ChatGPT

投資模擬交易平台開發計畫

本計畫書將逐步說明如何建立一個**股票模擬交易 App**，包括前後端架構設計、yfinance 股價資料串接，以及下單交易、損益計算、庫存管理等核心功能的實作步驟。本專案預計採用 Python Flask 作為後端，Swift(iOS)作為前端介面，並使用 Supabase 雲端資料庫與排程服務。請依照以下步驟模組化開發：

系統架構概述

- **前端 (iOS App)**：使用 Swift（可採用 SwiftUI）開發，用戶介面包含註冊/登入、首頁總覽、股票查詢、交易下單、持倉與損益顯示、交易紀錄、排行榜及AI聊天建議等視圖。前端不直接抓取 Yahoo 資料，而是透過呼叫後端 API 獲取即時股價與交易操作結果。
- **後端 (Flask API 服務)**：使用 Python Flask 搭建 RESTful API 伺服器，處理用戶認證、與資料庫交互、串接 yfinance 獲取市場數據、執行模擬交易邏輯（下單、計算損益、更新持倉）、以及提供排行榜計算等。部分較複雜的運算和資料處理會在後端進行，減輕前端負擔。
- **資料庫 (Supabase/PostgreSQL)**：使用 Supabase 提供的托管 PostgreSQL 資料庫儲存應用資料。資料表包含用戶帳戶、持倉、交易記錄、股票清單及績效記錄等。透過 Supabase 的 API 或資料庫用戶端庫，讓 Flask 後端讀寫資料。Supabase 也將用於定時執行排程任務（例如定期計算排行榜）。
- **股價資料來源 (yfinance)**：利用 yfinance Python 套件從 Yahoo Finance 獲取股票市場數據。後端透過 yfinance 提供的 API 獲取即時行情（有輕微延遲）及歷史價格，用於模擬交易和顯示走勢圖。yfinance 支援 WebSocket/AsyncWebSocket，可用於實現即時資料串流（如需頻繁更新時）。
- **第三方服務**：簡訊驗證服務（如 Twilio 或本地SMS API）用於手機號碼驗證；可能的AI分析服務（未來可整合聊天機器人模型產生交易建議）；Supabase Edge Functions 或 pg_cron 用於排程任務。

上述元件各司其職、前後端透過清晰的 API 介面互動，組成完整的模擬交易平台。

接下來，我們按照開發流程逐步闡述各個模組的實作步驟與細節。

開發步驟規劃

以下將前後端交錯分解開發步驟，按照功能模組逐一說明，確保架構清晰且可逐步完成整體系統。

第1步：開發環境設置與專案初始化

- **後端環境**：安裝並設定 Python 3 開發環境。建立一個 Flask 專案目錄，例如 `invest_simulator_backend/`。建議使用虛擬環境（venv）管理套件。透過 `pip install flask supabase py-postgresql yfinance pandas python-dotenv` 等指令安裝所需套件。
- **Flask**：用於建立 Web API 服務。
- **yfinance**：取得金融市場數據。
- **supabase-py**（或直接使用 `psycopg2`/SQLAlchemy）：與 Supabase 資料庫交互。
- **pandas**：資料處理（如未使用 Excel 目標清單則可選擇性安裝）。
- **python-dotenv**：載入環境變數（如 API 金鑰、資料庫連線字串等）。

- **前端環境**：設定 Xcode 專案（例如命名為 **InvestSimulatorApp**）。選擇使用 SwiftUI 架構建立 iOS App，確保 Deployment Target 符合大多數裝置。前端將需要能發送 HTTP 請求，建議使用 `URLSession` 或相關網路庫。無需特別安裝第三方庫即可使用內建 `URLSession`，如需 Chart 圖表可使用 Apple Charts 框架(iOS16+)或其他圖表庫。
- **版本控制**：初始化 Git 儲存庫（您已經有 GitHub 專案）。確保後端和前端程式碼分離到不同資料夾或倉庫模組，方便獨立開發與部署。使用有意義的 commit 訊息紀錄每步進展。

第2步：資料庫架構設計與 Supabase 串接

- **資料模型設計**：在 Supabase 上建立資料庫和資料表，定義各項資料結構。可透過 Supabase 網站的資料庫管理介面或 SQL 腳本建立。主要資料表包括：
- **Users 用戶表**：欄位包括用戶ID（主鍵）、手機號碼、姓名（或暱稱）、註冊日期、初始本金、當前可用現金餘額、總資產、累計損益、邀請碼等。
    - 初始本金預設 1,000,000 元模擬金。
    - 可用現金會隨交易變動，即總資產 = 可用現金 + 持股市值總和。
    - 邀請碼可隨機生成（例如用戶ID的哈希或UUID），用於好友註冊獎勵。
- **Positions 持倉表**：記錄每個用戶當前持有的股票部位。欄位包括用戶ID、股票代號、目前持有股數（或張數）、持倉平均成本、當前市值、所屬市場別等。
    - 持倉資訊用於計算浮動損益和顯示投資組合構成。
    - 平均成本需在多次買進時動態更新（加權平均）。
- **Transactions 交易紀錄表**：紀錄所有模擬交易明細。欄位包括交易ID、用戶ID、股票代號、交易類型（買入/賣出）、價格、數量、交易時間、手續費、交易稅、該筆交易實現損益等。
    - 每次下單（無論買賣）都插入一筆紀錄。買入的實現損益通常為0（因未賣出），賣出則根據買入成本計算損益。
    - 手續費和交易稅（賣出需課0.3%證交稅）在這裡記錄以便損益計算更精確。
- **Stocks 股票清單表**：（選擇性）儲存所有支援交易的股票代號及基本資訊（名稱、所屬市場、市值等）。這可用於驗證輸入及提供股票名稱顯示。資料可從公開來源匯入，例如透過 Yahoo 或證交所 API 獲取上市上櫃清單。也可在應用發佈前離線準備好清單。
- **PerformanceSnapshots 績效快照表**：定期儲存用戶資產數據供排行使用。欄位包括用戶ID、日期、總資產、市值、報酬率等。可每日或每週存一筆，用來計算週/月/年績效。亦可為每種周期建立單獨表或增加周期類型欄位。
- **Referrals 推廣邀請表**：（選擇性）記錄邀請關係。欄位包括邀請者用戶ID、被邀請者用戶ID、獎勵是否發放。用於避免重複獎勵。
- **Supabase 串接**：在 Flask 後端連接 Supabase 資料庫。有兩種方式：
- 使用 Supabase 提供的 Python 客戶端庫 `supabase-py`。需要提供 Supabase 專案的 URL 和 API Service Key。**在 `.env` 檔案中保存 Supabase 的連線字串或金鑰**，後端程式啟動時載入。透過客戶端可直接對資料表執行 CRUD 操作。例如：

```
from supabase import create_client, Client
supabase_url = os.getenv("SUPABASE_URL")
supabase_key = os.getenv("SUPABASE_SERVICE_KEY")
supabase: Client = create_client(supabase_url, supabase_key)
# 範例：插入一筆資料
supabase.table("Users").insert({"id": user_id, "phone":
phone, ...}).execute()
```

- 或者直接使用 Postgres 連接（例如使用 `psycopg2` 或 SQLAlchemy），透過 Supabase 提供的連接字串連到資料庫。這樣可以寫 SQL 查詢或使用 ORM 建模。在 `.env` 保存 `DATABASE_URL`（格式類似 `postgres://user:password@host:port/database`）。初期可用簡單的 SQL 語句操作資料庫，例如：

```
import psycopg2
conn = psycopg2.connect(os.getenv("DATABASE_URL"))
cur = conn.cursor()
cur.execute("SELECT * FROM Users WHERE phone=%s", (phone,))
```

- 為簡化開發，可封裝**資料庫存取服務（SupabaseService 或 DB Service）**：將所有與資料庫的CRUD邏輯集中在一個模組，例如 `db_service.py`。提供函式如 `create_user(...)`, `get_user(...)`, `update_position(...)`, `log_transaction(...)` 等，內部調用 supabase 或SQL操作，這樣在其他模組使用時更簡潔。這也是您提到的 `SupabaseService.createPosition / updatePosition` 等函式之所在。
- **資料表權限**：設定 Supabase 資料庫的存取策略。如透過後端服務存取且使用 service key，可以繞過 RLS限制。但未來若考慮直接從前端調用 Supabase API（例如查詢排行榜），需要為表設定 Row Level Security 和定義安全的Policy。當前我們假定所有資料操作經由後端受控進行。

第3步：用戶註冊與登入（手機驗證流程）

建立**會員系統**，讓用戶使用手機號碼註冊並取得模擬本金，後續每次操作需驗證身份。具體步驟：

- **手機號碼輸入**：在前端 App 上提供註冊畫面。用戶輸入手機號碼點擊註冊。前端調用後端 API（例如 `POST /auth/send_otp`）提交手機號碼。
- **發送驗證碼 OTP**：後端收到請求後，使用簡訊服務API發送6位數驗證碼到該手機號碼。可使用第三方簡訊服務供應商（如 Twilio、AsiaSMS 等）：
- 後端需要安裝對應的API套件或直接呼叫HTTP。將簡訊服務帳戶SID、Auth Token或API金鑰等敏感資訊放在 `.env`。
- 調用簡訊API的結果要處理成功或錯誤情況，將結果返回給前端（如短信已發送或發送失敗）。
- 同時在後端暫存該手機的 OTP，例如儲存在資料庫（建議有個 OTP 驗證碼表，包含手機號碼、OTP碼、過期時間），或者暫存在伺服器記憶體（開發測試時簡單存變數或快取）。為安全起見OTP應設有效期（如5分鐘）。
- **OTP驗證**：前端跳轉到輸入驗證碼畫面，供用戶填寫接收到的 OTP。然後呼叫後端 API（如 `POST /auth/verify_otp`），攜帶手機號碼和OTP碼。
- 後端校驗：比對用戶提交的OTP與伺服器儲存的是否一致且未過期。若正確：
    - 在 Users 資料表建立新用戶記錄（或如果該手機已註冊過則提示登入）。新用戶預設姓名可先使用手機尾號代稱，**可用現金設為 1,000,000** 初始金，其他欄位如註冊日期、邀請碼等填入。
    - 為新用戶生成**身份驗證憑證**：可採用 JWT (JSON Web Token) 或 session 機制。由於是行動App，建議 JWT。後端用預先設定的密鑰簽發 token，payload 至少包含用戶ID。將 token 返回給前端保存（通常存於 App 的安全儲存，例如鑰匙圈）。後續前端調用保護API時在 `Authorization` header 附帶此 token。
    - 如果有好友邀請碼，在此時處理獎勵：檢查請求中是否附帶 invite_code。若有且有效（在系統存在且未被該用戶使用過），則：

    - 查找對應邀請者，為邀請者與被邀請者雙方增加額外模擬金（可例如各加 $100,000，金額由您決定）。需要在 Users 資料表更新可用現金，同時可能紀錄在 Referrals 表避免重複使用。此處要一併插入交易紀錄（描述「邀請獎勵」類型的資金變動，以區分於交易損益）。
- OTP錯誤或逾時：返回錯誤訊息供前端提示，允許重試或重新發送。
- **登入流程**：之後用戶再次使用App時：
- 若沿用 OTP 每次登入，可直接走上述流程（輸入手機->收OTP->驗證）。為提高便利性，可以實現**記住登入**：當 JWT 未過期時，自動登入。或者允許用戶設定密碼，在首次註冊後引導設定密碼，之後可用手機+密碼透過 `POST /auth/login` 獲取 token。
- Flask 後端應提供 `/auth/login` 端點（手機號+密碼 或 手機+OTP 驗證），驗證成功發 token。由於 OTP 屬一次性，實際應用中常在首次註冊完成後要求設定密碼；如不設定密碼，也可選擇每次都OTP登入，看您偏好。
- **權限保護**：實作一個 Flask 中介函式，用於保護需登入的路由。在每次請求時驗證 Authorization header 的 JWT，確定用戶身份。未攜帶或驗證失敗則返回401未授權錯誤。這可確保交易下單、查看持倉等操作只能由合法用戶進行。

完成此步驟後，應用具備基本的會員系統。可以先行測試：註冊新用戶，資料庫中應產生用戶記錄及初始資金；嘗試帶 token 呼叫一個受保護的測試端點驗證授權功能。

第4步：即時股價資料取得（yfinance API 串接）

為了在模擬平台中提供即時行情與歷史走勢，我們需要將 Yahoo Finance 的數據接入後端：

- **yfinance 基本使用確認**：在開發環境下先進行簡單測試，確保 yfinance 可取得所需市場資料。例如，在 Python shell 中嘗試：

```
import yfinance as yf
tsmc = yf.Ticker("2330.TW")             # 2330.TW 表示台積電 (TWSE)
print(tsmc.fast_info["last_price"])   # 快速獲取最新價
print(tsmc.info.get("marketCap"))     # 基本資訊（市值等）
hist = tsmc.history(period="1mo", interval="1d")
print(hist.tail(5))                   # 最近5天的收盤價
```

  如果能順利取得數據，說明網路連線和套件正常。**注意**：Yahoo Finance 對台股代號的格式，一般上市股使用「.TW」為後綴，上櫃股使用「.TWO」。需確認不同市場股票代碼查詢的正確拼寫。例如 0050.TW（元大台灣50，上市），8299.TWO（群聯電子，上櫃）。興櫃及其他市場 Yahoo 可能沒有提供或格式不同。
- **股票代號與市場別處理**：後端應實作**股票代號標準化/市場識別**的功能，以便使用 yfinance 正確查詢：
- **本地股票代號輸入**：使用者在App中輸入時，可能習慣只輸入數字代碼或公司名稱。您需要將之轉換成 Yahoo 認可的 ticker 字串。可以採用靜態對照表或判斷邏輯。例如：
    - 先在 Stocks 表儲存每個股票的市場別，如 `2330` 對應市場 "TW"（上市）、`8299` 對應 "TWO"（上櫃）。
    - 後端提供函式 `format_ticker(symbol)`，傳入如 "2330" 輸出 "2330.TW"；或 "8299" -> "8299.TWO"。若資料庫有該代號則使用其市場欄位決定後綴；如沒有可嘗試預設上市.TW，若查詢失敗再嘗試.TWO（這種fallback策略確保不漏掉上櫃股）。
    - 若輸入的是公司名稱，則可能需要先在 Stocks 表比對名稱->代號，或調用外部API模糊搜索。初期可以簡化為要求用戶輸入正確的股票代號。

- **市場規則**：區分市場別在模擬下單時也有用。例如興櫃股票只能限價不能市價，漲跌幅限制也不同。但由於第一版僅支援上市/上櫃且市價單，我們可以先不考慮興櫃交易。仍可保留市場別資訊以備擴充。
- **封裝行情服務**：在後端建立`market_data.py`或類似模組，封裝取得行情的功能：
- 函式`get_realtime_price(symbol)`: 傳入股票代號(本地格式)，內部先透過 format_ticker 轉換，再用 yfinance 抓取當前價格。可使用`Ticker.fast_info["last_price"]`或`Ticker.history(period="1d", interval="1m")`取最後一筆。另外`Ticker.info`裡的`regularMarketPrice`也通常是即時價。考慮 yfinance 資料有幾秒延遲，但對模擬交易可接受。
- 函式`get_history(symbol, period, interval)`: 用於取得歷史走勢，例如 K 線圖資料。方便前端繪圖用。yfinance 提供 period 如 "1mo", "6mo", "1y", "max" 以及 interval 如 "1d"(日線), "1h"(60分線), "1m"(1分線) 等。可將前端請求的參數傳入使用。
- **批量下載**：如需同時查多檔股票行情（例如顯示整個持股清單的現價），yfinance 有`yf.download()`可以批量抓取。開發中可視情況使用。初期交易頻率不高時，直接單筆查詢即可。
- **快取策略**：如果某些行情資料被頻繁查詢（例如首頁反覆刷新的持股現價），可考慮在後端做短暫快取（例如用一個全域字典保存最近查詢結果和時間，10秒內重複請求直接回傳快取值），以減少對 Yahoo API 的呼叫頻率。這雖非必要但可優化效能。
- **即時數據串流**：如未來需要即時推播行情（而非輪詢），可研究 yfinance 的 WebSocket 功能或 Yahoo Finance 網頁使用的JSON端點。yfinance 提供的 AsyncWebSocket 可以訂閱實時數據，但由於 Yahoo 本身對免費數據的延遲和服務條款，這在模擬平台上僅作參考。初版我們不強制即時推送，可採用**前端定期拉取**或**用戶操作時再取**的模式：
- 例如用戶打開交易畫面時即刻調用`get_realtime_price`獲取當前價。或前端每隔60秒調用一次後端更新行情顯示（可用 Timer 或 Combine 定時器）。
- Flask 若要支援 WebSocket 需要額外的框架（如 Flask-SocketIO）。當用戶數多時也需注意效能。因此除非必要，可待未來升級再加入 socket 實時行情。

完成此步驟後，後端具備取得股票即時/歷史資料的能力。可開發簡單API測試，例如建立`GET /api/quote?symbol=2330`，返回 JSON：`{"symbol": "2330", "price": 600.5, "time": "2025-07-16T14:00:00"}`以供前端測試顯示。同樣`GET /api/history?symbol=2330&period=1mo&interval=1d`返回最近一個月的日K資料序列。這些API在下一步統一設計。

第5步：模擬交易核心邏輯（下單、庫存、損益計算）

交易引擎是整個應用的核心模組，負責根據用戶指令執行買賣操作、更新用戶的虛擬資產狀態並計算損益。此模組將在後端實現，並透過 API 供前端呼叫。以下細分其功能：

- **下單請求處理**：在 Flask 後端設計一個交易下單的端點，例如`POST /api/trade`或分成買賣各一個端點(`POST /api/buy`, `POST /api/sell`)。出於簡化，我們可以使用單一端點，根據傳入參數區分買或賣。
- 請求內容（JSON）需包含：
    - `symbol`: 股票代號（例如 "2330"）。
    - `action`: 動作類型，"buy" 或 "sell"。
    - `quantity`: 張數（整數，1張=1000股）或股數（如支援零股則用股數）。初版假設只支援整張交易，quantity 以「張」為單位。
    - `priceType`（可選）: 價格類型，如 "market" 市價單或 "limit" 限價單。初期可僅支援市價，價格直接取即時價執行；若支援限價，還需`price`欄位。
    - `price`（可選）: 當 priceType="limit" 時，用戶下單的價格。**注意**：模擬系統可立即撮合，如果當前市價達到限價條件則成交，否則可視為掛單（進階功能，下版再考慮掛單撮合機制）。

    - `orderType`（可選）: 交易類型，如 "stock"（現股）、"margin"（融資）、"short"（融券）。模擬環境下只允許現股交易，因此默認為 "stock"。這欄位預留未來擴充。
    - （由 token 可識別用戶ID，故無需在JSON明文傳 user_id）。
- 後端在接受請求後需進行**輸入驗證**：
    - 確認 symbol 有效（在股票清單表存在）。若不存在則返回錯誤提示「無此股票代號」。
    - 確認 quantity 為正數。若為負或0也返回錯誤。對於**賣出**指令，稍後還需檢查持倉是否足夠。
    - **價格檢查**：市價單不需要用戶提供價格；限價單若提供價格需為正數。可進一步限制限價價格不可高於當前漲停或低於跌停（如有漲跌幅限制，可從行情資訊中獲得當日漲跌停價）。
    - 市場別檢查：如果實施市場規則區分，當 symbol 所屬市場不允許市價單（如興櫃），且用戶下的是市價，則拒絕並提示「此股票僅能限價交易」；反之亦然。
- **行情取得**：後端根據 symbol 取得**成交價**：
- **市價單**：透過前述 `get_realtime_price(symbol)` 獲取目前市場價做為成交價。如市況停盤或查無價格，應回傳錯誤（盤後可選擇使用最後收盤價作為參考價執行交易，您的應用中提到「盤後固定價格」，可理解為如果用戶在非盤中時段下單，系統仍允許交易但以當日收盤價成交）。
- **限價單**：比較用戶輸入的價格與當前市場價：
    - **買入**：如果當前市場賣價<=限價，則以當前價成交（或直接以用戶的限價成交，簡化處理）；如果市價高於限價則視為無法成交（可以返回提示「限價低於市價，掛單未成交」或者直接拒絕）。
    - **賣出**：如果當前市場買價>=限價，則以當前價成交；否則無法成交。
    - （進階實現：也可將未成交部分掛在伺服器等待價格達標再成交，不過模擬系統可以不處理這麼複雜的撮合）。
- **交易執行**：根據動作 buy/sell 執行不同的模擬交易行為：
- **買入 (Buy)**：
    1. **計算交易金額**：成交價 * 張數 * 1000（每張股數）。
       e.g. 台積電現價 600 元，買 1 張需 600*1000 = 600,000 元。
    2. **檢查資金**：從用戶的可用現金餘額判斷是否足夠支付此次交易金額+手續費。手續費可模擬券商收費，例如千分之1.425。假設買入手續費 = 金額 * 0.001425（無證交稅）。如果餘額不足，則拒單返回「餘額不足」錯誤。
    3. **更新可用現金**：扣除交易金額與手續費。用戶可用現金 = 原可用現金 - (成交價*股數) - 手續費。
    4. **更新持倉**：在 Positions 表中增加或更新該股票：
    5. 若用戶之前沒有持有該股票，新增一筆持倉記錄，股數為此次購買數量*1000股，平均成本即為成交價（若要含手續費，可將手續費均攤計入成本，但一般手續費額度很小可忽略不計入成本，只從現金扣除即可）。
    6. 若已有持倉，則新的平均成本 =

$$(\text{原持倉成本} * \text{原持有股數}) + (\text{本次成交價} * \text{本次股數})$$

       / 新總股數。股數增加本次買入數量。調整持倉紀錄中的 cost 和 quantity。
    7. 更新持倉的市場別欄位（如果之前沒有記錄）以備後用。
    8. **記錄交易**：在 Transactions 表插入一筆買入紀錄。包括：股票代號，買入價，數量，時間戳，手續費（金額），**損益**欄位可填0因為尚未實現盈虧。orderType填"stock"。priceType填市價或限價。
    9. **返回結果**：將交易結果返回給前端，包括成交價、成交數量、新的現金餘額、持倉變化等。前端可據此更新UI。
- **賣出 (Sell)**：
    1. **持股檢查**：確認用戶當前持有該股票的張數是否>=欲賣張數。如不足，返回錯誤「持股不足」，拒絕交易。
    2. **計算交易金額**：成交價 * 張數 * 1000。同樣計算賣出手續費 = 金額 * 0.001425，另計交易稅（證券交易稅，台股賣出需繳千分之3，約0.003）。

3. **更新可用現金**：增加賣出所得現金（扣除手續費和交易稅後淨收款）。用戶可用現金 = 原可用現金 + (成交價*股數) - 手續費 - 證交稅。
4. **更新持倉**：減少該股票的持有股數。分兩種情況：
5. 若此次賣出後**持股仍有剩**：更新 Positions 表該股票的剩餘股數及**持倉成本**。持倉成本的調整可簡化處理：維持原平均成本不變（因為剩下的股仍是按原成本持有）。當然總市值會減少，但平均持倉成本不因賣出而改變（先進先出計算個別批次成本是更嚴謹的方法，但也更複雜）。模擬系統可採用平均成本法計算損益。
6. 若此次賣出**清倉（全部賣光）**：從 Positions 表移除該股票記錄，表示不再持有。
7. **計算損益**：針對此次賣出交易計算實現盈虧：
8. 採用平均成本法：盈虧 = (成交價 - 持倉平均成本) * 賣出股數 - 手續費 - 交易稅。
   例如原成本500元，現價550元賣出2張(2000股)，則未扣費毛利=(550-500)*2000=100,000*；若手續費+稅共~*0.45%*，約*550*2000*0.0045=4,950元，則淨利=100,000-4,950≈95,050元。
9. 將此盈虧數值填入交易紀錄的損益欄位。正數代表獲利，負數代表虧損。
10. 同時更新 Users 表中該用戶的**累計損益**（或可即時計算，詳見下述總資產計算部分）。
11. **記錄交易**：插入賣出紀錄，包括成交價、數量、手續費、稅、此筆盈虧等。
12. **交易分析**：觸發 AI 策略建議模組（詳見後續步驟），對此筆交易的表現進行分析。後端可在記錄完交易後，根據該用戶完整交易紀錄或近期行為產生建議文字。分析內容可以包含：買入點位、賣出點位的比較、持倉期間的漲跌幅、以及若產生虧損則可能的原因（例如是否追高殺低）等，給出中肯的建議。**商業化置入**：在建議內容中，如有合作的投資課程，可根據此用戶的狀況插入課程連結。例如虧損累累則推薦風險管理課程，獲利不佳則推薦技術分析課程等。
13. **返回結果**：包括成交詳情、當前現金餘額、損益結果、以及（如果有）AI分析建議內容。

- **總資產與損益管理**：
- **即時總資產計算**：用戶的總資產 = 可用現金 + 所有持倉市值總和。每次交易後，可用現金已更新，而持倉市值因價格波動也在變化。後端在返回交易結果時可以順便計算新的總資產與目前的浮動盈虧。浮動盈虧 = 總資產 - 初始本金 - 已實現盈虧。另一種計算方式是累計已實現盈虧存於 Users 表，再加上當前持倉浮盈即可得總盈虧。
- **損益紀錄**：交易帶來的**已實現損益**已在 Transaction 紀錄。用戶的**累計報酬率**可計算為 (目前總資產 / 初始本金 - 1)*100%。建議在 Users 表存一個 `total_profit`（累計盈虧金額）欄位，每次賣出時累加盈虧，買入不變。則 `總資產 = 初始本金 + total_profit + 持倉浮盈`。此欄位在排行時也可用。
- **手續費與稅**：手續費和交易稅的比率可以設為常數配置在後端，例如：

```
BROKER_FEE_RATE = 0.001425  # 手續費千分之1.425
TAX_RATE = 0.0030           # 證交稅千分之3（僅賣出收取）
```

  模擬時可假定手續費四捨五入至元。當交易金額很大時，務必用浮點仔細計算或使用Decimal避免誤差。
- **防呆與錯誤處理**：加強交易指令處理的健壓（fool-proofing）：
- 去除輸入代號中的空白或奇怪字元（如有中文括號）以避免找不到股票。
- 如果任何一步檢查或執行遇到問題，要確保**原子性**：例如賣出時如果扣款後在更新持倉時失敗，需補償回滾，不能讓數據不一致。使用資料庫交易 (transaction) 來確保同一請求內多筆資料表更新的原子性。
- 後端要對潛在錯誤返回適當的HTTP狀態碼和訊息。例如餘額不足:400 Bad Request附帶錯誤原因，非法參數:422 Unprocessable Entity等等。前端拿到錯誤需提示用戶重試或修正。

完成交易引擎的邏輯實裝後，可以進行基本測試：模擬一個用戶買入股票，檢查資料庫 Users 可用現金減少、Positions 新增記錄、Transactions 新增買單紀錄；再進行賣出，檢查現金增加、持倉更新或刪除、交易紀錄填入盈虧且用戶累計盈虧更新。如此反覆測試不同情境（全部賣清、部分賣出、多次買入平均成本計算、限價單成交與否等），確保交易模擬正確。

第6步：後端 API 設計與實作

有了會員系統、行情服務、交易引擎的支撐，我們可以整理定義**Flask API路由**以供前端調用。遵循 RESTful 風格，考慮以下端點：

- **認證與用戶**：
- `POST /auth/send_otp`：發送手機驗證碼（Step3已述）。
- `POST /auth/verify_otp`：校驗OTP並註冊/登入，回傳JWT及用戶資訊。
- `POST /auth/login`：（如實現密碼）驗證帳密換取JWT。
- `GET /auth/profile`：獲取當前用戶資訊（需要驗證）。返回姓名、可用現金、總資產、累計損益、報酬率等概要。前端進入App或刷新首頁時可呼叫。
- `PUT /auth/profile`：更新用戶資訊（例如修改暱稱等，非關鍵必需可後加）。
- **行情資料**：
- `GET /api/quote?symbol=<代號>`：取得指定股票即時行情。返回欄位包括股票代號、名稱、當前價格、漲跌幅、當日最高最低、成交量、時間等。可直接調用 yfinance 獲得相關 info 和 fast_info。
- `GET /api/history?symbol=<代號>&period=<期間>&interval=<間隔>`：取得歷史價格序列，用於畫K線圖或走勢。回傳該區間內按日/分鐘等的收盤價、成交量等列表。
- （未來可加）`GET /api/search?query=<關鍵字>`：股票代號/名稱搜尋，返回匹配的股票清單。這需要預先在 Stocks 表存公司名稱，可透過LIKE查詢或引入全文檢索。初版可不做，由使用者自行查代號。
- **交易相關**：
- `GET /api/portfolio`：取得當前用戶的投資組合資訊。返回可用現金、總資產、持倉清單陣列。持倉清單包含每檔股票的代號、名稱、持有股數、現價、持倉市值、平均成本、浮動盈虧（現價股數 - 成本股數）、以及在整個投資組合中的比例等。前端首頁或資產頁需要展示這些資訊並可用圓餅圖表示比例。
- `GET /api/transactions`：取得歷史交易紀錄列表（可加參數如近期N筆或者分頁）。用於交易明細查詢或提供給AI分析。
- `POST /api/trade`：執行模擬交易下單（Step5詳述）。JSON內容參見上步。返回結果包括 updated portfolio summary 或至少返回該筆交易結果（價格、數量、現金結餘等）和訊息。成功時HTTP 200，有錯誤時HTTP 4xx。
- **防重複提交**：要注意確保一筆交易請求只執行一次。可在後端對每個請求做唯一性處理（例如短時間重複的相同請求拒絕），防止網路延遲時使用者重按造成重複扣款。
- **排行榜與績效**：
- `GET /api/rankings?period=<週或月或年>`：取得模擬交易比賽排行榜。`period`參數可能值："weekly", "monthly", "yearly" 對應週/月/年排行。返回該期間的前N名用戶ID及報酬率、盈虧等，以及本用戶自己的名次和表現。
- `GET /api/performance`：取得當前用戶的累計績效（總資產、累計報酬率、可能還包括日/週/月曲線）。這在前端也許和 profile 合併，但可分開管理。
- **其他**：
- `POST /api/referral`：（選擇性）提交好友邀請碼。JSON包含 invite_code，被邀請者token識別本人。後端驗證碼有效性，進行獎勵發放（更新 Users 餘額和 Referrals 表記錄）。返回結果（成功或錯誤訊息）。
- `GET /api/analysis`：（選擇性）根據用戶歷史交易生成分析報告。由於AI分析也可能在賣出後自動推送，所以這端點可有可無。如果用戶想主動查看自己一段時間的表現，這裡可以整理統計，例如勝率、平均損益、最大獲利/虧損、常犯錯誤提示等。

實作提示： - 在 Flask 中使用 Blueprint 將模組化這些路由，例如 auth相關路由一組、行情一組、交易一組，方便維護。 - 每個請求進入點首先要做 JWT 權限驗證（auth路由除外）。可用 Flask-JWT-Extended 或自行驗證

token。 - 回傳格式使用 JSON，並且包含 `success` 或 `error` 欄位。統一格式方便前端處理。 - 日誌記錄：在伺服器端對每筆交易執行、排程任務執行等關鍵步驟寫入日誌檔，有助除錯。注意避免在日誌中記錄敏感資訊（如使用者密碼或JWT密鑰）。 - 錯誤處理：使用 Flask 的錯誤處理機制，自定義錯誤輸出格式。例如用 `@app.errorhandler(Exception)` 捕捉未預期錯誤，回傳 JSON 錯誤訊息，避免伺服器500崩潰時用戶端無響應。

完成API設計並實作後，可以用工具測試（如 Postman 或 curl）每個路由是否依預期運作，準備進入前端串接。

第7步：iOS 前端開發與介面設計

前端將以 Swift 開發，負責與使用者互動並調用後端 API。需確保模組分工明確（例如網路層、資料模型、視圖等）。重點視圖與功能如下：

- **網路層封裝**：建立一個服務類（例如 `APIClient` 或 `NetworkManager`）處理與後端的通訊。使用 `URLSession` 發送 HTTP 請求，解析回傳的 JSON。可以封裝通用的 GET/POST 方法，以及對應上述各API的方便函式：
- `sendOTP(phone)`、`verifyOTP(phone, code)`、`login(phone, pwd)` 等 -> 返回 token 等。
- `fetchProfile()`、`fetchPortfolio()`、`placeTrade(order)`、`fetchQuote(symbol)`、`fetchHistory(symbol,period)`、`fetchRankings(period)`...
- 在此類中全域管理一個 `authToken`，每次請求時在header附加 `Authorization: Bearer <token>`。並處理可能的401未授權情況（如token過期時通知用戶重新登入）。
- 可利用 Swift 的 Codable 結構將 JSON 解析成模型，例如建立 `StockQuote`、`Portfolio`、`TradeResult`、`RankingEntry` 等 struct，使數據更易於使用。
- **註冊/登入視圖**：
- **PhoneInputView**：讓用戶輸入手機號並按下「發送驗證碼」。調用 `APIClient.sendOTP`。收到成功回應後，UI切換到 OTP 輸入。
- **OTPVerifyView**：輸入6碼驗證碼並提交 `APIClient.verifyOTP`。成功則拿到 token 和用戶資訊，存儲 token（Keychain存儲或 AppStorage）。然後切換到主畫面(Main/HomeView)。
- 在驗證過程中增加 Loading 狀態、錯誤提示，如驗證碼錯誤可讓使用者重試或重新發送。
- 如果規劃採用密碼登入，則在 PhoneInputView 之後增加 SetPasswordView；下次登入時走 LoginView（輸入手機+密碼調用 login API）。
- **主首頁 (HomeView)**：
- 此畫面展示用戶帳戶概況和入口導航：
    - 帳戶總覽：顯示**總資產**、**可用現金**、**累計報酬率**（例如「本期報酬率」或「累計報酬率」）。這些可在載入Home時呼叫 `fetchProfile` 或 `fetchPortfolio` 拿到數值。總資產=現金+持股市值合計，報酬率= (總資產/初始本金-1)*100%。
    - 投資組合圓餅圖：用 SwiftUI Charts 或類似庫根據持倉各股票市值比例繪製圓餅圖，直觀展示資產配置。需要持倉列表資料（各股票市值），來自後端 `/portfolio` 返回。若比例過小的可合併為「其他」以保持圖表清晰。
    - 快捷操作按鈕：如「開始交易」、「查看排行」、「交易紀錄」、「好友邀請」等導航按鈕。這些可以是 NavigationLink 導向其他子視圖。
    - **行情快照**：可選地，顯示幾檔熱門股票或用戶關注股票當前價，用於豐富首頁資訊（這需要引入自選股功能或推薦板塊，可列入未來優化）。
- **股票查詢與詳細**：

- **QuoteSearchView**：讓用戶查找股票資訊。可以是一個搜尋欄，輸入公司名稱或代號即時篩選出匹配結果（需要本地有股票清單，可以在App啟動時從後端獲取一次Stocks表資料，或每次在後端搜索）。點選某項後，跳轉到股票詳細頁。
- **StockDetailView**：顯示個股詳細資訊，包括：
    - 股票名稱、代號、當前價格、漲跌、成交量等（使用 `fetchQuote` 獲取）。
    - 一個小型走勢圖：可用 Charts 庫繪製近幾日價格走勢（調用 `fetchHistory(symbol, period="1mo", interval="1d")` 拿到數據）。或提供按鈕切換日線/週線/月線圖（不同參數調 API）。
    - 「買入」按鈕（如果用戶未持有該股）或「賣出」按鈕（如果有持倉）等行為入口。點擊後進入下單視圖，帶入該股票代號。
    - 其他資訊：如本益比、市值、股利等，可從 `Ticker.info` 中獲得，如果需要展示基本面信息。（初版重點在交易，可簡化不做。）
- **下單視圖 (TradeView)**：
- 無論從StockDetailView點擊買/賣，或從首頁「開始交易」進入，都進入同一個下單介面，差別在於預先選定了股票與動作與否。
- 該視圖包含：
    - 股票代號/名稱（若從詳情帶入則鎖定，否則提供輸入或選擇器讓用戶挑選持倉或搜尋股票）。
    - 切換「買入 / 賣出」的控制（例如SegmentedPicker）。切換時可連動更新一些提示（如可用餘額 vs 可賣股數）。
    - 輸入欄位：數量（張）。如果要支援零股，可改成股數輸入並標明，或提供切換整股/零股模式。初期僅整股交易，數量應為整數張。
    - 價格類型選擇：預設「市價」。如允許也可選「限價」，此時顯示一個價格輸入欄。價格欄位可預填當前市價並允許調整。需注意限制用戶輸入不合理價格。
    - 總金額試算：即時計算「預計花費/收入 = 價格張1000」，以及下方提示「扣除手續費後約 ... 元」。讓用戶了解資金佔用情況。
    - 可用餘額/持有股數提示：在買入模式下顯示「可用現金：...」，在賣出模式下顯示「持有張數：...」供用戶參考。
    - 提交按鈕：「確認買入」或「確認賣出」。點擊觸發交易：
    - 呼叫後端 `POST /api/trade`。為避免重複下單，可在按鈕觸發後短暫disable並顯示 loading。
    - 收到成功回應後，給出交易成功通知（Alert或者Banner）。同時更新本地的資產數據，例如透過返回的 portfolio 資訊刷新首頁或持倉列表。如果有AI建議內容，也可以在成功彈窗或跳轉到分析視圖顯示。
    - 如果後端回傳錯誤（如餘額不足），則彈出錯誤Alert提示用戶。
    - **UX細節**：可加入安全問詢，避免錯按：如點擊「確認賣出」彈出對話框讓用戶再次確認賣出某股票幾張。
- **持倉與交易紀錄視圖**：
- **PortfolioView**：列出用戶目前所有持倉（可整合在首頁或獨立頁面）。列表每項包含股票代號、名稱、持有股數（張）、當前價、持倉市值、浮動盈虧額與百分比。可用直觀的紅綠色表示漲跌或盈虧。
    - 每項點擊可跳轉到 StockDetailView 或直接跳到 TradeView的賣出模式，方便快速賣出。
    - 頁面上方還可顯示總持股市值、總浮動盈虧。
- **TransactionsView**：交易紀錄頁，按時間列出所有買賣操作。每筆記錄顯示日期時間、股票代號、動作類型、數量、價格、手續費、損益。損益為賣出才有數值，買入紀錄顯示「--」或0。
    - 可以加入篩選或排序，例如只看某股票的交易紀錄，或依盈虧排序等（非必要）。
    - 這頁資料由 `GET /api/transactions` 提供。由於資料量隨時間增大，可實作分頁或滾動加載，後端支持 limit/offset 參數。
- **AI 交易分析與建議視圖**：

- 每當用戶完成一筆賣出交易，如果後端返回了建議內容，前端可以彈出一個**分析卡片**或**對話視窗**顯示本次交易的分析總結和建議策略。例如：
    - 「你在2330的這筆交易虧損5%。買入價高於近期平均，賣出時趨勢下跌。建議下次設定止損點，控制風險。【點擊這裡了解風險管理課程】」
    - 建議內容中可嵌入鏈接，點擊後可能開啟App內的網頁視圖或外部瀏覽器至課程頁。
- 此外，可設計一個**ChatBotView**，模擬與投資助理AI聊天的介面。用戶可以提問如「目前有什麼交易建議？」、「我的投資表現如何？」，AI 可以根據交易記錄和行情給出回答。
    - 目前AI可先簡單規則實現，例如固定回答投資心得或從資料庫抓一些統計呈現。未來可整合OpenAI的API，將用戶交易數據作為prompt生成更個性化的建議。
    - ChatView在技術上與一般聊天界面類似：上方是ScrollView列出對話訊息，下方TextField讓用戶輸入問題並送出，送出後呼叫後端AI分析API（或直接在App本地用簡單邏輯處理）。
    - 本專案已有提到一個 chatview，說明您可能已經實作了基本的聊天UI，後端可以配合提供一個 `/api/chat` 的AI接口。但優先級可稍後，再完善體驗。
- **排行榜視圖 (RankingView)**：
- 提供週排行、月排行、年排行的切換（SegmentedControl 或 Picker）。選擇後呼叫對應的 `fetchRankings(period)` API 獲取資料。
- 顯示排行榜列表：例如前10名的用戶名稱（或匿名）、報酬率、資產等。如果為隱私，可只顯示部分暱稱如「用戶***5」等。
- 突出顯示**自己**的排名：如果自己不在前列，也應該在列表底部顯示「你的排名：第X名，週報酬率Y%」。後端在返回排行榜時可以帶上用戶本人的排名。
- UI可以使用 List 列表呈現，每項顯示名次、名稱、報酬率；冠軍可以特別標註🏆符號等以增加趣味。
- 此功能鼓勵競爭與留存，用戶可以每週查看自己的排名是否進步。
- **好友邀請介面**：
- 在個人或更多頁面提供邀請功能。顯示我的邀請碼，點擊可以複製或分享（透過 UIActivityViewController 分享文字到Line/訊息等）。
- 讓用戶輸入他人的邀請碼領取獎勵。可以在註冊時就有輸入，也可以在這裡補輸入。輸入後調用 `/api/referral`，成功則提示獲得獎勵金額並更新餘額顯示。
- **導航架構**：
- 使用 TabView 或 NavigationView 管理主要介面。可能的架構：
    - Tab1: 首頁 (含持倉概要)
    - Tab2: 交易/搜尋 (交易功能入口，可整合股票搜尋和下單)
    - Tab3: 排行榜
    - Tab4: 訊息/聊天 (AI助理、通知)
    - Tab5: 個人/Profile (包含邀請好友、設定)
- 也可以使用抽屜菜單或單一頁面多導航。但 Tab 的方式直觀易用。
- 適當使用 NavigationLink 進入細節頁，如從首頁->持倉列表->股票詳情->交易。
- 確保狀態管理：可以使用 `ObservableObject` ViewModel 在不同視圖共享資料（例如 PortfolioViewModel提供資產、持倉資料給首頁和持倉頁）。每次交易後更新該 ViewModel 以反映最新狀態。

經過這一步，前端介面各主要功能將基本就緒。建議先以**假資料**搭建UI（如先使用mock JSON文件或本地生成假數據填充列表），待與後端串接測試API後再調整。這樣能並行開發前後端，提高效率。

第8步：定時任務與績效排行實現

為了計算週/月/年績效並生成排行榜，我們需要在後端或資料庫設置定時任務 (Scheduled Tasks) 週期性地彙總用戶表現。Supabase 提供了 pg_cron 或 Edge Functions 來實現此目的：

- **績效指標定義**：排行榜通常以報酬率（收益率）排序。需明確計算方式：
- **週報酬率**：本週的盈虧百分比。可定義每週一開盤前作為起始點。例如每週一的總資產做基準，到週末收盤的總資產比較計算。(若用戶中途有充值模擬金或提領則要排除，不過本平台沒有真實金流除了邀請獎勵，可視為額外本金，需要在計算報酬率時扣除影響，較為複雜，可忽略。)
- **月報酬率**：同理，每月初資產 vs 月末資產。
- **年報酬率**：可以是年度累計，或直接用當年年初至今。
- 為簡化，**也可以採用累計報酬率**來排名，例如不分週月，只比較從帳戶建立到現在的總收益率。然而活動性會差些。建議仍按週月年給不同榜單，提高參與度。
- **定時快照**：
- 透過 Supabase **pg_cron** 模組在資料庫層執行定時任務。首先在資料庫中安裝 pg_cron (Supabase已支援，需在SQL編輯器ENABLE)。然後撰寫 SQL 排程，例如：

```
SELECT cron.schedule('weekly_snapshot', '0 0 MON * *', $$
  INSERT INTO PerformanceSnapshots (user_id, period, date, total_assets)
  SELECT id, 'weekly', CURRENT_DATE, (cash_balance + ...計算持倉市值...)
FROM Users;
$$);
```

  以上是一個概念，實際SQL要計算每個用戶的總資產。同理設 monthly_snapshot, yearly_snapshot 安排對應的 cron 表達式（每月第一日0時，每年1月1日等）。
- 或者，使用 Supabase **Edge Function** 編寫一段伺服器端程式（使用 Deno/TypeScript），在其中查詢資料表計算報酬率，然後由 Supabase Scheduler 觸發每天/每週執行該函式。Edge Function 的好處是可以寫複雜邏輯，用程式計算再寫回資料庫。Cron 表達式可以設在 Supabase設定裡。
- 若不使用Supabase內建，也可以**自行在後端部署 Cron**：例如使用 `APScheduler` 在 Flask 啟動時排程，或利用雲端服務（Heroku Scheduler, AWS CloudWatch Events等）每天/每週發請求到一個隱藏API去計算。為集中管理，建議用Supabase pg_cron在DB端完成快照，以減少服務器負載。
- **計算方法**：假設我們在 Users 表保存了 `initial_capital`（初始本金）和 `total_profit`（累計實現盈虧），還有目前持倉浮盈可計算：
- 週報酬率 = (本週末總資產 - 上週末總資產) / 上週末總資產 * 100%。這需要知道上週末每個用戶的總資產，可從 PerformanceSnapshots 表讀取 'weekly'類型最後一筆作基準。
- 但更簡單方式：每次 snapshot 時直接計算**截至當時**的年初至今、月初至今、週初至今的報酬率並儲存，無需倒推基準。例如：
  - 每週五收盤後，對每個user計算 `weekly_return = (current_total_assets / last_week_total_assets - 1)*100%`，並存入一個 RankingResults 表或更新 Users 表的 weekly_return欄位。
  - 但是 last_week_total_assets 就是上週snapshot存的數值，要取出計算。所以兩階段：snapshot存每周初始值；每日或每週末另跑計算。
  - 或者在週末直接比對PerformanceSnapshots表中 period='weekly' 倒數第二筆和最後一筆。
- 這略顯繁瑣。另一種折衷：**每晚收盤**記錄所有用戶當日總資產PerformanceSnapshots(date=今日)。則：
  - 週排行：取 `今日` 與 `7天前` 兩個快照的差值計算每個人漲跌幅。

    - 月排行：取 `今日` 與 `30天前`（或上月同日）。
    - 年排行：取 `今日` 與去年最後一天或今年第一天。
    - 有可能週或月內用戶新加入或未持股，需要處理除數為0的情況。可以約定新加入當週初始資產=初始本金（獎勵另算）。
- 為簡化實作，我們可以：
    - **在 Users 表直接保存**目前累計報酬率（從起初到現在）。排行榜時直接排序這個值。然後**另外**保存 weekly_roi, monthly_roi, yearly_roi 欄位每天更新。
    - 例如使用Edge Function每天0點執行：對每個用戶計算weekly_roi = (current_assets / assets_7_days_ago - 1)*100%，monthly_roi類似，yearly_roi類似。直接更新Users表欄位。這樣查排行榜時只需 `ORDER BY weekly_roi DESC` 即可。
    - 此方法實現需要每天取得7天前、30天前、年初的資產值。可透過 PerformanceSnapshots 幫助。
    - 初版可聚焦**累計報酬率**排行以快速完成功能，週/月榜可作為拓展。
- **排行榜生成**：
- 無論以上哪種方式，最終都需要將**計算出的報酬率**存儲，以便前端請求時快速獲取。建議建立一張 **Ranking** 或 **Leaderboard** 資料表，存每種周期最新的前N名結果，這樣前端不用拉整個User表排序（如果用戶很多效率低）。
- 或乾脆直接查 Users 表取需要的欄位排序（在可控用戶數下可行）。Supabase的PostgREST可以直接用 order, limit 參數。
- 例如 `GET /rest/v1/Users?select=name,weekly_roi&order=weekly_roi.desc&limit=10` 直接返回前10週績效。若已透過JWT認證，可能需要註冊RLS policy允許讀取他人績效（排行榜應該是公開的匯總資訊，可以放寬或做個安全視圖）。
- 在Flask，我們可以將排行榜API實作為：

```
@app.route('/api/rankings')
def get_rankings():
    period = request.args.get('period')  # 'weekly'/'monthly'/'yearly'
    top_list = db_service.get_top_rankings(period, limit=10)
    user_rank = db_service.get_user_rank(user_id, period)
    return {"period": period, "top": top_list, "user": user_rank}
```

  其中 db_service.get_top_rankings 可能就是個查詢排序的封裝。
- 資料內容：top_list 裡每個元素含名次、用戶名(或代號)、報酬率。user_rank 包含當前用戶的名次和報酬率。這些數字可以在Edge Function計算時順便排名或在查詢時用窗口函數ROW_NUMBER()計算。
- **驗證**：完成排行後端計算後，多造幾個用戶賬號做交易（可以直接在資料庫調整資產）以產生不同績效，測試排行榜是否正確。確認週期切換、名次計算、自己的名次顯示都如預期。

第9步：整合測試與優化

當前後端主要功能都實作完成後，需要進行全面的測試，確保每個部分協同工作，並做最後的優化：

- **功能測試**：模擬完整用戶流程：
- 註冊新帳號 -> 登入 -> 瀏覽行情 -> 下單買入 -> 確認資金/持倉更新 -> 下單賣出 -> 查看損益分析 -> 查看交易紀錄 -> 對照資料庫驗證。
- 測試錯誤流程：餘額不足時下單、持股不足時賣出、輸入錯誤代號、OTP錯誤等，看前端提示是否友好正確。
- 多用戶之間邀請碼流程：A邀請B，B註冊輸入碼，檢查A、B的資金是否各增加獎勵，且不能重複獲取。

- 排行榜長時間測試：修改時間或模擬定時任務執行，檢查週/月/年排行的變化是否符合預期（例如某用戶本週盈利最高，週排行第一；下週如果平掉則新週排行清零等）。
- **性能考量**：
- yfinance呼叫效率：如果發現每次下單都調用行情API導致延遲，可以考慮在**下單前前端就已經有價格**（例如TradeView中實時顯示價格），下單請求時也帶上該價格以供後端校驗使用，這樣後端可以省一次API查詢。但為了資料準確，後端仍以自身查到的為準，前端帶價格只能作參考或防止用戶惡意改價。
- 資料庫效率：定期清理過舊的交易紀錄或將其存檔，避免 Transactions 表無限制膨脹影響查詢。或者對常用查詢欄位建立索引（例如 user_id, date 等）。
- 圖片與外部連結：若嵌入課程連結，注意前端跳轉處理。課程推薦內容可考慮由後端根據策略返回一個課程ID或URL，前端渲染成點擊連結。
- **安全與容錯**：
- 保護後端金鑰：在整個應用部署前，檢查 `.env` 檔案確保包含必要的密鑰，如Supabase Service Key、JWT密鑰、SMS API Key等，程式碼中不要將這些硬編碼。部署時配置環境變數或確保 `.env` 不被上傳版本控制。
- 防止惡意操作：由於是模擬環境，尚無真金錢損失，但也要避免有人利用漏洞作弊排行。例如攔截API自行修改返回值或繞過前端調用後端接口。使用JWT和資料庫檢查可以防基本的作弊。但如果有人破解您的通訊協定，也僅影響他自己帳號的數據，不會有金錢損失。仍應在排行榜中做合理性檢查（例如報酬率超現實的可能標記人工審核）。
- **部署**：
- 後端 Flask 可以部署在 Heroku、Railway、AWS EC2、Fly.io 等雲平臺。需要配置持續運行的服務，安裝所需套件以及將 `.env` 變數設好。確認伺服器對外的安全性（例如設定只允許HTTPS訪問，JWT密鑰足夠複雜）。
- 前端 iOS App 在 Xcode 測試無誤後，可打包 TestFlight 測試版邀請測試者試用。上架前記得在 App Store 後台填寫相關模擬交易無金流的備註（避免誤會為真實交易App）。
- **迭代優化**：
- 根據測試者反饋完善UI/UX，例如增加教學提示（第一次使用引導用戶完成一筆交易），顯示更多統計（日內損益曲線等）。
- 機器人功能持續優化：將 AI 建議從簡單規則升級為更智能的分析，可以考慮紀錄下更多交易上下文，在賣出時送到雲端AI模型生成建議文字返回。此外可增加其他功能如「股價查詢」（透過對話詢問機器人特定股票現價或新聞）。
- 增加更多交易標的：目前僅支援股票現股交易，可研議加入ETF、期貨或美股模擬等，當然資料來源和模擬規則需要相應拓展。
- UI的多語系支援（繁中、簡中、英文）如果面向更廣受眾，可以納入規劃。
- **資料備份**：使用 Supabase 自帶每日備份功能，確保模擬資料庫萬一出問題可以還原，避免用戶辛苦累積的紀錄遺失。

經過以上步驟，將完成一個**前後端協作、模組清晰**的股票模擬交易平台。此平台提供從行情查詢、模擬下單、損益計算到社群競賽排行的完整體驗。您可以將本計畫書交給協作者（如 Claude）逐步實作代碼：先後端核心、再前端串接，循序漸進開發。在開發過程中不斷測試和優化，最終實現提升用戶金融認知、防範盲從詐騙的初衷目標。

請按照計劃書步驟開始實作，祝開發順利！
