transaction

III. API USAGE

To retrieve A-share stock information, our application primarily relies on APIs provided by Mairui Data (MyData) and Sina A Share. These APIs offer comprehensive resources, including stock lists, bidding information, and real-time prices, which are crucial for the app's functionality. Both APIs provide free access with certain limitations, which we carefully considered during development to ensure smooth operation.

One of the main challenges with using free API plans is the restricted number of requests. For example, Mairui Data allows only 50 requests per day under its free plan, though there is no time limit for the account. This limitation makes the API suitable for prototyping purposes, as the frequency of API calls during development is relatively low. However, to optimize API usage and prevent unnecessary requests, we integrated Core Data as a temporary local database within the app.

The system is designed to request the full stock list from Mairui API only during the initial setup—either the first time the app is installed or when the simulator is used. This stock list is then stored in Core Data, effectively acting as a cache. Subsequent data queries are directed to Core Data rather than the API, allowing the app to operate efficiently without exceeding the request limit. By treating Core Data as a caching layer, the app reduces dependency on external APIs while ensuring fast and consistent data access.

To simulate real-time updates, the system is programmed to refresh the stock list every 10 days by re-fetching data from the API. This approach strikes a balance between minimizing API usage and maintaining an up-to-date stock list for users. Through this mechanism, the app ensures that users can access the latest public A-share stock information without compromising the stability of the system.

To address the potential risk of exceeding API request limits or encountering invalid API keys, we implemented a fault-tolerant mechanism for managing API keys. Specifically, we registered three separate API keys, which the system selects randomly for each request. This strategy distributes the load across multiple keys, reducing the likelihood of

overloading a single key. Additionally, this setup provides a fallback in case one of the keys becomes invalid or unavailable, ensuring uninterrupted service for users.

The combination of using Core Data as a caching layer and managing multiple API keys makes the system highly robust and efficient. These measures allow the app to remain within the constraints of free API plans while providing a seamless user experience. The integration of these APIs has proven to be feasible and effective for the development and testing of the prototype.

A detailed breakdown of the Mairui Data API usage, including request limits and key configurations, is summarized in following Table 1.

Certificate Version	Daily Request Limit	Total Request Limit	Usage Period	Price
Free ✅ Version	50 requests	Unlimited	Unlimited	Free
Monthly Subscription	Unlimited	Unlimited	1 month	¥168
Annual Subscription	Unlimited	Unlimited	1 year	¥588
Gold Version	10,000 requests	Unlimited	5 years	¥158
Platinum Version	Unlimited	2 million requests	Unlimited	¥188
Diamond Version	Unlimited	Unlimited	5 years	¥988

Table 1. Mairui API Usage Plan

The Sina A Share API stands out compared to the Mairui Data API due to its complete lack of request limitations. It is entirely free to use, even without requiring the registration of an API key or license. This makes it particularly advantageous for retrieving real-time stock data, as developers do not have to worry about exhausting request quotas or managing API keys.

However, there is one critical requirement for accessing the data through the Sina API: all requests must include a specific Referer header with the value "https://finance.sina.com.cn/". If this header is omitted, the API will return a “403 Forbidden” response, denying access to the requested resources.

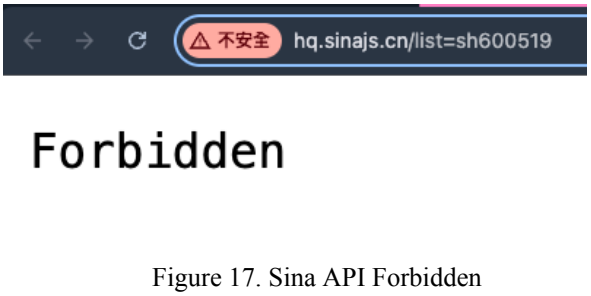


Figure 17. Sina API Forbidden

To address this, we implemented a straightforward solution in our SwiftUI backend by ensuring every URLRequest sent to the Sina API includes the required Referer header. Here is an example of how this is handled programmatically:

```
var request = URLRequest(url: url)
request.setValue("https://finance.sina.com.cn/", forHTTPHeaderField: "Referer")
print("getting.....")
URLSession.shared.dataTask(with: request) { [weak self] data, response, error in
```

By adding the Referer header shown in Figure 18 to each request, we ensure compliance with the API’s access policy and prevent any “Forbidden” errors(Figure 17). This small adjustment allows seamless integration with the Sina A Share API, enabling us to fetch real-time stock data such as current prices, trading volumes, and percentage changes.

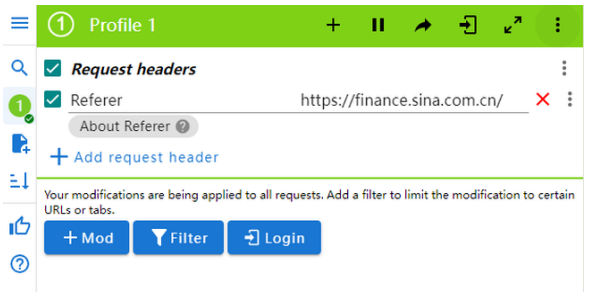


Figure 18. Request Headers on Sina API

This approach ensures that our application can fully leverage the Sina API’s capabilities while maintaining stability and reliability. Furthermore, by automating the inclusion of the required header in our network request logic, we eliminate the need for manual intervention or additional configurations, simplifying the development process.

Now let’s dive into the detailed request and response body for each API. For the Sina A Share API, requests typically follow a standard HTTP GET method. The request URL includes specific parameters such as the stock code, while the response body is usually formatted in JSON or a similar lightweight structure. The response contains key information, including stock prices, bid/ask data, and historical trading volumes.

E. Mairui - MyData

The Mairui Data API is a crucial resource for retrieving comprehensive stock information for all listed A-share stocks. This API provides essential data such as stock codes, stock names, and their respective trading markets, all in a standardized JSON format. Developers can use the HTTP GET method to request data or access it directly through a browser.

The API endpoint is structured to include a unique license key, allowing for personalized access. For instance, in the URL http://api.mairui.club/hslt/list/{your_license_key}, {your_license_key} represents the registered API license required for authentication. The returned JSON response typically includes fields like stock code (dm), stock name (mc), and trading market (jys), where the market is denoted as sh for the Shanghai Stock Exchange or sz for the Shenzhen Stock Exchange. For example, a request for stock list data

might return entries such as “601398” (Industrial and Commercial Bank of China) under the sh market. A concrete example is shown in Figure 19 with specifications in Table 2.

Field Name	Data Type	Description
dm	String	The six-digit stock code. For example, 601398 represents ICBC (Industrial and Commercial Bank of China).
mc	String	The name of the stock. For example, “Industrial and Commercial Bank of China.”
jys	String	The market in which the stock is listed. Possible values: sh (Shanghai Stock Exchange) and sz (Shenzhen Stock Exchange).

Table 2. Mairui API Specification

Below is an example of the JSON response for the API:

```
[
    {
        "dm": "601398",
        "mc": "工商銀行",
        "jys": "sh"
    },
    {
        "dm": "000001",
        "mc": "平安銀行",
        "jys": "sz"
    }
]
```


Figure 19. Response JSON from Mairui API

The API updates its stock list daily at 16:00 to ensure data remains accurate and up-to-date. However, free-tier accounts are limited to 50 requests per day, which can be a challenge in high-demand scenarios. To mitigate this, our application employs Core Data as a local cache. During the app’s initial setup or first use, the full stock list is fetched and stored in Core Data. From that point onward, the app reads the stock data from the local cache rather than making redundant API calls, ensuring efficient resource usage and compliance with API limits. The stock list is programmed to refresh every 10 days via the API to simulate real-time updates while minimizing request frequencies.

By utilizing the Mairui API, our application can provide users with a complete and structured stock database. This forms the backbone for key functionalities such as the stock search, watchlist management, and trading operations. Through a combination of API integration and local caching, we achieve a balance between efficient data retrieval, reliability, and reduced external dependencies.

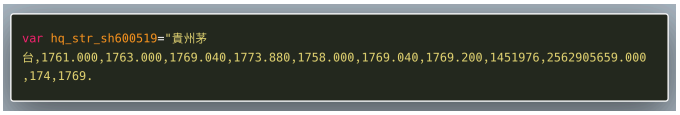
The Mairui Data API offers an extensive range of stock-related information, including indices, industry lists, company details, fund flows, gainers and losers pools, margin trading, and real-time transactions. These features make it a robust and versatile option for comprehensive financial analysis. However, its free version imposes daily request limits and requires a registered license key, which adds complexity to its integration. While these capabilities are impressive, they go beyond the scope of our prototype, which mainly focuses on retrieving detailed stock price information.

To address this, we opted for a simpler solution by integrating the Sina A Share API. Although it offers fewer features, the Sina API provides essential real-time stock price data without request limits or the need for license registration. This makes it a more practical choice for our lightweight prototype, enabling efficient data retrieval while avoiding the constraints imposed by Mairui. This tradeoff reflects our focus on functionality and simplicity, ensuring the prototype meets its core requirements without unnecessary overhead.

F. Sina A Share

The Sina A Share API is a powerful tool for retrieving both real-time stock data and historical K-Line (candlestick) data. Its accessibility, free of request limits or license requirements, makes it highly practical for lightweight applications like prototypes. Below, we introduce the API request format and provide detailed examples of its usage.

1) Real-Time Stock Data: By using the real-time API, you can obtain the latest stock prices, trading volume, and other information such as price fluctuation. The Sina A Share API uses a simple HTTP GET method to retrieve stock data. The general request format is http://hq.sinajs.cn/list=[stock_code]. To ensure proper access, the API requires the inclusion of a Referer header with the value https://finance.sina.com.cn/. Without this header, the server will respond with a “403 Forbidden” error, denying access to the requested data. By adding the Referer header, the API responds successfully with the stock data. To fetch real-time trading data for the stock “Guizhou Maotai” (sh600519), the following request is sent: http://hq.sinajs.cn/list=sh600519. The response returned by the API is a plain text string containing all relevant stock data, as shown below:



```
var hq_str_sh600519="貴州茅
台,1761.000,1763.000,1769.040,1773.880,1758.000,1769.040,1769.200,1451976,2562905659.000
,174,1769.
```


This response string is compact yet highly informative. Each value corresponds to specific trading details. To help interpret this data, the Table 3 below maps key fields in the response to their meanings:

Field	Description	Example Value
Stock Name	The full name of the stock	貴州茅台
Opening Price	The stock’s price at the start of the trading day	1761.000

Previous Close Price	The closing price of the previous trading day	1763.000
Current Price	The stock's current trading price	1769.040
Highest Price	The highest price during the trading day	1773.880
Lowest Price	The lowest price during the trading day	1758.000
Buy 1 Price/Volume	The top bid price and corresponding volume	1769.040 / 174
Sell 1 Price/Volume	The top ask price and corresponding volume	1769.200 / 100
Trading Volume	The number of shares traded during the session	1,451,976
Total Transaction Value	The total value of all traded during the session	2,562,905,659.00
Date & Time	The timestamp of the returned data	2023-11-28, 15:00:00

Table 3. Sina A-share API Specification

2) Historical K-Line Data: By using the minute-level API, you can access minute-by-minute trading data, including opening price, closing price, highest price, and lowest price. The Sina A Share API provides robust support for retrieving historical stock data through candlestick charts, commonly referred to as K-Line data. These charts visually represent stock performance over various timeframes, including minute-level, daily, weekly, and monthly intervals. This feature is essential for analyzing trends, identifying patterns, and making informed trading decisions. The API endpoint for retrieving raw historical K-Line data is as follows: http://money.finance.sina.com.cn/quotes_service/api/json_v2.php/CN_MarketData.getKLineData. This endpoint requires specific query parameters to define the scope of the data:

- symbol: The stock code (e.g., sh600519 for Guizhou Maotai).
- scale: The time interval for the data (e.g., 1 for minute-level, 240 for daily).
- ma: Whether to include moving averages (no for none).
- datalen: The number of data points to retrieve (e.g., 1023 for the last 1023 records).

3) K-Line Chart Images: In addition to raw data, the Sina API simplifies K-Line chart generation by providing direct access to pre-rendered chart images for daily, weekly, monthly, and minute-level intervals. These images can be embedded directly into the application without requiring additional charting libraries, significantly reducing development complexity. In our application, the K-Line Chart Images provided by the Sina A Share API are the primary visual representation of stock performance trends. These pre-rendered charts allow for a lightweight and efficient way to display stock data across different time frames directly within the StockDetailView. They include essential trading insights such as daily price movements, candlestick data, and other key metrics.

The StockDetailView serves as a comprehensive hub for users to view detailed stock information. By integrating the K-Line Chart Images, the view provides visual insights into stock performance through multiple timeframes: intraday (minute-level), daily, weekly, and monthly candlestick charts. These charts are seamlessly displayed within the UI under sections labeled as "Intraday", "Daily K-Line", "Weekly K-Line", and "Monthly K-Line".

For example, when users query the stock 601318 (Ping An Insurance), the system retrieves and displays the following data:

- Intraday Chart:

  Displays minute-level price changes throughout the current trading day, fetched from http://image.sinajs.cn/newchart/min/n/sh601318.gif The result is the following Figure 20.

Figure 20. Intraday Chart

- Daily K-Line Chart:

  Displays daily candlestick data for the stock, fetched from http://image.sinajs.cn/newchart/daily/n/sh601318.gif The result is the following Figure 21.

Figure 21. Daily K-Line Chart

- Weekly K-Line Chart:

  Displays daily candlestick data for the stock, fetched from http://image.sinajs.cn/newchart/weekly/n/601318.gif The result is the following Figure 22.

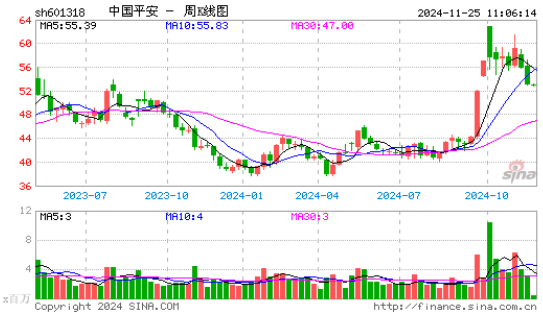


Figure 22. Weekly K-Line Chart

- Monthly K-Line Chart:

  Illustrate broader performance trends over monthly intervals, fetched from http://image.sinajs.cn/newchart/monthly/n/601318.gif The result is the following Figure 23.

Figure 23. Monthly K-Line Chart

The integration of K-Line Charts from the Sina A-Share API enables our mobile app's StockDetailView to provide users with a highly detailed and visually intuitive interface. As shown in the figure, the view incorporates comprehensive tools for analyzing stock performance across various timeframes, along with real-time trading data.

In the StockDetailView, users can seamlessly toggle between different K-Line charts, such as Intraday, Daily, Weekly, and Monthly, using the provided tabs. Each tab dynamically fetches the corresponding K-Line chart through API calls to Sina's servers. For instance:

- Intraday Chart reflects real-time minute-level price movements.
- Daily K-Line captures daily candlestick data for analyzing short-term trends.
- Weekly and Monthly K-Lines provide a broader perspective on stock performance over longer periods.

These charts ensure that users can switch views effortlessly, as shown in Figure 24, empowering them to make informed decisions based on detailed market trends. The backend system is configured to fetch and update the K-Line chart images dynamically, ensuring that the data displayed is always current and accurate.

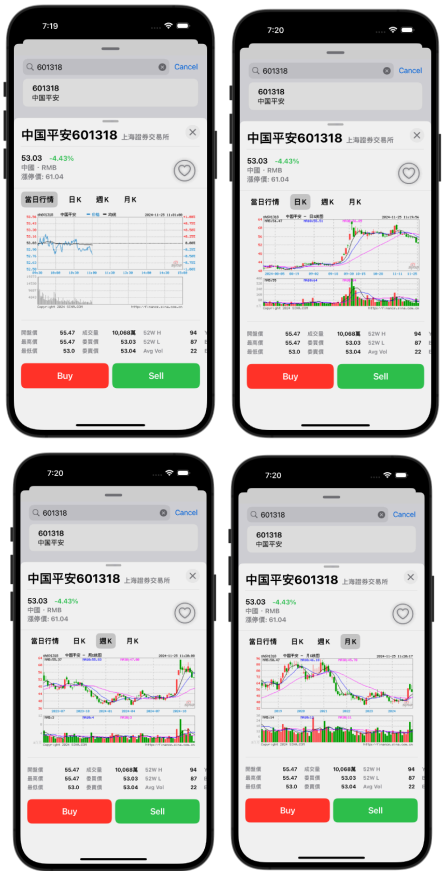


Figure 24. Charts in Stock Detail View

The architecture implemented in our StockDetailView closely follows modern financial trading systems. By integrating real-time API calls and pre-rendered K-Line chart images, we successfully simulate the core functionalities of a professional stock trading platform. Users benefit from the same level of clarity and interactivity available in commercial applications, making this prototype a strong foundation for further development in financial technology applications.

IV. API Function Call

The Webservice class is the foundation of API integration in our application, designed to handle network requests, parse responses, and provide structured data to the ViewModel layer. This implementation aligns with the MVVM (Model-View-ViewModel) architecture, ensuring a clean separation between data processing and user interface logic. Leveraging modern Swift features such as async/await and URLSession, the Webservice class achieves asynchronous, efficient, and highly maintainable API handling.

For example, the getStocks function demonstrates how to fetch a list of stocks from Mairui's API. It dynamically constructs the URL using URLComponents to ensure flexibility and avoid hardcoded strings. As shown in the code snippet below, the function uses a predefined array of API keys to distribute requests across multiple keys, mitigating the risk of exceeding API limits. As the following code snippet

shows, we have totally three backup API keys and our program will randomly pick one to authorize the API function call each time. This can lead to fault tolerance and also balance the workload among different API keys.

```
func getStocks() async throws -> [Stock] {
    var components = URLComponents()
    let apiKeys: [String] = [
    let apiKey = apiKeys.randomElement()!
    components.scheme = "https"
    components.host = "api.mairui.club"
    components.path = "/hslt/list/\(apiKey)"
```

The API response is then fetched using URLSession.shared.data(from:), ensuring an asynchronous and non-blocking execution. The response is validated for an HTTP status code of 200 before proceeding with decoding, as seen here:

```
let (data, response) = try await URLSession.shared.data(from: url)
// 打印API返回的資料
  if let dataString = String(data: data, encoding: .utf8) {
      print(dataString)
  }
guard (response as? HTTPURLResponse)?.statusCode == 200 else {
    throw NetworkError.badID
}

do {
    let stock_Response = try JSONDecoder().decode([Stock].self, from: data)
      print(stock_Response)
      return stock_Response.stocks // 如果解碼成功，直接返回股票數組
    return stock_Response
} catch {
    print("解碼錯誤: \(error)") // 打印錯誤信息
    return [] // 解碼失敗時返回一個空數組
}
```

Once the data is validated, it is decoded into an array of Stock objects using Swift's JSONDecoder. Any decoding errors are caught and logged, ensuring robust error handling:

```
do {
    let stock_Response = try JSONDecoder().decode([Stock].self, from: data)
      print(stock_Response)
      return stock_Response.stocks // 如果解碼成功，直接返回股票數組
    return stock_Response
} catch {
    print("解碼錯誤: \(error)") // 打印錯誤信息
    return [] // 解碼失敗時返回一個空數組
}
```

Similarly, the getSinaStockInfo function exemplifies how to handle more complex API requirements, such as setting HTTP headers and decoding data encoded in non-standard formats. To access the Sina API, the function appends a Referer header to comply with the API's restrictions:

```
func getSinaStockInfo(stockExchange: String="sh", stockId: String="600519") async throws -> [SinaStockInfo] {
    var components = URLComponents()
    components.scheme = "https"
    components.host = "hq.sinajs.cn"
    components.path = "/list=\(stockExchange)\(stockId)"

    guard let url = components.url else {
        throw NetworkError.badURL
    }

    var request = URLRequest(url: url)
    request.setValue("https://finance.sina.com.cn/", forHTTPHeaderField: "Referer")
```

Additionally, the response data is decoded using the gb18030 encoding, tailored for Chinese-language content. Helper methods such as extractStockData and parseStockInfo are used to further process the response, transforming raw strings into structured objects:

```
guard let dataString = String(data: data, encoding: .gb18030) else {
    print(2)
    throw NetworkError.badID
}
//print("dataString: \(dataString)")

do {
    if let stockData = extractStockData(from: dataString) {
        //print(stockData)
        if let sinaStockInfo = parseStockInfo(from: stockData) {
            print(sinaStockInfo)
            return [sinaStockInfo]
        } else {
            print("Failed to parse stock information.")
            return []
        }
    } else {
        print("Failed to extract stock data.")
        return []
    }
} catch {
    print("解碼錯誤: \(error)") // 打印錯誤信息
    return [] // 解碼失敗時返回一個空數組
}
```

By modularizing the decoding logic, the Webservice class ensures flexibility and maintainability. These helper functions are designed to handle edge cases such as missing or malformed data gracefully, as demonstrated below:

```
func parseStockInfo(from string: String) -> SinaStockInfo? {
    // 去掉最后的逗号
    let cleanedString = string.hasSuffix(",") ? String(string.dropLast()) : string
    let components = cleanedString.components(separatedBy: ",")
    // 加入檢查段，不然模擬器會崩潰
    guard components.count <= 34 else {
        print("Expected less and equal to 34 components, but found \(components.count)")
        return nil
    }

    return SinaStockInfo(
        name: components[0],
        openPrice: Double(components[1]) ?? 0.0,
        previousClosePrice: Double(components[2]) ?? 0.0,
        currentPrice: Double(components[3]) ?? 0.0,
        highestPrice: Double(components[4]) ?? 0.0,
        lowestPrice: Double(components[5]) ?? 0.0,
        bidPrice: Double(components[6]) ?? 0.0,
        askPrice: Double(components[7]) ?? 0.0,
        volume: Int(components[8]) ?? 0,
        turnover: Double(components[9]) ?? 0.0,
        bid1Volume: Int(components[10]) ?? 0,
        bid1Price: Double(components[11]) ?? 0.0,
        bid2Volume: Int(components[12]) ?? 0,
        bid2Price: Double(components[13]) ?? 0.0,
        bid3Volume: Int(components[14]) ?? 0,
        bid3Price: Double(components[15]) ?? 0.0,
        bid4Volume: Int(components[16]) ?? 0,
        bid4Price: Double(components[17]) ?? 0.0,
        bid5Volume: Int(components[18]) ?? 0,
        bid5Price: Double(components[19]) ?? 0.0,
        ask1Volume: Int(components[20]) ?? 0,
        ask1Price: Double(components[21]) ?? 0.0,
        ask2Volume: Int(components[22]) ?? 0,
        ask2Price: Double(components[23]) ?? 0.0,
        ask3Volume: Int(components[24]) ?? 0,
        ask3Price: Double(components[25]) ?? 0.0,
        ask4Volume: Int(components[26]) ?? 0,
        ask4Price: Double(components[27]) ?? 0.0,
        ask5Volume: Int(components[28]) ?? 0,
        ask5Price: Double(components[29]) ?? 0.0,
        date: components[30],
        time: components[31]
    )
}
```

```
func extractStockData(from response: String) -> String? {
    guard let range = response.range(of: "=") else {
        return nil
    }
    let extractedString = String(response[range.upperBound...])
    let trimmedString = extractedString.trimmingCharacters(in: .whitespacesAndNewlines)
    // 去掉最后的分号
    if trimmedString.hasSuffix(";") {
        return String(trimmedString.dropLast())
    }
    return trimmedString
}
```

The Webservice class also plays a pivotal role in integrating these API calls into the broader application architecture. For

instance, getStocks is invoked within the StockListViewModel, allowing the ViewModel to fetch and process stock data asynchronously. This design not only abstracts away the complexities of API integration but also ensures that the user interface remains responsive by performing all network operations in the background. By centralizing API logic within the Webservice class, the application achieves a robust and scalable structure, facilitating future enhancements such as adding new endpoints or adjusting existing ones with minimal code duplication.

In conclusion, the Webservice class is a critical component of the application's architecture, handling API requests and response parsing with precision. Through its modular design, error handling, and integration with SwiftUI's ViewModel layer, it ensures that the application delivers a seamless and reliable user experience.

```swift
@MainActor
class StockListViewModel: ObservableObject {

    @Published var stocks: [StockViewModel] = []

    func getStockList() async {
        do {
            if let documentsDirectory = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).first {
                // 创建指向特定文件的 URL
                let fileURL = documentsDirectory.appendingPathComponent("stock_list.json")
                // 现在您可以使用 fileURL 来读取或写入该文件
                if FileManager.default.fileExists(atPath: fileURL.path) {
                    do {
                        print("股票列表文件存在於: \(fileURL)")
                        let data = try Data(contentsOf: fileURL)
                        let decoder = JSONDecoder()
                        let stocks = try decoder.decode([Stock].self, from: data)
                        self.stocks = stocks.map(StockViewModel.init)
                        // print(stocks)
                    } catch {
                        print("无法读取文件: \(error)")
                    }
                } else {
                    print("文件不存在，需要从 API 请求数据")
                    // 执行 API 请求...
                    let stocks = try await Webservice().getStocks()
                    self.stocks = stocks.map(StockViewModel.init)
                    // print(stocks)
                    print(type(of: stocks))
                    print(type(of: self.stocks))

                    // Save it to local json file
                    do {
                        let data = try JSONEncoder().encode(stocks)
                        if var url = FileManager.default.urls(for: .documentDirectory, in: .userDomainMask).first {
                            url.appendPathComponent("stock_list.json")
                            print("Stock List is saved at: \(url)")
                            try data.write(to: url)
                        }
                    } catch {
                        print(error)
                    }
                }
            } else {
                print("无法找到 Documents 目录")
            }
```

V. QUEUE PURCHASE

Queue Purchase and Auto Buy are the main contributions of this paper trading app. We have a special investing strategy requiring the Queue Purchase and Auto Buy mechanism. Briefly to say, we enable users to add the desired stocks into Queue waiting for transaction. Once the stock achieves the target price, such as Limit up, the Auto Buy will be triggered and build the transaction.

The following Figure 25 is the Queue Purchase View. In the input box, users can key in the desired stock using StockId or StockName just like the Search View, the corresponding stock will be listed below let users select. Second, users are allowed to enter the number of shares they want to add into the queue.

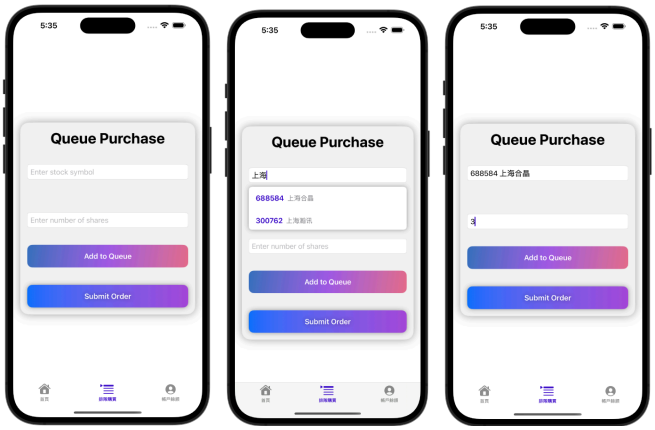


Figure 25. Queue Purchase View

When every input value is completed, users can hit the "Add to Queue" button to add selected stock into the queue. In the Queue, it also shows the current price and Limit Up for corresponding stocks. Users are also able to cancel specific stocks in the queue if they regret it by clicking the "Cancel" caption.

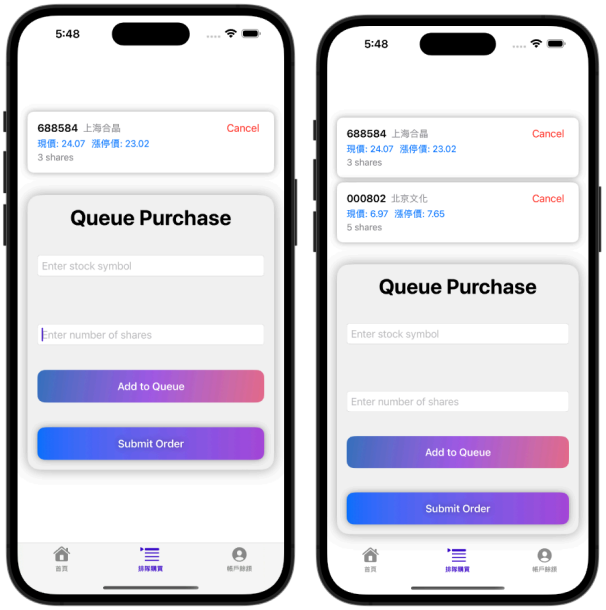


Figure 26. Add Stock to Queue Purchase

Once the queue is occupied, our backend will check if the criteria matched for every 10 seconds. If the stock price satisfied our criteria, it will automatically send out the order and update the transaction.

G. *Auto Buy Mechanism*

The Queue Purchase system includes an automated backend process that evaluates the queued stocks against predefined criteria every 10 seconds. The following code shows our fetchCurrentPrice function plays a pivotal role in ensuring that real-time stock data is consistently retrieved and evaluated. By utilizing the Sina A Share API, this function retrieves the stock's current price (currentPrice) and calculates its limit-up price

(limitUpPrice) based on the previous close price. The calculation follows a standard formula: $Limit\ -\ Up\ Price\ =\ Previous\ Close\ Price \times 1.1$

```
if let randomItem = queueList.randomElement() {
    fetchCurrentPrice(for: randomItem.stock) { currentPrice, limitUpPrice in
        if let currentPrice = currentPrice, let limitUpPrice = limitUpPrice {
            let difference = limitUpPrice - currentPrice
            if difference > 0 && difference <= 100 {
                // 呼叫 BuyPlaceOrderHelper.placeOrder
                let success = BuyPlaceOrderHelper.placeOrder(
                    stockID: randomItem.stock.stockId,
                    stockName: randomItem.stock.name,
                    stockExchange: randomItem.stock.exchange,
                    numberOfShares: randomItem.quantity,
                    stockPrice: currentPrice,
                    context: viewContext // 引用 Core Data 的上下文
                )

                if success {
                    // 發送成功通知
                    let content = UNMutableNotificationContent()
                    content.title = "自动下单成功"
                    content.body = "股票 \(randomItem.stock.stockId) (\(randomItem.stock.name)) 已成功下单。"
                    content.sound = .default

                    let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger:
                    UNUserNotificationCenter.current().add(request)

                    // 移除該股票
                    removeFromQueue(stock: randomItem.stock)
                } else {
                    // 發送失敗通知
                    let content = UNMutableNotificationContent()
                    content.title = "下单失败"
                    content.body = "股票 \(randomItem.stock.stockId) (\(randomItem.stock.name)) 下单失败，可能余额不足
                    content.sound = .default

                    let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger:
                    UNUserNotificationCenter.current().add(request)
                }
            }
        }
    }
}
```

Our investment strategy is centered on a simple but effective principle: when the stock price exceeds its Limit Up price by 5%, the system triggers an automatic purchase. The following steps occur during the execution:

1. Price Evaluation: The backend system continuously checks the real-time price of each stock in the queue using the Sina A Share API. If the current price meets the criteria (Limit Up + 5%), the system proceeds to the next step.

```
private func handleSubmitOrder() {
    if !queueList.isEmpty {
        timer?.invalidate()
        timer = Timer.scheduledTimer(withTimeInterval: 10.0, repeats: true) { _ in
            if let randomItem = queueList.randomElement() {
                fetchCurrentPrice(for: randomItem.stock) { currentPrice, limitUpPrice in
                    if let currentPrice = currentPrice, let limitUpPrice = limitUpPrice {
                        let difference = limitUpPrice - currentPrice
                        if currentPrice > difference * 1.05 && difference > 0 && difference <= 100 {
                            // 呼叫 BuyPlaceOrderHelper.placeOrder
                            let success = BuyPlaceOrderHelper.placeOrder(
                                stockID: randomItem.stock.stockId,
                                stockName: randomItem.stock.name,
                                stockExchange: randomItem.stock.exchange,
                                numberOfShares: randomItem.quantity,
                                stockPrice: currentPrice,
                                context: viewContext // 引用 Core Data 的上下文
                            )
```

2. Balance Verification: Before executing the purchase, the system checks the user's account balance to ensure there are sufficient funds to complete the transaction.

- If the balance is insufficient, a notification is sent to the user, indicating that the order cannot be completed due to a lack of funds. Figures 27 and 28 illustrate the outcomes of an insufficient balance and a successful transaction, respectively. This notification system ensures that users are promptly informed about their transaction status, preventing unexpected failures during order execution. By integrating real-time balance verification, the application enhances user experience and financial awareness, allowing traders to make informed decisions. Furthermore, this mechanism mimics real-world trading platforms, reinforcing financial discipline and simulating the importance of maintaining sufficient funds before placing an order.

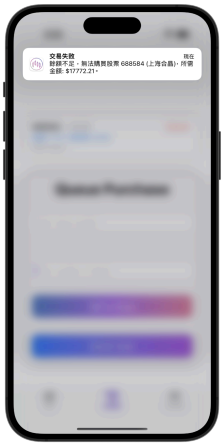


Figure 27. Insufficient Funds notification

- If the balance is sufficient, the system proceeds with the transaction, ensuring a seamless and efficient trading experience. This process involves verifying the user's available funds before executing the order, preventing transaction failures due to insufficient balance. Once the purchase is successfully completed, a notification is sent to the user confirming the order execution, as shown in Figure 28. This feature enhances user confidence by providing immediate feedback on their trading activities. Additionally, it reinforces the realism of the paper trading system by simulating real-world financial transactions, where order execution depends on fund availability. By integrating this notification mechanism, the application improves transparency and user engagement, allowing traders to monitor their transactions in real-time.

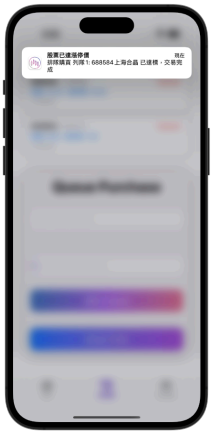


Figure 28. Sufficient Funds Notification

3. Transaction Execution: Upon successful verification, the system automatically executes the order. The purchased stock is added to the user's portfolio, and the account balance is updated to reflect the deduction. Simultaneously, a "Purchase Successful" notification is sent to the user, confirming the transaction.

H. *User Notification*

The notification system in this project is built using the **UserNotifications** framework, providing real-time feedback essential for the **Queue Purchase** mechanism. This functionality ensures users are informed about the success or failure of their transactions, enhancing the app's transparency and usability.

When the application starts, the system requests permission to send notifications via **UNUserNotificationCenter**. The following code demonstrates this process:

```
private func requestNotificationPermission() {
    UNUserNotificationCenter.current().requestAuthorization(options: [.alert, .sound, .badge]) { granted, error in
        if granted {
            print("Notification permission granted.")
        } else if let error = error {
            print("Failed to request notification permission: \(error)")
        } else {
            print("Notification permission denied.")
        }
    }
}
```

This ensures compliance with iOS privacy requirements by explicitly asking users for notification permissions.

The main logic for triggering notifications is implemented in the handleSubmitOrder method, where the system monitors the stock queue and verifies whether the target price is reached. If the stock price meets the criteria (e.g., exceeding the limit-up price), the system attempts a transaction. If the transaction succeeds, a notification confirms the purchase. If the transaction fails due to insufficient funds, another notification informs the user about the failure. This is demonstrated in the following code snippet:

```
if success {
    // 發送成功通知
    let content = UNMutableNotificationContent()
    content.title = "自动下单成功"
    content.body = "股票 \(randomItem.stock.stockId) (\(randomItem.stock.name)) 已成功下单。"
    content.sound = .default

    let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: nil)
    UNUserNotificationCenter.current().add(request)

    // 移除該股票
    removeFromQueue(stock: randomItem.stock)
} else {
    // 發送失敗通知
    let content = UNMutableNotificationContent()
    content.title = "下单失败"
    content.body = "股票 \(randomItem.stock.stockId) (\(randomItem.stock.name)) 下单失败，可能余额不足。"
    content.sound = .default

    let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: nil)
    UNUserNotificationCenter.current().add(request)
}
```

The UNMutableNotificationContent object is used to dynamically generate the notification's title, body, and sound. Each notification is uniquely identified using UUID().uuidString to prevent conflicts, and it is scheduled for immediate delivery using UNNotificationRequest.

To handle notifications when the app is in the foreground, a custom NotificationDelegate is implemented:

```
class NotificationDelegate: NSObject, UNUserNotificationCenterDelegate {
    static let shared = NotificationDelegate()

    private override init() {
        super.init()
    }

    func userNotificationCenter(_ center: UNUserNotificationCenter
                                , willPresent notification: UNNotification
                                , withCompletionHandler completionHandler: @escaping (UNNotificationPresentationOptions) -> Void) {
        completionHandler([.banner, .sound])
    }
}
```

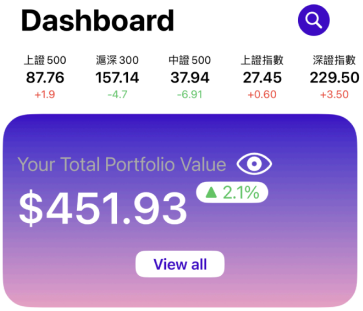
This ensures notifications are displayed as banners and play sounds even when the app is active, providing a seamless user experience.

The notification system integrates tightly with the stock monitoring and transaction logic. For example, when a queued stock is checked for its current price, the fetchCurrentPrice function fetches real-time data from the backend. If the price meets the target, the system processes the transaction and sends the appropriate notification. This integration guarantees that every important transaction update is communicated to the user promptly and effectively. The notifications' real-time and dynamic nature align with modern app standards, enhancing both the user experience and system interactivity.

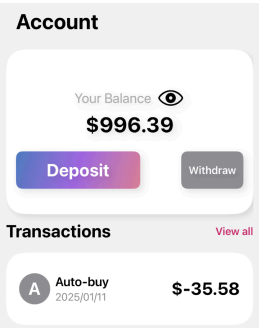
I. *Post-Transaction Update*

After a successful transaction, the following updates occur within the app:

- The **Portfolio** view is refreshed to include the newly purchased stock, along with updated inventory values and portfolio metrics.



- The **Account Balance** view reflects the deducted amount in the transaction history, categorized as an Auto-Buy, with the corresponding stock ID, transaction amount, and date displayed.



Once all the stocks in the queue have been successfully processed and purchased, the system clears the queue to ensure it is ready for new transactions. This functionality is implemented by iterating through the queueList state variable,

where each stock item is dynamically removed after its corresponding transaction is executed. Once the queue is emptied, the queueList is programmatically reset to an empty array as shown in Figure 29. This update is reflected in the user interface, ensuring consistency between the backend logic and the app's visual state. At the same time, a notification is dispatched to inform the user that the queue has been cleared, providing feedback that all queued transactions have been successfully completed. This seamless integration of backend operations and user interface updates ensures clarity and usability in managing queued stock purchases.

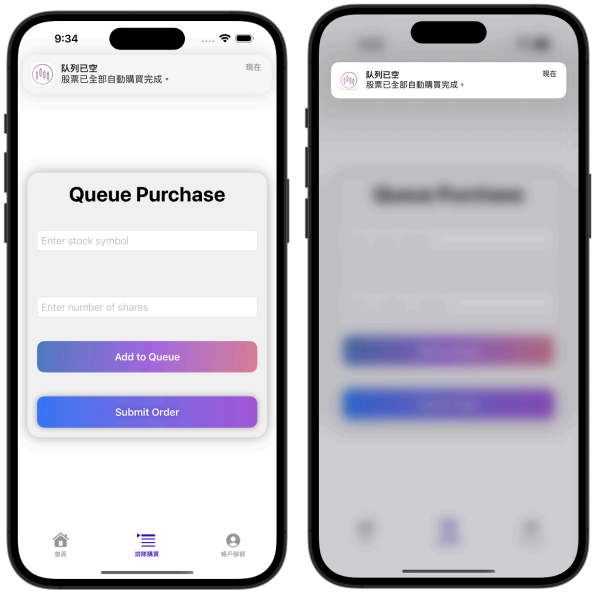


Figure 29. Queue List Emptied Post-Transaction

The core functionality of clearing the queue and sending notifications is implemented in the handleSubmitOrder method, as shown in the provided code snippet. This method employs a Timer to periodically iterate over the queueList, checking if the conditions for a transaction are met. For each stock in the queue, the fetchCurrentPrice function retrieves the current price and the calculated limit-up price. If the current price satisfies the condition (e.g., exceeding the limit-up price by a certain threshold), the system proceeds to execute the purchase using the BuyPlaceOrderHelper.placeOrder function. This function also verifies the user's account balance to ensure sufficient funds are available.

When a transaction is successful, the stock is removed from the queue by invoking the removeFromQueue method, which updates the queueList state and triggers a UI refresh to reflect the change. Additionally, a success notification is sent to the user using the UNMutableNotificationContent class, which constructs the notification message. This notification is delivered via UNUserNotificationCenter, ensuring that the user is immediately informed about the completed transaction.

If the queueList becomes empty, the timer is invalidated, halting further price checks. At this point, a final notification is dispatched to inform the user that the queue has been cleared, as depicted in the screenshots. This integration of logic and notifications ensures a smooth user experience and consistent system behavior. The provided screenshots illustrate the notification sent to the user when the queue is cleared, confirming the successful execution of all transactions.

```
            if queueList.isEmpty {
                timer?.invalidate()
                timer = nil
                let content = UNMutableNotificationContent()
                content.title = "队列已空"
                content.body = "股票已全部自動購買完成。"
                content.sound = .default

                let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: nil)
                UNUserNotificationCenter.current().add(request)
            }
        }
    } else {
        timer?.invalidate()
        timer = nil
        let content = UNMutableNotificationContent()
        content.title = "队列为空"
        content.body = "目前列队中无股票"
        content.sound = .default

        let request = UNNotificationRequest(identifier: UUID().uuidString, content: content, trigger: nil)
        UNUserNotificationCenter.current().add(request)
    }
}
```

VI. Conclusion

This project represents a significant contribution to the field of financial technology by developing a comprehensive mobile prototype for stock paper trading, combining cutting-edge software frameworks and robust data integration. Leveraging APIs from Sina and Mairui, the application seamlessly integrates real-time and historical financial data, enabling users to engage with accurate, dynamic market information. This integration, facilitated by the SwiftUI framework, ensures a smooth and responsive user experience while maintaining an intuitive interface. Key features include portfolio management, detailed stock views with interactive K-Line charts, and account balance tracking, all designed to replicate essential functionalities found in professional trading platforms.

One of the standout features of this application is the Queue Purchase functionality, which introduces a layer of algorithmic trading simulation typically absent in traditional paper trading systems. By enabling users to set predefined conditions for automatic transactions, such as the trigger of limit-up prices, this feature bridges the gap between manual trading simulations and automated trading strategies. This simulation allows users to experiment with algorithmic trading concepts, offering educational value while mimicking real-world trading systems. Additionally, the application sends timely notifications to users when transactions are successfully executed, further enhancing user engagement and providing transparency throughout the trading process.

To ensure data integrity and reduce reliance on API calls, the application employs Core Data for local storage. This design

decision not only minimizes redundant API requests but also guarantees the persistence of critical user data, such as stock lists, portfolio holdings, and transaction histories. By storing this information locally, the application remains functional even in offline scenarios, thereby improving user reliability and reducing latency during data retrieval. Core Data's integration showcases the potential for mobile applications to handle large-scale data efficiently without compromising performance or user experience.

The modular design of the system further underscores its adaptability and scalability. By encapsulating API calls within the Webservice class and separating concerns through the MVVM (Model-View-ViewModel) architecture, the application provides a strong foundation for future extensions. Potential areas for further development include incorporating additional asset classes, such as cryptocurrencies or ETFs, introducing predictive analytics for stock trends, and integrating with live trading systems for real-world application. The current architecture also supports straightforward updates, allowing for the addition of new data sources, enhanced visualization tools, and advanced algorithmic strategies without requiring significant modifications to the core system.

Moreover, the use of SwiftUI demonstrates the practicality of building modern financial tools on mobile platforms. Its declarative syntax and built-in support for responsive layouts make it an ideal choice for creating an engaging and visually appealing user interface. The application's ability to dynamically display detailed stock data, such as K-Line charts and price movements, highlights the versatility of SwiftUI in handling complex financial visualizations. By incorporating these interactive features, the project not only meets user expectations for modern trading platforms but also sets a benchmark for future mobile financial applications.

In summary, this project successfully merges advanced financial concepts with mobile technology, delivering a sophisticated and practical tool for stock paper trading. By addressing the challenges of data integration, offline functionality, and algorithmic trading simulation, the application provides a valuable resource for both novice and experienced users. Its flexible architecture and robust design offer a strong foundation for further innovation, positioning this work as a stepping stone toward the development of more advanced financial technologies. Ultimately, this project underscores the potential of mobile platforms to revolutionize the accessibility and functionality of trading tools, paving the way for future advancements in the field.

References

This project draws on a variety of resources to inform its design and implementation. The MyData API documentation [1] serves as a crucial foundation, providing detailed information on how to retrieve essential stock market data, including comprehensive stock lists and trading information. This resource has enabled the seamless integration of real-time stock tracking and data retrieval capabilities into the application.

Further insights were gained from the Juhe Data platform [2], which offers additional context for implementing external APIs in financial applications. These resources collectively informed the development of robust data-fetching mechanisms and enhanced the application’s ability to provide accurate, up-to-date information.

Unpublished studies on stock management applications [3, 4] were also instrumental in shaping the project. These works explored key functionalities such as AutoBuy and real-time monitoring, which served as a conceptual basis for implementing the Queue Purchase feature. By incorporating these advanced features, the application bridges the gap between conventional paper trading systems and modern automated trading strategies.

Inspiration was also drawn from open-source repositories of paper trading simulators and stock management systems [5, 6]. These projects provided practical examples of modular design and architectural flexibility, which were adapted to suit the specific requirements of this application. The integration of these principles ensures scalability and lays the groundwork for future enhancements, such as cryptocurrency trading and real-world financial system integration.

Additionally, instructional resources from iOS development courses, particularly those focusing on SwiftUI [7], played a significant role in shaping the application’s user interface and functionality. These materials guided the implementation of a responsive, user-friendly design, which aligns with current best practices in mobile application development.

By synthesizing technical documentation, academic research, and practical development guides, this project achieves a comprehensive and innovative approach to building a financial technology prototype.

[1] MyData API documentation - https://www.mairui.club/hsdata.html

[2] Juhe Data - https://www.juhe.cn/news/index/id/7854

[3] Wangfang Li, “Android Stock Management Application – AutoBuy and line function implementation,” unpublished.

[4] Shiyi Zhang, “Android Stock Management Application – Monitor and Waiting in Line Functions, ”unpublished

[5] SpaceMonkeyClan - Paper Trading Stock Simulator - https://github.com/SpaceMonkeyClan/Paper-Trading-Stock-Simulator

[6] dkhamsing - Stock Simulator - https://github.com/dkhamsing/stocks

[7] Stanford CS193P - Developing Apps for iOS - https://cs193p.sites.stanford.edu/