Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN

Software Architecture & Design – Sprint 3

Team Members:

Evan Balson – BAL18466416

Maahia Rahman – RAH23614335

Hala Bakhtiar – BAK23592238

Mahbouba Rezaei – REZ23579670

Roehampton University

Course Number: CMP020N207S

Lecturer: Dr Shekoufeh Kolahdouz Rahimi

Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN

StockShark

THE SOFTWARE ARCHITECTURE & DESIGN

Overview

StockShark is a web-based application that empowers users to view, compare, and manage stock portfolios (or watchlists). The application integrates with external market data APIs (e.g., Alpha Vantage) to retrieve real-time and historical stock data. It provides features such as:

- User Registration and Authentication: New users can register, log in, and manage their profile.
- Market Overview and Stock Comparison: Users can view market data, compare stocks over custom date ranges, and see graphical representations using Chart.js.
- Watchlist/Portfolio Management: Users can create and manage portfolios (watchlists) where they add their favorite or frequently traded stocks.
- Data Persistence: All user and portfolio data are persisted in a MySQL database using a robust schema with proper referential integrity.
- Interactive, Modern UI: A dark-themed interface with responsive design and interactive charts for a smooth user experience.

Architecture & Design:

The StockShark project follows a 5-tier architecture with the following layers:

Presentation Layer:

JSP Pages and HTML/CSS/JavaScript: User interface is rendered by JSPs that incorporate JSTL, custom EL expressions, and utilize CSS for styling. Client-side interactivity is provided via JavaScript and Chart.js for dynamic charting.

Controller Layer:

Servlets: Java servlets handle HTTP requests and responses. Notable servlets include LoginServlet, RegistrationServlet, CompareStocksServlet, PortfolioServlet, and WatchlistServlet. They are mapped using annotations (e.g., @WebServlet). They manage request/response cycles, encapsulate user interactions and forward data to the view fulfilling user stories such as viewing personalized market data and portfolios.

Business Logic Layer:

The business logic is organized into compound components like User_Management_Compound and Data_Handling_Component. This abstraction facilitates meeting complex user interactions (e.g., adding portfolios, updating charts) while keeping the controllers lean.

Data Access Layer:

Database Access Classes: Classes like PortfolioDAO encapsulate database access. They run SQL queries using JDBC with proper resource management (try-with-resources) and manage transactions. Instead of a DAO, our project uses a Database Client class to handle all persistence operations. This ensures that queries and updates such as retrieving portfolio details by joining portfolios with ‘portfolio items’ are centralized. This design was chosen to maintain flexibility and consistency with our overall application workflow.

Integration Layer:

The application integrates with external APIs (such as Alpha Vantage) to fetch real-time stock data. User stories requiring updated market information are supported through API calls managed via the Database Client. The API responses (JSON) are parsed and converted into domain objects (like StockData).

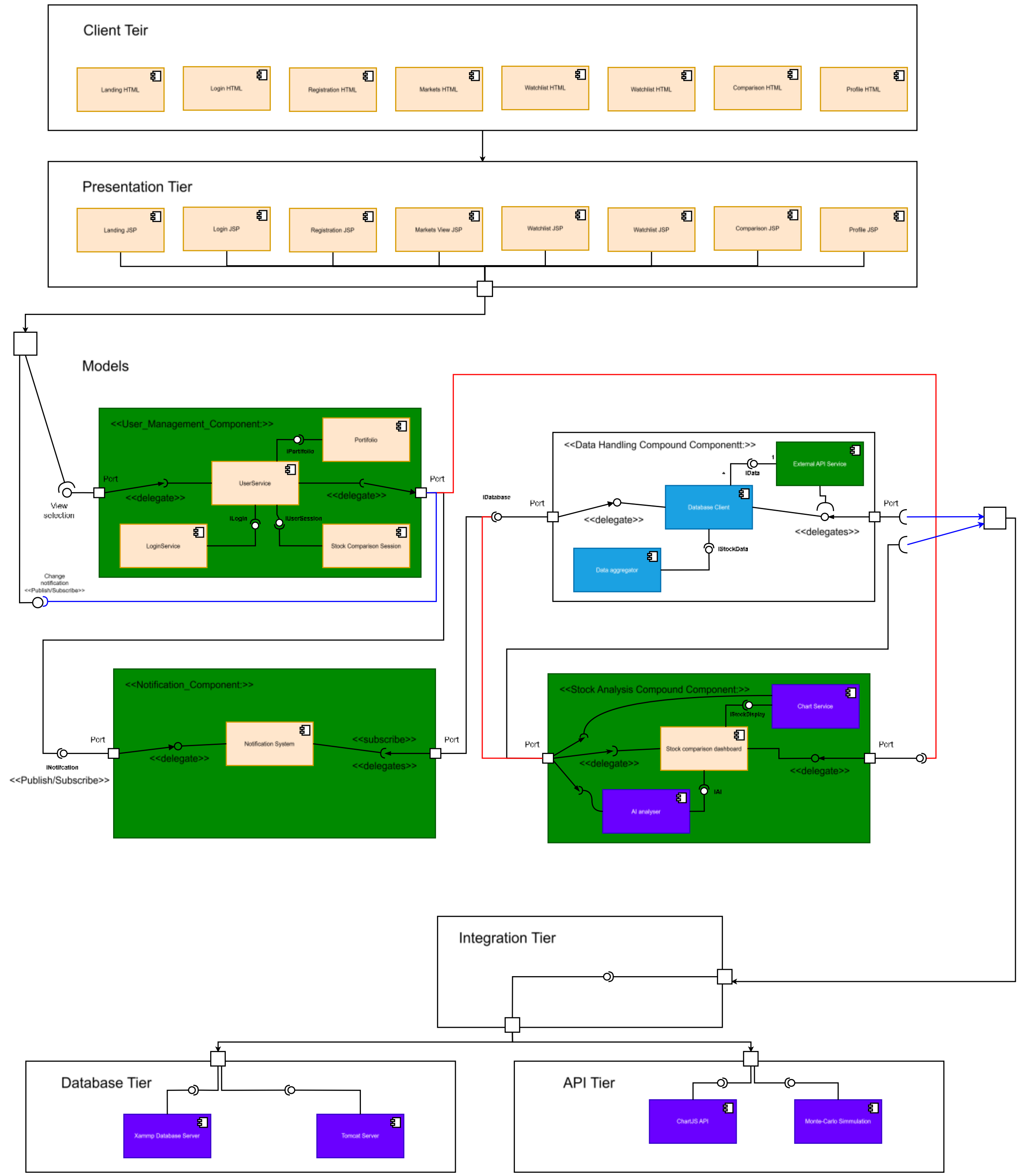
Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN

5 Tier Architecture Component Diagram:

Stock Shark

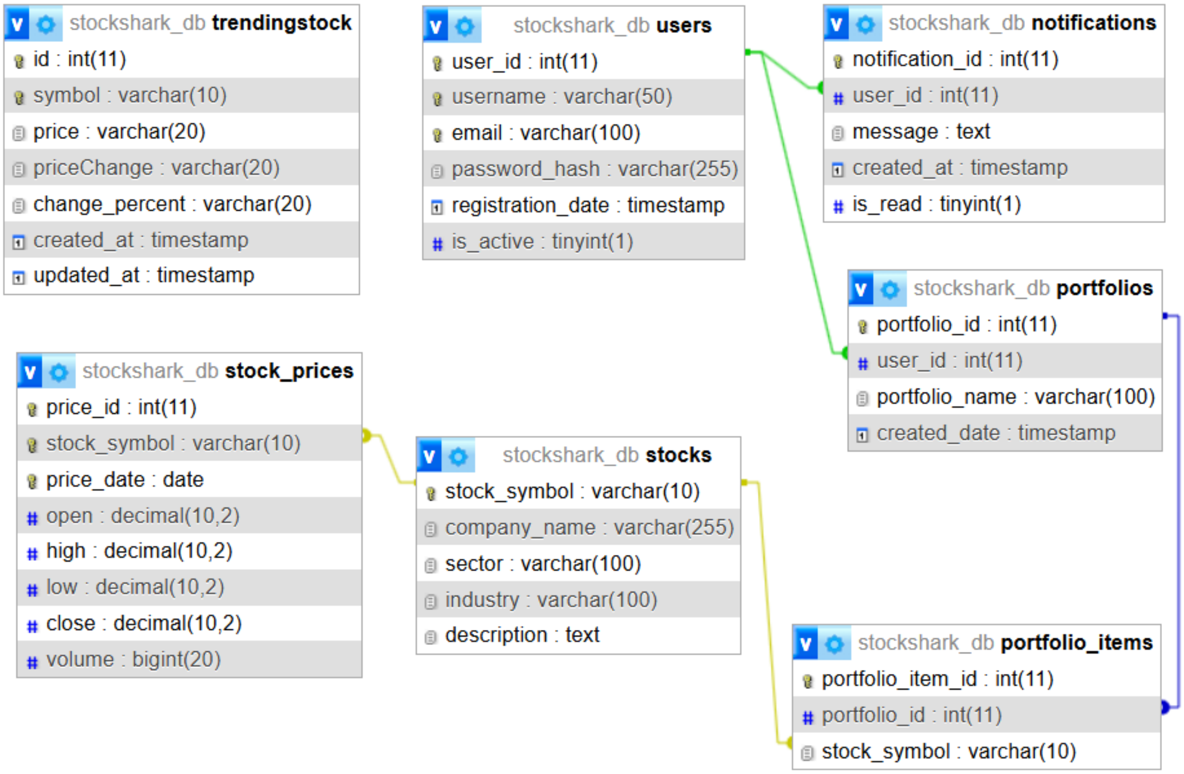
THE SOFTWARE ARCHITECTURE & DESIGN



How is this achieved?

The Database:

The database schema was designed to enforce data integrity and performance while supporting key functionalities like user management, stock data retrieval, and portfolio tracking. This design ensures that critical user data is stored securely and reliably, directly addressing user stories for privacy and accessibility.



stockshark_db **trendingstock**
- id : int(11)
- symbol : varchar(10)
- price : varchar(20)
- priceChange : varchar(20)
- change_percent : varchar(20)
- created_at : timestamp
- updated_at : timestamp

stockshark_db **users**
- user_id : int(11)
- username : varchar(50)
- email : varchar(100)
- password_hash : varchar(255)
- registration_date : timestamp
- is_active : tinyint(1)

stockshark_db **notifications**
- notification_id : int(11)
- user_id : int(11)
- message : text
- created_at : timestamp
- is_read : tinyint(1)

stockshark_db **portfolios**
- portfolio_id : int(11)
- user_id : int(11)
- portfolio_name : varchar(100)
- created_date : timestamp

stockshark_db **stock_prices**
- price_id : int(11)
- stock_symbol : varchar(10)
- price_date : date
- open : decimal(10,2)
- high : decimal(10,2)
- low : decimal(10,2)
- close : decimal(10,2)
- volume : bigint(20)

stockshark_db **stocks**
- stock_symbol : varchar(10)
- company_name : varchar(255)
- sector : varchar(100)
- industry : varchar(100)
- description : text

stockshark_db **portfolio_items**
- portfolio_item_id : int(11)
- portfolio_id : int(11)
- stock_symbol : varchar(10)

Key Tables:

- **users:** Stores user credentials and profile details.
- **stocks:** Contains static information about stocks.
- **stock_prices:** Maintains historical (and potentially real-time) pricing information.
- **portfolios:** Saves user-defined portfolios or watchlists.
- **portfolio_items:** Associates specific stocks with a given portfolio.
- **notifications:** (Planned) Will store user notifications.

Deployment:

The deployment environment was selected to balance ease of development with production readiness. XAMPP is used as a local MySQL development server, and Apache Tomcat serves as the web/application server, ensuring broad compatibility and robust performance.

Environment Details:

- **Database Server:**
  MySQL provided through XAMPP.
- **Application Server:**
  Apache Tomcat.
- **Build Tools:**
  Maven is used for building and packaging the application as a WAR file.
- **Configuration:**
  Database configuration details (host, port, database name, user credentials) are managed centrally via the DatabaseConfig class.

Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN

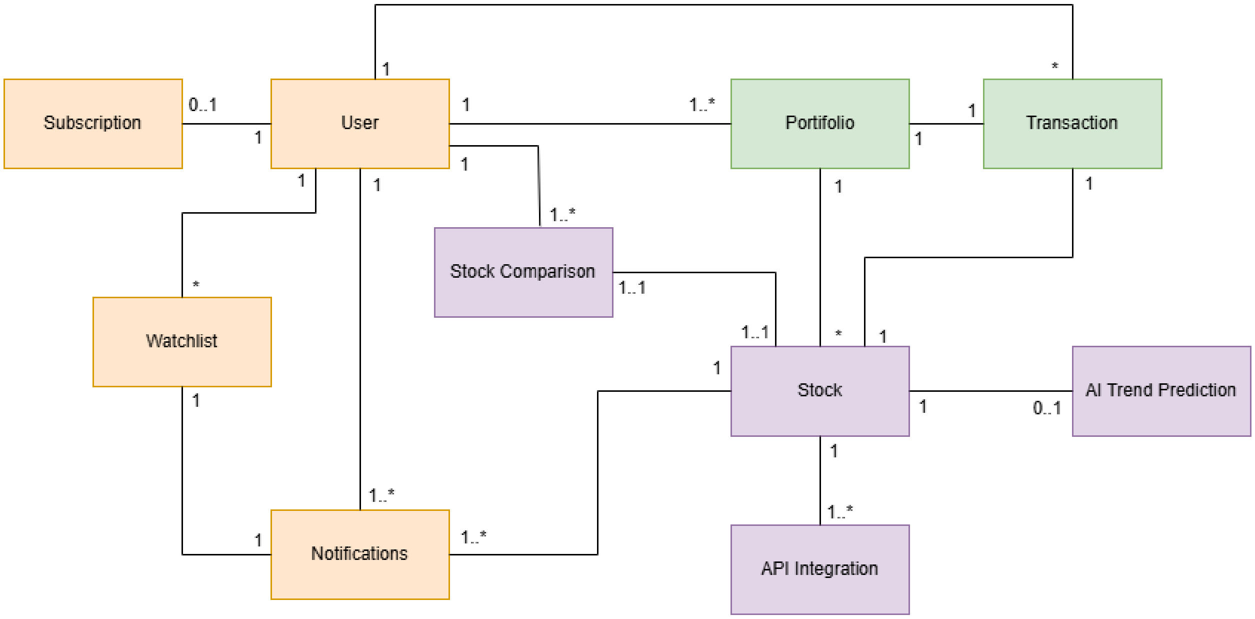
Building The Architecture

SockShark's architecture is built upon the core principles of Clean Architecture, which aims to create systems that are independent of frameworks, testable, and maintainable. This design follows the SOLID principles, making the system strong and easy to change.

- Single/Closed Principle (OCP): StockShark is designed to be open for extension but closed for modification. New features are added by introducing new code, rather than altering existing, stable code.
- Dependency Inversion Principle (DIP): High level modules, such as use cases that implement business logic, do not depend on low level modules, such as data access layers. Instead, low level modules depend on abstractions defined by high level modules.

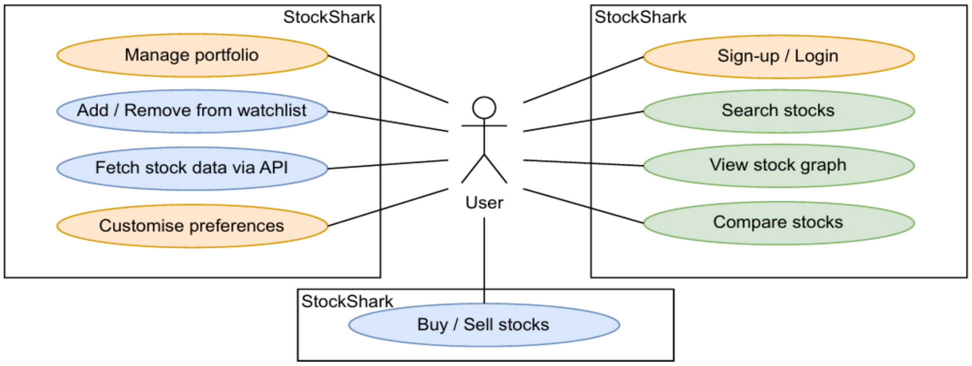
By separating these layers, we create a system that's flexible, maintainable, and easy to understand. This makes it easier for our team to add new features, fix bugs, and adapt to changing requirements, ultimately providing you with a better StockShark experience.

Business Concept Model



The Business Concept Model is structured to provide a user-centric, scalable, and modular system for managing stock investments and transactions. At its core, the User entity connects to essential functions like portfolios, transactions, subscriptions, watchlists, and notifications, ensuring seamless interaction with the platform. The Stock entity is deeply integrated with real-time market updates via API integration, AI trend predictions, and stock comparisons, allowing for data-driven decision-making. The model also enforces flexibility, allowing users to manage multiple portfolios and transactions while ensuring that each action is properly tracked. The relational structure supports future expansion, such as adding new AI models, additional investment tools, or automation, without disrupting existing functionalities. By balancing functionality and scalability, this design enables a robust financial system that caters to both beginner and advanced investors.

Use Case Model



The StockShark Use Case Diagram is structured to provide a clear and user-centric experience, dividing functionalities into three main categories: portfolio management, stock exploration, and trading transactions. Users can manage their investments by adding or removing stocks from their watchlist, customizing preferences, and fetching real-time stock data via API. The system also supports stock research through features like stock search, comparison, and historical graph visualization. Additionally, the buy/sell stocks functionality ensures seamless trading operations. The inclusion of sign-up/login enhances security and access control. This structured approach ensures scalability, allowing for future enhancements like AI-driven insights or automated trading while maintaining an intuitive and efficient user workflow.

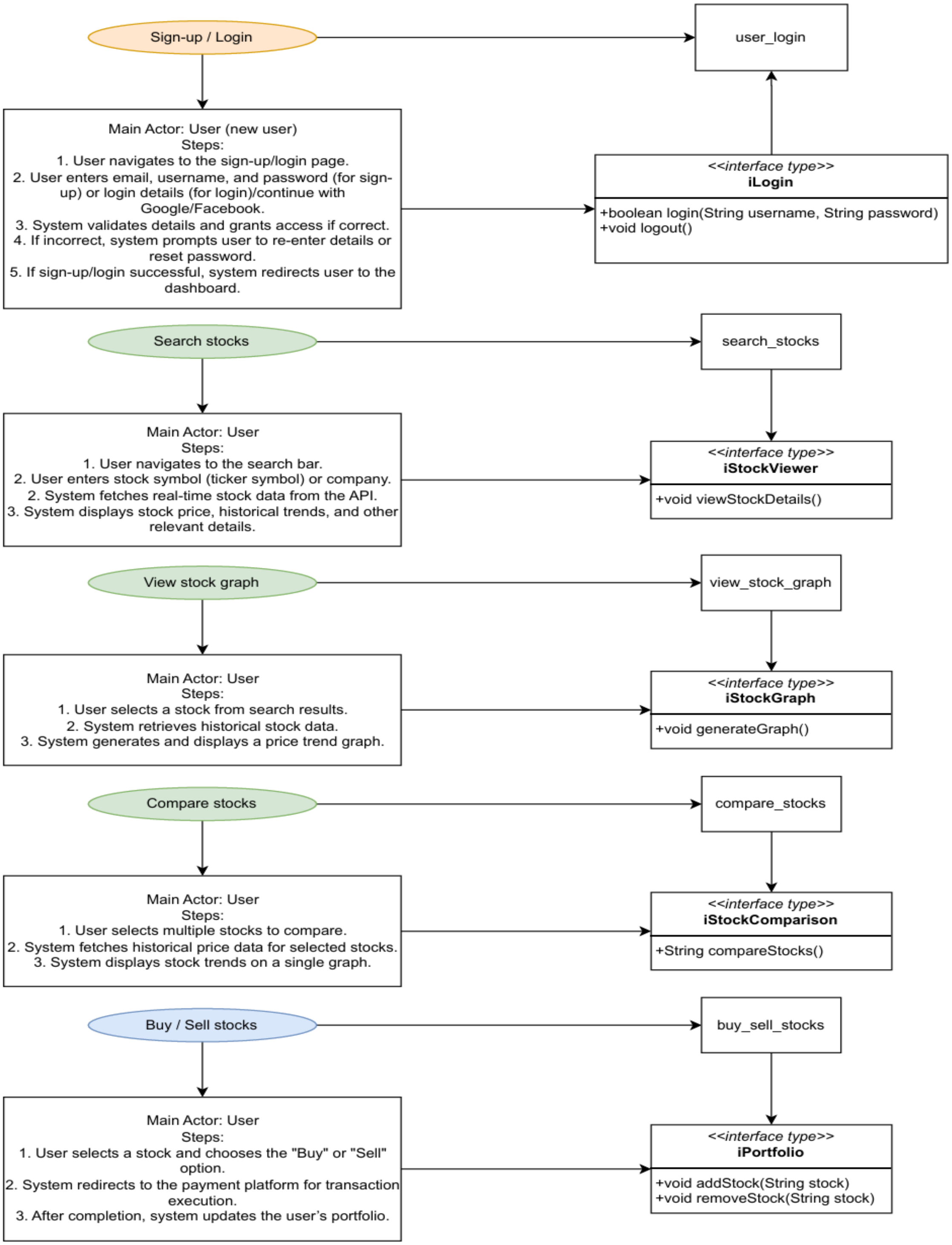
Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN

System Interfaces

Stock Shark

THE SOFTWARE ARCHITECTURE & DESIGN



Sign-up / Login

Main Actor: User (new user)
Steps:
1. User navigates to the sign-up/login page.
2. User enters email, username, and password (for sign-up) or login details (for login)/continue with Google/Facebook.
3. System validates details and grants access if correct.
4. If incorrect, system prompts user to re-enter details or reset password.
5. If sign-up/login successful, system redirects user to the dashboard.

user_login

<<interface type>>
iLogin
+boolean login(String username, String password)
+void logout()

Search stocks

Main Actor: User
Steps:
1. User navigates to the search bar.
2. User enters stock symbol (ticker symbol) or company.
2. System fetches real-time stock data from the API.
3. System displays stock price, historical trends, and other relevant details.

search_stocks

<<interface type>>
iStockViewer
+void viewStockDetails()

View stock graph

Main Actor: User
Steps:
1. User selects a stock from search results.
2. System retrieves historical stock data.
3. System generates and displays a price trend graph.

view_stock_graph

<<interface type>>
iStockGraph
+void generateGraph()

Compare stocks

Main Actor: User
Steps:
1. User selects multiple stocks to compare.
2. System fetches historical price data for selected stocks.
3. System displays stock trends on a single graph.

compare_stocks

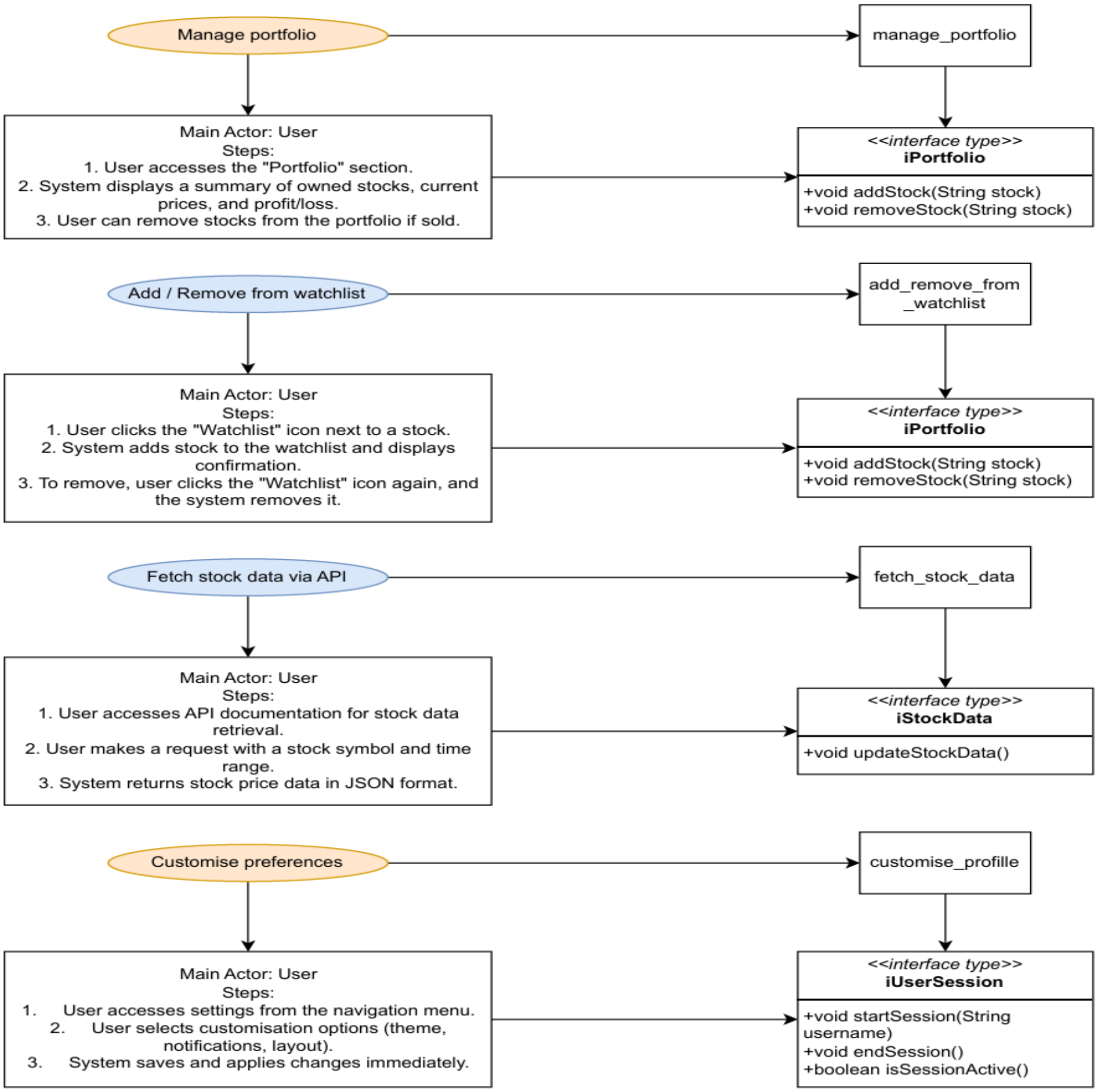
<<interface type>>
iStockComparison
+String compareStocks()

Buy / Sell stocks

Main Actor: User
Steps:
1. User selects a stock and chooses the "Buy" or "Sell" option.
2. System redirects to the payment platform for transaction execution.
3. After completion, system updates the user's portfolio.

buy_sell_stocks

<<interface type>>
iPortfolio
+void addStock(String stock)
+void removeStock(String stock)



Manage portfolio

manage_portfolio

Main Actor: User
Steps:
1. User accesses the "Portfolio" section.
2. System displays a summary of owned stocks, current prices, and profit/loss.
3. User can remove stocks from the portfolio if sold.

<<interface type>>
iPortfolio
+void addStock(String stock)
+void removeStock(String stock)

Add / Remove from watchlist

add_remove_from_watchlist

Main Actor: User
Steps:
1. User clicks the "Watchlist" icon next to a stock.
2. System adds stock to the watchlist and displays confirmation.
3. To remove, user clicks the "Watchlist" icon again, and the system removes it.

<<interface type>>
iPortfolio
+void addStock(String stock)
+void removeStock(String stock)

Fetch stock data via API

fetch_stock_data

Main Actor: User
Steps:
1. User accesses API documentation for stock data retrieval.
2. User makes a request with a stock symbol and time range.
3. System returns stock price data in JSON format.

<<interface type>>
iStockData
+void updateStockData()

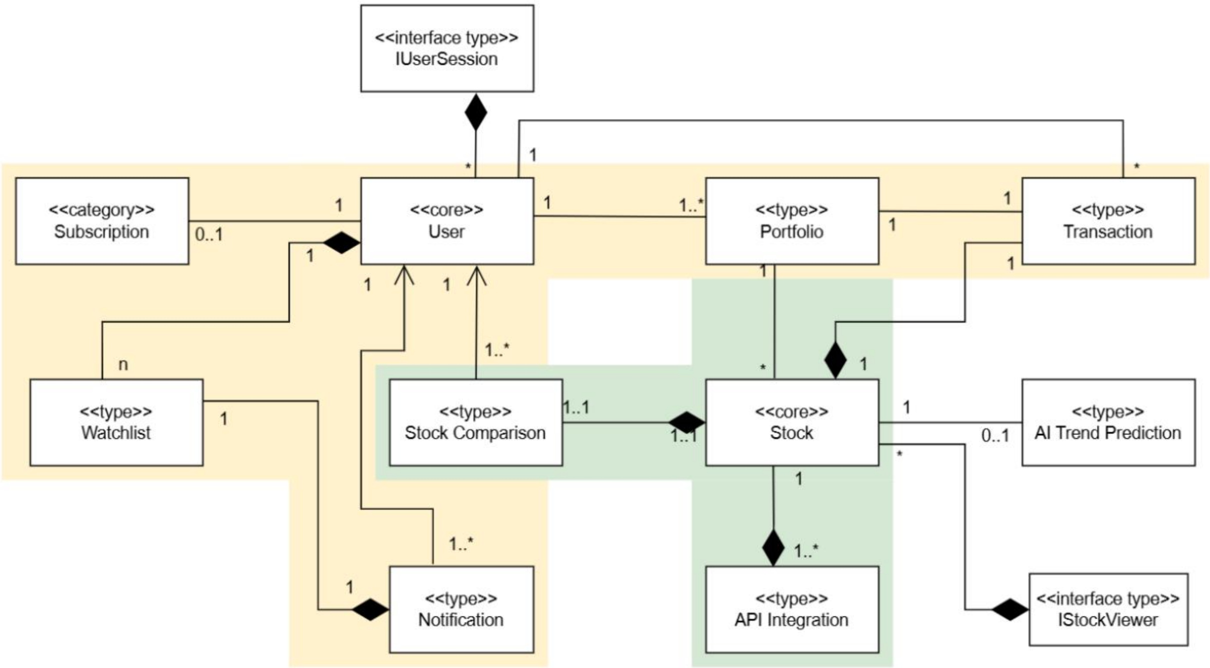
Customise preferences

customise_profille

Main Actor: User
Steps:
1. User accesses settings from the navigation menu.
2. User selects customisation options (theme, notifications, layout).
3. System saves and applies changes immediately.

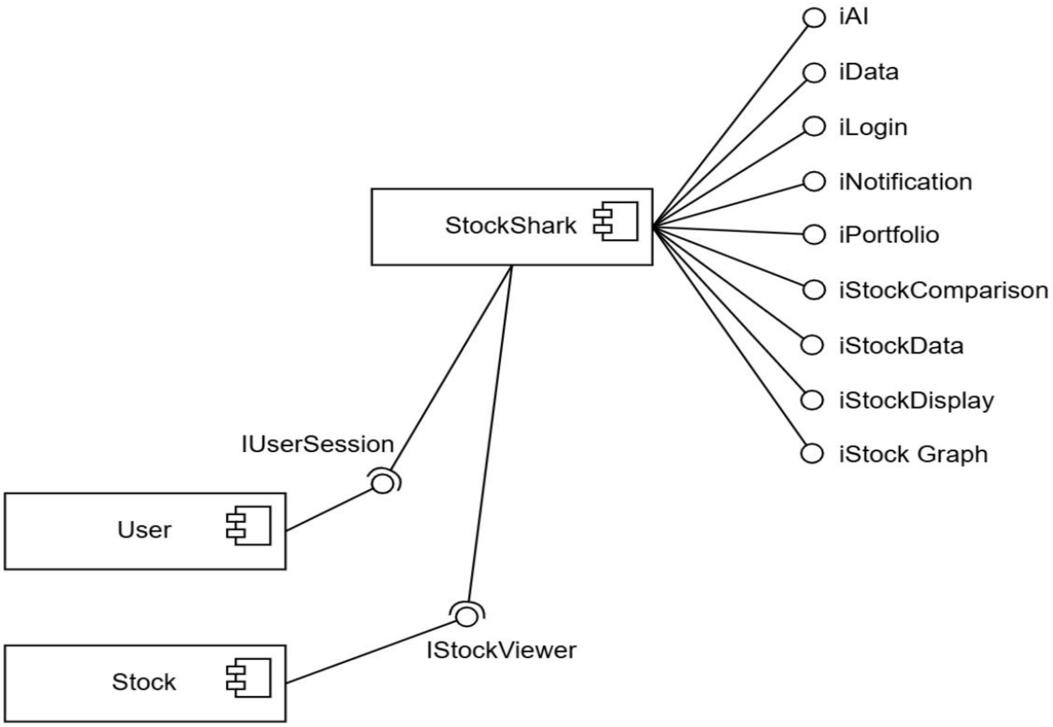
<<interface type>>
iUserSession
+void startSession(String username)
+void endSession()
+boolean isSessionActive()

Business Type Model



The Business Type Model in the diagram is structured to represent a scalable and modular stock trading system with distinct core entities, functional types, and interfaces. At the center, the User entity acts as the primary connection point, interacting with subscriptions, portfolios, transactions, notifications, and watchlists. The model ensures that users can manage their investments efficiently while maintaining real-time stock tracking through API integrations. The Stock entity, another core component, connects to stock comparison, AI trend prediction, and API integration, ensuring data-driven decision-making. Additionally, interface types like IUserSession and IStockViewer support session management and stock visualization, enhancing the platform’s usability. The composition relationships (diamonds) indicate ownership structures, ensuring that key elements like portfolios, watchlists, and notifications are user-specific, while stocks exist independently. This model is designed for flexibility and expansion, allowing future enhancements such as automated trading, advanced analytics, or AI-driven insights without disrupting the core architecture.

Initial System Architecture



The Initial System Architecture diagram represents the foundational structure of the StockShark platform, showcasing key system components and their interactions. At the core, StockShark serves as the central system, connecting various interfaces such as iAI, iData, iLogin, iNotification, iPortfolio, iStockComparison, iStockData, iStockDisplay, and iStockGraph, ensuring modularity and extendability. The User entity is linked through the IUserSession interface, allowing authentication and session management, ensuring secure access to the system. Meanwhile, the Stock entity interacts with StockShark via the IStockViewer interface, enabling real-time stock data retrieval and visualization. This architecture provides a clear separation of concerns, ensuring scalability, maintainability, and integration of advanced functionalities like AI-based trend analysis and stock comparison. The structured design allows for future enhancements, such as automated trading or additional analytics, while keeping the system flexible and user-friendly.

Project Management

Code-Related Deliverables

Implementation of Clean Architecture Principles

- **Single Responsibility Principle (SRP):**
  Each class in the StockShark project is designed to have one responsibility. For example, servlets only manage HTTP requests/responses, while separate classes handle business logic and database interactions. This separation ensures that changes in one area (such as the API integration) do not impact others (e.g., user authentication).
- **Open/Closed Principle (OCP):**
  The system is built to be extendable without modifying existing code. For instance, new features can be added by extending base classes or interfaces rather than altering core logic. This allows us to implement future enhancements such as advanced chart interactivity and AI-based predictions without impacting current functionality.
- **Liskov Substitution Principle (LSP):**
  All subtypes and implementations are interchangeable with their base types. This means that classes implementing interfaces (like our Data_Handling_Component) can be substituted without altering the correctness of the program. This facilitates unit testing and future maintenance.
- **Interface Segregation Principle (ISP):**
  Clients are not forced to depend on methods they do not use. Interfaces in the project (for example, those defining user management functions and data handling operations) are focused and minimal. This ensures that each component only implements what is necessary, enhancing readability and maintainability.
- **Dependency Inversion Principle (DIP):**
  High-level modules do not depend on low-level modules; both depend on abstractions. The design uses abstract interfaces for key components (e.g., for database operations and API services), allowing us to change implementations without modifying the core business logic. This increases flexibility and supports unit testing by allowing dependency injection.

Component and Interface Implementation

- The **Database Client** (formerly DAO) encapsulates all database operations, ensuring that all SQL interactions are centralized and changeable without affecting business logic.
- **User Management and Data Handling Components** abstract complex operations (like authentication, portfolio management, and API integration) behind clean interfaces. These components satisfy the principles above while delivering modular and testable code.

Team Management

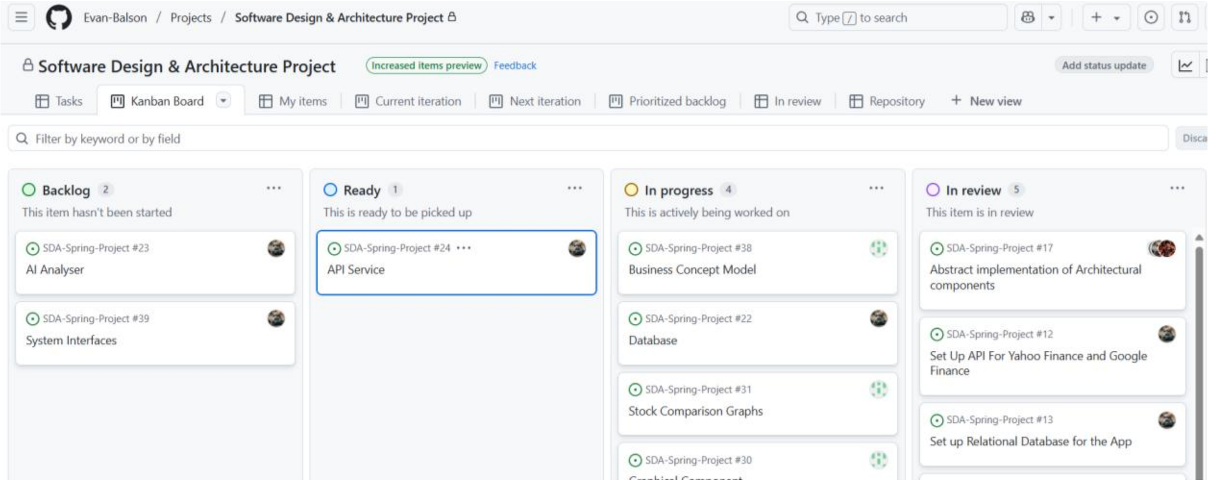
Team Coordination

- **Team Leader: Evan Balson**
  - **Responsibilities:** Directed Sprint 2 tasks, monitored progress, and ensured timely completion.
  - **Focus Areas:** System Interfaces, composite component diagrams, high-fidelity wireframes, and coding implementation for the Database, AI Analyzer, and API Service.
- **Scrum Master: Maahia Rahman**
  - **Responsibilities:** Scheduled weekly team meetings, managed the Business Concept Model, and created low-fidelity wireframes.
  - **Focus Areas:** Coding implementation for the Graphical Component and Stock Comparison Graph.
- **Mahbouba:**
  - **Responsibilities:** Managed the Business Type Model and participated in the documentation process.
  - **Focus Areas:** Coding implementation for the Notification System and Stock Comparison Session.
- **Hala:**
  - **Responsibilities:** Oversaw Use Case development, Initial Architecture, and defined Business Interfaces.
  - **Focus Areas:** Coding implementation for Portfolio, Login/Logout, and User modules. Also contributed to project documentation.

Process

- **Task Distribution:**
  Tasks were divided among team members based on individual strengths and preferences to maximize efficiency and quality.
- **Code Review and Integration:**
  To avoid disruptions in the main branch, the team used pull requests to review code changes before merging them. This process helped ensure that new implementations adhered to our architectural principles and were properly tested.

GitHub Updates:



Software Design & Architecture Project

Backlog 2
This item hasn't been started

- SDA-Spring-Project #23 — AI Analyser
- SDA-Spring-Project #39 — System Interfaces

Ready 1
This is ready to be picked up

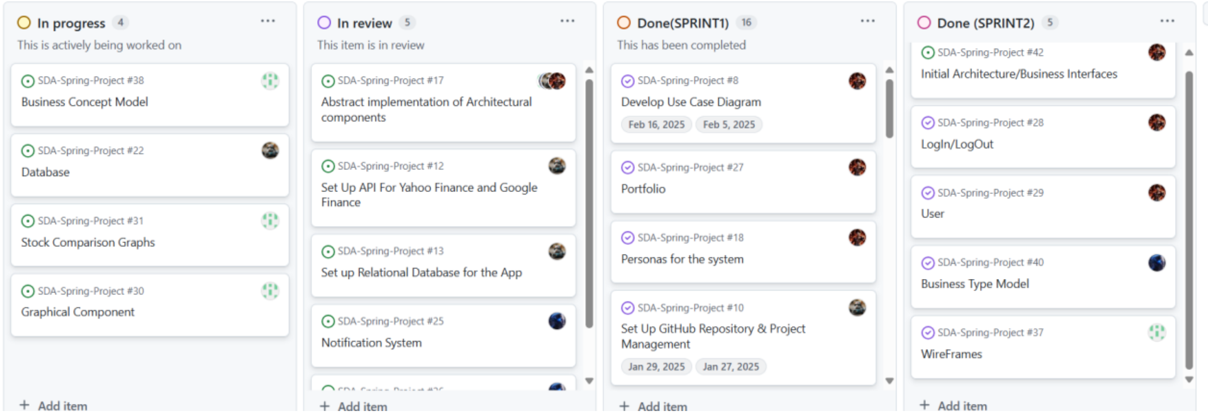
- SDA-Spring-Project #24 — API Service

In progress 4
This is actively being worked on

- SDA-Spring-Project #38 — Business Concept Model
- SDA-Spring-Project #22 — Database
- SDA-Spring-Project #31 — Stock Comparison Graphs
- SDA-Spring-Project #30

In review 5
This item is in review

- SDA-Spring-Project #17 — Abstract implementation of Architectural components
- SDA-Spring-Project #12 — Set Up API For Yahoo Finance and Google Finance
- SDA-Spring-Project #13 — Set up Relational Database for the App



In progress 4
This is actively being worked on

- SDA-Spring-Project #38 — Business Concept Model
- SDA-Spring-Project #22 — Database
- SDA-Spring-Project #31 — Stock Comparison Graphs
- SDA-Spring-Project #30 — Graphical Component

+ Add item

In review 5
This item is in review

- SDA-Spring-Project #17 — Abstract implementation of Architectural components
- SDA-Spring-Project #12 — Set Up API For Yahoo Finance and Google Finance
- SDA-Spring-Project #13 — Set up Relational Database for the App
- SDA-Spring-Project #25 — Notification System

+ Add item

Done(SPRINT1) 16
This has been completed

- SDA-Spring-Project #8 — Develop Use Case Diagram — Feb 16, 2025 — Feb 5, 2025
- SDA-Spring-Project #27 — Portfolio
- SDA-Spring-Project #18 — Personas for the system
- SDA-Spring-Project #10 — Set Up GitHub Repository & Project Management — Jan 29, 2025 — Jan 27, 2025

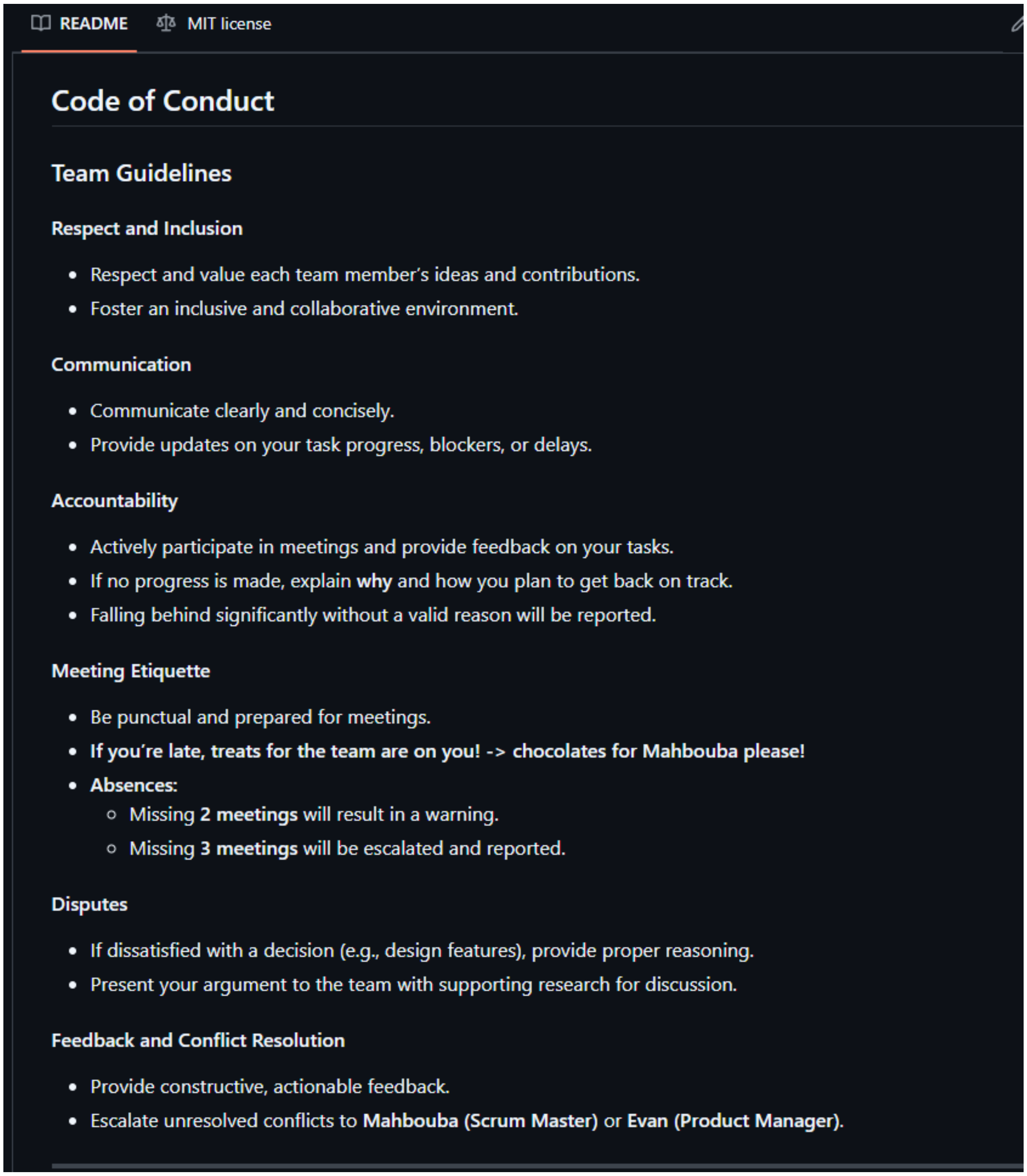
+ Add item

Done (SPRINT2) 5

- SDA-Spring-Project #42 — Initial Architecture/Business Interfaces
- SDA-Spring-Project #28 — LogIn/LogOut
- SDA-Spring-Project #29 — User
- SDA-Spring-Project #40 — Business Type Model
- SDA-Spring-Project #37 — WireFrames

+ Add item

Team Code of Conduct



Code of Conduct

Team Guidelines

Respect and Inclusion

- Respect and value each team member's ideas and contributions.
- Foster an inclusive and collaborative environment.

Communication

- Communicate clearly and concisely.
- Provide updates on your task progress, blockers, or delays.

Accountability

- Actively participate in meetings and provide feedback on your tasks.
- If no progress is made, explain **why** and how you plan to get back on track.
- Falling behind significantly without a valid reason will be reported.

Meeting Etiquette

- Be punctual and prepared for meetings.
- **If you're late, treats for the team are on you! -> chocolates for Mahbouba please!**
- **Absences:**
  - Missing **2 meetings** will result in a warning.
  - Missing **3 meetings** will be escalated and reported.

Disputes

- If dissatisfied with a decision (e.g., design features), provide proper reasoning.
- Present your argument to the team with supporting research for discussion.

Feedback and Conflict Resolution

- Provide constructive, actionable feedback.
- Escalate unresolved conflicts to **Mahbouba (Scrum Master)** or **Evan (Product Manager)**.

References:

[1]. GeeksforGeeks, *“Complete Guide to Clean Architecture”, [Online], Available:* Complete Guide to Clean Architecture - GeeksforGeeks, [Accessed: March. 16, 2025]

[2] Yahoo Finance, "Yahoo Finance," [Online]. Available: https://finance.yahoo.com. [Accessed: Apr. 16, 2025].

[3] Alpha Vantage, "Alpha Vantage Documentation," [Online]. Available: https://www.alphavantage.co/documentation/. [Accessed: March. 16, 2025].

[4] Chart.js, "Chart.js Documentation," [Online]. Available: https://www.chartjs.org/docs/latest/. [Accessed: March. 16, 2025].

[5] Apache Tomcat, "Apache Tomcat Documentation," [Online]. Available: https://tomcat.apache.org/tomcat-9.0-doc/. [Accessed: March. 16, 2025].

[6] XAMPP, "XAMPP Documentation," [Online]. Available: https://www.apachefriends.org/index.html. [Accessed: March. 16, 2025].