YANE-STOCK

Key Requirements for Our Bitcoin Project

What Must the System Do?

- The system should handle sending and receiving Bitcoin transactions via the API.
- It should stay up-to-date and synchronize with the current blockchain to ensure the latest blocks and transactions are reflected.
- Extra (Security): Protect Bitcoin data and private keys using encryption.
- User Interface: In later sprints, provide a robust, user-friendly interface for interaction.
- It should handle many users and transactions efficiently.

Scope: What Will We Implement for Sprint 1

- Classes: stock, data, user, user_finance, user_owned_stocks.
- API Connection: Learn how to connect to the API in later sprints. For now, having the foundation is sufficient.
- Repository Setup: Ensure branches and repositories are set up correctly.
- Documentation and Planning: Complete documentation and planning before coding, then move to diagrams.
- Task Assignment: Assign tasks to group members (e.g., Adam will create the code of conduct).
- Other Documentation: Assign additional documentation tasks to different group members.

Functional Requirements

1. Wallet Creation and Management
    - Users can create a Bitcoin wallet with a public/private key pair and securely store the private key.
        - Key Challenge: Ensuring private keys are securely stored and recoverable using a seed phrase.
2. Sending Bitcoin Transactions
    - Users can send Bitcoin to others, with transactions signed by their private key and broadcast to the network.
        - Key Challenge: Correctly calculating transaction fees and ensuring transactions are broadcasted without issues.
3. Receiving Bitcoin
    - Users can generate unique Bitcoin addresses to receive payments.
        - Key Challenge: Ensuring addresses are valid and managing address generation securely.
4. Viewing Transaction History

- Users can view their transaction history, including details like sender, receiver, and status.
    - Key Challenge: Efficiently fetching and displaying real-time blockchain data.

Non-Functional Requirements

1. Security
    - Sensitive data (e.g., private keys) must be encrypted and stored securely.
        - Key Challenge: Preventing unauthorized access and attacks like man-in-the-middle.
2. Performance
    - The system should process transactions quickly and interact with the Bitcoin network with minimal delays.
        - Key Challenge: Optimizing the app for low-latency interactions with the blockchain.
3. Scalability
    - The system should support more users and transactions as the user base grows.
        - Key Challenge: Ensuring the backend can handle increasing data and user load.
4. Reliability
    - The system should function without failures, even under high load, and recover from errors gracefully.
        - Key Challenge: Ensuring high availability and managing transaction failures.

Technical Requirements

1. Blockchain Integration
    - The system interacts with the Bitcoin blockchain to send/receive transactions and check balances.
        - Key Challenge: Choosing between using third-party APIs or running a lightweight Bitcoin node.
2. Data Storage
    - Wallet information and transaction history must be securely stored locally.
        - Key Challenge: Secure storage of private keys while allowing easy access to wallet data.
3. User Authentication
    - Users should authenticate to access their wallet (optional).
        - Key Challenge: Implementing secure yet simple authentication methods.

Possible Risks and Mitigation

1. Risk: Private Key Exposure or Loss
    - Mitigation: Encrypt private keys and guide users on backing them up securely.
2. Risk: Network Connectivity Issues During Transactions
    - Mitigation: Implement retry mechanisms and provide real-time feedback on transaction status.

Assumptions and Constraints
Assumptions

- Users will have an internet connection to interact with the Bitcoin network.
- The system won't run a full Bitcoin node but will rely on third-party APIs.

Constraints

- Limited infrastructure resources, so public APIs will be used with potential rate limits.
- The app should be compatible across desktop and mobile devices but might have platform-specific limitations.

Scope

Data retrieval using API

Data Storage using SQL

Charting features - comparison graphs

Basic UI accepting input

Offline mode

Key Architectural Concepts •Components–Computation/data

- Connectors–Information interchange
- Configurations–Instantiate components
- connectors in particular arrangements

Key Requirements for our Bitcoin Project

Scope: What will we implement for sprint 1

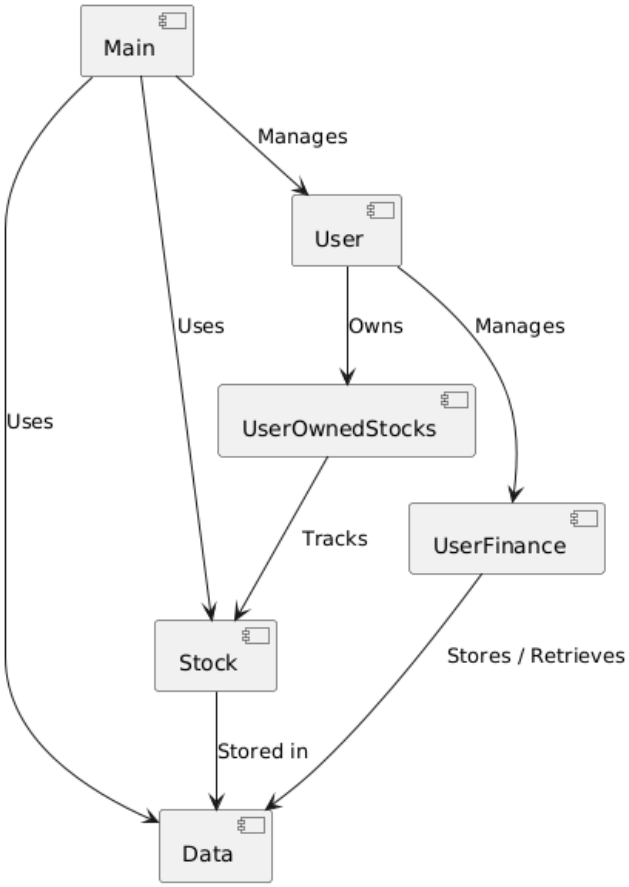
- Classes: stock, data, user, user_finance, user_owned_stocks we will change up the names but use these for now.
- Connect the API, we will learn how to do this in later sprints. Having the foundation for now is fine.
- Ensure the branches and repositories are set up correctly.
- Have the documentation and planning done before we code then move onto the diagrams.
- We assign tasks for example Adam creating the code of conduct.
- Other documentations will be assigned to different group members.

Requirement Identification

Branching Summary for Sprint 1:

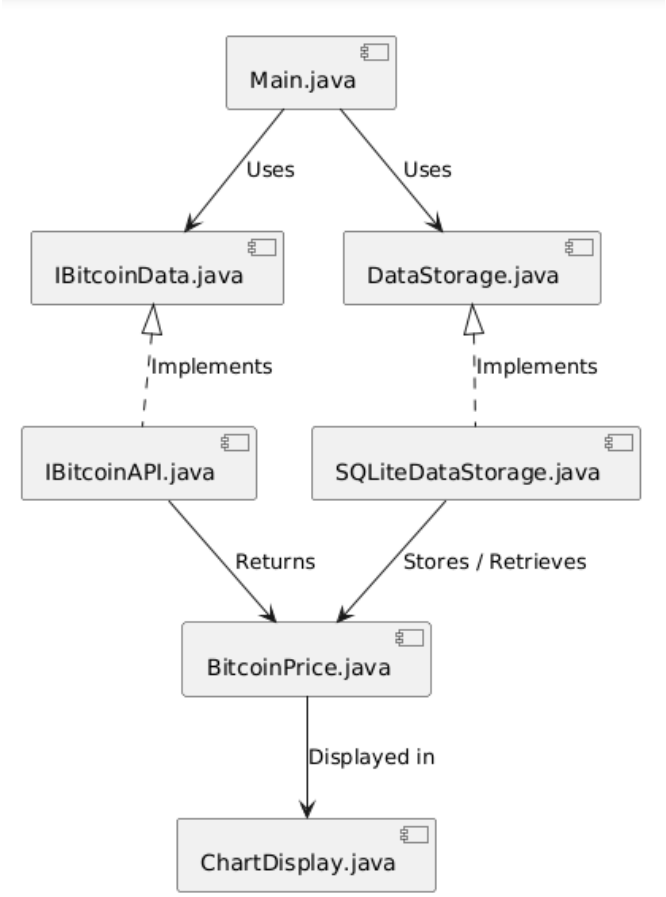
1. main — The stable, production-ready branch.
2. develop — The working branch where all Sprint 1 tasks will be merged.
3. feature/project-setup — For setting up the GitHub project and initial configurations.
4. feature/requirements-scope — For documenting and detailing the requirements and project scope.
5. feature/architectural-design — For creating the component specification diagram and architectural principles.

The project is about creating a simple Bitcoin wallet application that lets users create wallets, send and receive Bitcoin, and view transaction history. The app will securely store private keys and interact with the Bitcoin network to process transactions. The main goal is to build a secure and easy-to-use system for managing Bitcoin, ensuring that users can safely send and receive coins while keeping their information protected.



- Main Class:
- It is the entry point of the program.
- Uses → `Stock` and `Data` directly.
- Manages → `User`.
- User Class:
- Owns → `UserOwnedStocks`.
- Manages → `UserFinance`.
- UserOwnedStocks Class:
- Tracks → `Stock`.

- UserFinance Class:
- Stores/Retrieves → Data.
- Stock Class:
- Stored in → `Data`.
- Data Class: Serves as the storage unit for both `Stock` and `UserFinance`.



- Main.java
- The entry point of the application.
- Uses → IBitcoinData.java (To fetch Bitcoin price data).
- Uses → DataStorage.java (To store/retrieve Bitcoin price data).
- IBitcoinData.java (Interface)
- Defines a contract for fetching Bitcoin data.
- Implemented by → IBitcoinAPI.java.
- IBitcoinAPI.java
- The concrete implementation of IBitcoinData.java.
- Returns → BitcoinPrice.java (Contains the latest Bitcoin price).
- DataStorage.java (Interface)
- Defines methods for storing and retrieving data.
- Implemented by → SQLiteDataStorage.java.
- SQLiteDataStorage.java

- Handles storing and retrieving Bitcoin price data in a SQLite database.
- Stores/Retrieves → BitcoinPrice.java.
- BitcoinPrice.java
- Stores Bitcoin price data obtained from IBitcoinAPI.java.
- Displayed in → ChartDisplay.java
- ChartDisplay.java
- Displays Bitcoin price data visually.
- Receives data from → BitcoinPrice.java.

What We Will Implement In the Future

These diagrams demonstrate how we have used certain design components to maintain our goals as best as we can, for example, we have used encapsulation in order to keep different components of the project separate giving it more ability to have further additions in the future, like when we integrate an API or when we need to further add layers like more complex storage into our later stages.