*5.4 Prototype*

**Investor Centre Ltd Prototype System**

**1. Overview**

This prototype represents the foundational development of a financial trading and investment management system for Investor Centre Ltd. The goal is to transition from a paper-based system to a digital platform that improves efficiency, security, and data management. The prototype focuses on user account management, trade logging, and database interactions.

**2. System Architecture**

The project is structured into three main modules:

* **db\_setup.py**
  + Handles database creation and schema definition using SQLite.
  + Ensures tables such as user accounts and transactions exist.
* **data\_access.py**
  + Provides functions to interact with the database (insert, retrieve, update data).
  + Implements trade logging, account management, and financial data retrieval.
* **main.py**
  + Implements the core application logic and user interface.
  + Manages user input, processes transactions, and displays data.

**3. Features Implemented in the Prototype**

**Database Management**

* **User Accounts:** Stores user credentials and account details.
* **Trade Transactions:** Logs trades with attributes such as date, currency pair, and amount.
* **Data Integrity:** Ensures tables are properly structured to prevent data redundancy.

**User Interface (GUI)**

* **User Login & Registration:** Basic authentication for system access.
* **Trade Execution:** Allows users to record transactions.
* **Data Retrieval:** Fetches historical trades for analysis.

**4. Features Omitted in the Prototype**

* **No Validation or Data Encryption:** Security measures such as password hashing are not implemented.
* **Styling**: This prototype features a very minimalist styling.
* **Customer balance:** The prototype currently doesn’t let customers view, top up or withdraw; they are assigned their account balance from the previous system manually.
* **Chart Plotting:** This prototype doesn’t let customers plot financial charts for analysis.
* **Live price updates:** This prototype doesn’t allow customers to view a live profit/loss, only when they close the trade.

**5. Evaluation & Future Enhancements**

**Strengths**

* **Well-Structured Database:** Tables are designed to store key financial data efficiently.
* **Separation of Concerns:** Different modules handle database operations, business logic, and user interaction separately.
* **Scalability:** The system can be expanded to include more financial instruments and trading functionalities.

**Limitations**

* **Limited Error Handling:** Basic exception handling is used, but comprehensive logging and debugging tools need to be added.
* **Security Gaps:** Lack of encryption, authentication, and transaction validation.
* **User Interface Constraints:** The current UI is minimal, requiring improvements for a professional financial system.

**Future Improvements**

* Implement strong user authentication with encryption.
* Improve styling.
* Reduce code duplication.

**6. Conclusion**

This prototype establishes the core structure of Investor Centre Ltd’s trading management system. While it effectively demonstrates trade logging, database integration, and account management, future iterations should focus on security, UI enhancements, and real-time data processing to create a fully functional platform.

**Prototype Analysis**

**A black background with white text

Description automatically generated**

First, I had to import all the necessary libraries that would be used in this prototype. I chose to use the tkinter library to create the user interface, as it provides a simple yet effective way to develop GUI applications. The ttk module was also imported from tkinter to style widgets such as buttons, labels, and dropdown menus, making the interface more visually appealing. The messagebox module was included to enable user interaction through informational and error messages. Additionally, I used the font module to allow customization of text styles within the interface.

For financial data retrieval, I utilized the yfinance library, which allows for seamless access to real-time and historical market data. To visualize trading data effectively, I incorporated mplfinance, a powerful library for plotting financial charts. The sv\_ttk module was imported to enhance the appearance of the UI with a modern theme. The data\_access file was included to manage interactions with the database, ensuring smooth data handling. Furthermore, I used the datetime module to handle date and time formatting within transactions, while the re module assists in validating user inputs, such as email addresses. Lastly, I imported warnings to manage any non-critical runtime warnings, ensuring a smooth user experience.

A screen shot of a computer program

Description automatically generated

A screen shot of a computer program

Description automatically generated

In the above code, the login function is responsible for handling the user authentication process.

It begins by declaring the global variables customer\_id and customer\_balance, which will store user-specific details after a successful login. The function then retrieves the username and password from the input fields using entry\_username.get() and entry\_password.get().

The function first checks if either field is empty. If so, it displays an error message using messagebox.showerror, prompting the user to fill in all required fields before proceeding. If both fields are filled, it calls the get\_login\_details function to verify if the provided credentials match a valid customer account. If the login is successful, a confirmation message is displayed via messagebox.showinfo, and the open\_home function is executed to load the main user interface.

If the credentials match an administrative account instead, the function calls get\_login\_details\_staff, allowing staff members to log in. In this case, the system displays an admin-specific success message and directs the user to the open\_home\_staff function. If the credentials are invalid, the function presents an error message informing the user of incorrect login details. Finally, customer\_id and customer\_balance are retrieved using get\_customer\_details, ensuring that relevant customer information is available for later use.

The open\_home function is responsible for creating the main user interface after a successful login. It begins by hiding the original login window using root.withdraw(), ensuring that the user cannot interact with it while the new window is active.

A new Toplevel window is then created and assigned the title **"Welcome"**, with a fixed size of 400x600. The function also applies a light theme to the window using sv\_ttk.set\_theme("light"), improving the visual design of the interface.

Within the new window, several elements are added to help user interaction. A header label titled **"Home"** is displayed in large font, followed by a welcome message introducing **Investor Centre Ltd**. Below this, multiple buttons are arranged to allow users to access different functionalities. These include a **"Chart"** button that opens a chart configuration tool, a **"Manage Orders"** button for handling transactions, a **"Balance"** button to update the user’s balance, and a **"View History"** button to display past trade activities. Additionally, a **"Logout"** button is provided, which destroys the new window and restores the login window using root.deiconify(), allowing the user to return to the login screen.

This structure ensures a seamless transition from login authentication to the main system interface, maintaining a clear separation between user authentication and application functionality.

**A screenshot of a phone

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

**A screenshot of a computer program

Description automatically generated**

The create\_account\_window function is responsible for opening a new window where users can register an account. It begins by creating a Toplevel window titled "Create Account" with a fixed size of 400x520 pixels, ensuring a consistent user experience. The sv\_ttk.set\_theme("light") function is applied to maintain a professional and modern interface. To enhance usability, a padding dictionary is defined for uniform spacing between UI elements.

The interface consists of labeled input fields that allow users to enter essential details such as first name, surname, date of birth, email, phone number, username, and password. Each label and entry field is neatly organized using grid() layout management, ensuring alignment and readability. A title label "Create Account" is prominently displayed at the top of the form with bold formatting to emphasize the window’s purpose. The password entry field uses show="\*" to mask user input, enhancing security.

At the bottom of the form, two buttons are provided: a "Create Account" button and a "Close" button. The "Create Account" button triggers the create\_new\_account function, which validates user input before attempting to store account details in the database. The "Close" button allows users to exit the window without making any changes.

The create\_new\_account function retrieves user inputs from the entry fields and applies a series of validation checks. First, it ensures that no fields are left empty. Then, it validates specific fields using regular expressions:

* The date of birth must follow the DD/MM/YYYY format.
* The email must match a standard email pattern.
* The phone number must contain exactly 11 digits.
* The first name and surname must contain only alphabetic characters.

If any validation fails, an appropriate error message is displayed using messagebox.showinfo, guiding the user to correct their input. If all validations pass, the function calls create\_account (assumed to be defined in data\_access.py) to attempt account creation. If successful, a success message confirms the account creation, and the window closes. If the account creation fails, likely due to a duplicate username, an error message informs the user.

This structured approach ensures that only valid user details are stored, improving data consistency and security while enhancing the overall user experience.

**A screenshot of a computer

Description automatically generated**

**A screen shot of a computer program

Description automatically generated**

The manage\_orders\_window function is responsible for creating a new window that allows users to place buy or sell orders for currency pairs. It begins by initializing a separate window using tk.Toplevel(), setting its title to **"Manage Orders"** and defining its size as 400x515 pixels. The sv\_ttk.set\_theme("light") function is applied to maintain a consistent visual style. The function first retrieves the latest price data for the default selected currency pair using the yfinance library.

This is done by fetching the **open price** from a one-day, one-minute interval dataset and rounding it to four decimal places for accuracy.

A dropdown menu (ttk.Combobox) allows users to select a currency pair from predefined options. This dropdown is linked to an event listener (major\_dropdown.bind("<<ComboboxSelected>>", update\_open\_price)) that updates the displayed price whenever a different pair is chosen. Below the dropdown, a label displays the **current price** of the selected currency pair, ensuring users have up-to-date market information before placing an order. An entry field labeled **"Enter GBP Amount"** allows users to input the amount they wish to invest.

The interface includes multiple buttons for different functionalities. A **"Buy"** button is provided to initiate a buy order, triggering the manage\_orders\_buy function, while a **"Sell"** button allows users to sell their selected currency pair by calling manage\_orders\_sell. A **"Current Orders"** button opens another window displaying the user’s active orders, ensuring they can track their trades. Finally, a **"Close"** button is included to exit the order management window.

The manage\_orders\_buy function is responsible for executing buy orders and validating user input. It begins by setting the order type to **'BUY'** and attempts to retrieve the user’s entered amount in GBP. If the input is not a valid integer, an error message is displayed using messagebox.showerror. The function also checks whether the user has **sufficient funds** to place the trade; if the entered amount exceeds the available balance, an error is triggered to prevent overdrawing.

The function then validates whether the selected currency pair exists in the predefined list (major\_options). If the user enters an invalid pair, an error message is displayed. Once the currency pair is confirmed, its corresponding ticker symbol is retrieved from the currency\_pairs dictionary. A new window is then created using tk.Toplevel() to serve as an **order confirmation page**, where the trade details are displayed before finalizing the order.

The function fetches **real-time price data** for the selected currency pair using yfinance, ensuring that the order is placed at an accurate market rate. The current time is also retrieved using datetime.now() and formatted to the nearest second for record-keeping. Labels are displayed to show essential order details, including the **currency pair, order size, and price**.

Before finalizing the order, the function ensures that an amount has been entered and displays a confirmation message. The trade details are then stored in the database using the store\_order function, which logs the **customer ID, currency pair, order type, amount, price, and timestamp**. If the order is successfully stored, a success message is displayed to the user, confirming the trade. If an error occurs during the database transaction, an error message informs the user that the order could not be saved. This ensures a smooth and error-handling trade execution process, allowing users to confidently place trades while minimizing input errors.

**A screenshot of a computer screen

Description automatically generatedA screenshot of a phone

Description automatically generated**

**A screen shot of a computer program

Description automatically generatedA screen shot of a computer program

Description automatically generated**

The current\_orders\_window function is responsible for displaying a list of ongoing orders in an interactive and structured format. It begins by creating a new window using tk.Toplevel(), setting its title to **"Current Orders"** and defining its size as 870x440 pixels. To ensure a consistent and professional appearance, the light theme is applied using sv\_ttk.set\_theme("light"). The function then initializes a Treeview widget, which serves as a table to display orders in a structured manner. This table contains seven columns: **Order ID, Customer ID, Currency Pair, Order Type, Amount, Price, and Order Time**, each with clearly defined headings and appropriate column widths for readability. The column widths and text alignment are specified to maintain an organized layout, ensuring that key trade details are easy to interpret.

To populate the table, the function retrieves live order data from the database by calling fetch\_orders(), which fetches all active trades. The retrieved data is dynamically inserted into the Treeview, allowing users to view real-time trading activity. Additionally, a **"Modify Trades"** button is included, enabling users to adjust their existing orders by calling the modify\_orders\_window function. To ensure ease of navigation, a **"Back"** button is also present, allowing users to close the window and return to the previous screen. The function runs independently of the main interface, providing a detailed view of active trades without disrupting the core functionality of the application.

The history\_window function is designed to display the user’s trade history, allowing them to review past transactions and assess performance. It follows a similar structure to current\_orders\_window, creating a separate window titled **"Trade History"** with dimensions 870x400. The interface utilizes a Treeview widget to present historical trade data in a structured format, with columns including **History ID, Order ID, Currency Pair, Order Type, Amount, Price, Order Time Close, Pip Difference, and Profit/Loss**. These columns provide comprehensive details about closed trades, including when the trade was executed, its final price, and the resulting profit or loss. Column widths and text alignment are carefully defined to ensure clear data presentation.

To populate the trade history table, the function establishes a connection to the **main.db** database and retrieves relevant records from the history table using an SQL SELECT query. If an error occurs during data retrieval, an error message is displayed using messagebox.showerror, ensuring that users are notified of any issues. Once the data is successfully fetched, each record is dynamically inserted into the Treeview, allowing users to browse their past trades efficiently. A **"Close"** button is provided at the bottom of the window, enabling users to exit the trade history interface when finished. By implementing this functionality, the system allows users to analyse their trading activity over time, assess performance trends, and make informed financial decisions based on historical data.

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

### ****Evaluation****

● **The use of tkinter and ttk for the user interface ensures a simple and accessible design.**  
○ This allows for a user-friendly interface, making it easy for users to navigate and interact with the system without prior technical knowledge.

● **The implementation of sv\_ttk improves the visual appeal of the application.**  
○ The modernized theme enhances the professionalism of Investor Centre Ltd, aligning the interface with financial software standards.

● **Dropdown menus for currency selection allow for efficient trade input.**  
○ This ensures that users can quickly select their desired currency pair rather than manually entering values, reducing input errors, and improving usability.

● **Real-time data retrieval using yfinance ensures accurate market prices for trading.**  
○ This allows traders to make informed decisions based on live market data, improving the reliability of trade execution.

● **The integration of sqlite3 enables persistent storage of user and trade data.**  
○ This ensures that all transactions and user accounts are stored securely, reducing the risk of data loss while maintaining financial records.

● **The system includes validation for user input fields to improve data accuracy.**  
○ Ensuring correct formats for phone numbers, email addresses, and dates reduces the chances of errors and improves the credibility of stored information.

● **Error handling using messagebox provides clear feedback to the user.**  
○ This enhances user experience by informing them about incorrect inputs, unsuccessful transactions, and system errors, reducing confusion.

● **The separation of database operations into data\_access.py promotes code modularity.**  
○ This improves maintainability by ensuring that all database-related functions are centralized, making future modifications easier.

### ****Limitations and Areas for Improvement****

● **The system lacks strong security measures such as password hashing and encryption.**  
○ Storing passwords in plain text poses a security risk, and implementing encryption techniques would enhance data protection.

● **The current system includes buttons with no current functionality.**  
○ Buttons such as ‘Balance’ and ‘Chart’ do not have any current functionality, this leads to an unfinished feeling system, that makes the customer experience worse.

● **The system does not support multiple users with different roles (e.g. admin, trader).**  
○ Adding role-based authentication would allow for better user management, enabling administrative features for monitoring transactions.

● **The system is tedious to navigate, each button opens a new window rather than moving through the pages like.**  
○ Opening the new window and then closing the previous window will allow customers to navigate through the system more easily.