CRYPTOSTOCK DOCUMENTATION
COMP.SE.110

27.10.2024

Project description:

CryptoStock is a web application that displays the cryptocurrencies and stocks to the user. It allows the user to compare any stock against any cryptocurrency available in the app. User also can save various stocks or cryptocurrencies as their favorite for their convenience.

Frontend Architecture

The CryptoStock frontend is a component-based React application, which interacts with a Java-based backend to retrieve and display stock and cryptocurrency data. As the project follows the MVC architecture, the React frontend is used to manage the View layer of the design pattern. In detail, React, JavaScript, HTML, CSS, SASS & Bootstrap are combined and applied into the implementation process for quick development, allowing us to reach the “MVP stage” quickly enough to meet the courses’ deadlines while still achieving expected output. Diving deep into the architectural patterns, we used the atomic pattern, which helped us break the components into smaller ones. That helped us achieve better separation of concern and made the code easier to read and debug for each component. We also used the render prop pattern, so that we could send data from one component to another for various tasks, such as in the comparison page(the graph component on the page, to be specific).The diagram of Frontend architecture was designed to visualise the structure and connection between components intuitively, which can be found here.

Technologies Used

- **React:** used as component-based architecture, allowing modular and reusable UI components. This helps to manage the View layer of CryptoStock application.

- **JavaScript:** A natural “supplement” to React as a primary scripting language to build dynamic and interactive features.
- **HTML:** provides the foundation structure of the UI.
- **CSS and SASS:** the CSS will be used to style the web application, while SASS extends CSS to include variables and nested rules to simplify the styling process. This provides more flexibility in working with CSS.
- **Bootstrap:** offers several advantages for web development. It provides a variety of built-in components like Button, FormControl, and Modal, supporting in speeding up the design implementation time. Additionally, the responsive grid system helps us to ensure the dynamic design on different devices.

Core components

1. Authentication
    - In the authentication process, there are two components: login and register pages (components). These components were determined as important parts of the application, enhancing user experience, and improving system’s security.
    - The login and register components will communicate with the backend for validation and token handling. This would be done by using REST API to fetch and update data, which results in sending requests to /login and /register endpoints in this case.
    - Overall, these pages will handle the user input and send authentication data to the backend side. The validation token can be saved in local storage for session management.
2. Profile page
    - This page displays the user-specific data and allows users to make changes on their personal data. There are two separate sections on the page as settings and preferences. The users can view their information and possible actions in the settings parts, while they can find their favourite stocks and cryptocurrencies in the preferences area.

- For better management, three different components were created to make the project easier in finding bugs and developing processes. Via REST API, the frontend can make calls to retrieve updated user info from the backend. In this page, the endpoints would be /user/{id}/info and user/{id}/preferences.

3. Comparison page(work in progress)
   - This component provides users with the opportunity to compare stocks and cryptocurrencies. It contains interactive elements such as dropdown menus and charts. The page was divided into two sections, one is the comparison function and other is the graph area. This allows users to select and compare assets, resulting in a comprehensive chart with detailed data of the comparing items.
   - The data of stocks and cryptocurrencies here are also fetched and retrieved from the backend based on user selections. The requests will be sent from client side to the server for getting the financial data with the endpoints: /stocks, /stocks/{stockid}, /coins, /coins/{coinid}. The retrieved data will be passed into the Graph component to generate a chart and render visual data to the user.
4. Preferences(work in progress)
   - This core component allows users to mark stocks and cryptocurrencies as favourites, which will be saved in the backend for future access. In specific pages such as Stocks page or Profiles page, the favourite assets will be shown as a part of the page, offering quick access, and improving user experience.
   - The main functionalities of this component enable users to add or remove favourite stocks and cryptocurrencies. The data is retrieved and illustrated in a list of favourite items' icons. Any changes made by users will be updated in the backend with the endpoint /preferences.

Boundary and interface

- Information flows: in this system, the frontend communicates with the backend via REST API endpoints. This allows the application to fetch and update the data between client side and server side. In the frontend architecture, different pages or components will send requests to specific endpoints:
    - Authentication: Login/Register page sends requests to /login and /register.
    - Stock and Crypto data: these data are used in various pages with requests going to /stocks, /stocks/{stockid}, /coins, /coins/{coinid}.
    - Preferences: the favourite items will be fetched and updated with the /preferences endpoint.
- The communication between components will be handled by using different methods. The props and state will be implemented to pass data between parent and child components. For global state, the Redux management library can be utilised to handle the application state, enhancing the development process as well as debugging stage. However, due to time constraints and the size of the project, we made a decision not to implement Redux in the current stage. This means the library can be applied into the project later in the future if needed.
- Frontend interface:
    - User authentication interface (Login page, registration page): this facilitates login/signup process and verifies user credentials.
    - Data display interface (Stocks page, cryptos page, comparison page, profile page): this interface helps to manage data from backend endpoints with different methods (GET, POST) to retrieve and display on pages.

- Chart (Graph) interface: this integrates with chart libraries (chart.js, react-chartjs-2) to visually represent stock and cryptocurrency, either individually or in comparison.

Prototype

A figma prototype of CryptoStock can be found here.

Backend Architecture

The CryptoStock backend is a monolithic Java application designed using the Model-View-Controller (MVC) architectural pattern. It manages stock and cryptocurrency data, user accounts, and statistical analysis. The backend will be containerized using Docker for seamless deployment and scalability.

Technologies Used

- **Java**: Backend logic and controller development.
- **Spring Boot java server**: Handles HTTP requests and provides the server core with corresponding utils such as Jackson for handling JSON requests, RESTful Web Services, Exception handling, etc.
- **PostgreSQL**: database for storing user info
- **Docker**: Containerization for deployment and portability.
- **JSON (Jackson)**: For data serialization
- **External APIs**:
    - AlphaVantage - an API that provides up-to-date stock data
    - Coin API - an API that provides up-to-date cryptocurrencies data

Core Components

1. **Controllers Layer (Controller)**
    - Handles HTTP requests, managing the routing of API calls for stocks, cryptocurrencies, user accounts, and statistics. It serves data to the React frontend through defined REST endpoints. Currently, the app has 4 controllers:
        i. CoinController. It implements /coins, /coins/{id}, and /coins/{id}/charts endpoints which are responsible for returning cryptocurrency data. To provide the data the

controller uses the CoinStatService model instance that fetches and handles data from the Coin API

ii. StatisticsController. It implements /comparisonChart endpoint which is responsible for returning chart data for requested cryptocurrency and stock instances. Currently, the data is mocked in the MockDataService model instance which is used by the controller, but in the final app version it will be requesting data from Coin and Stock models and combining them into a ComparisonChart interface to return to the user.

iii. StockController. It provides /stocks, /stocks/{symbol}, /stockNames and /stocks/{symbol}/history endpoints which are responsible for returning stock data. To provide the data the controller uses the StockStatService model instance that fetches and handles data from the AlphaVantage API

iv. UserController. It provides /login, /register, /preferences, and /user endpoints that allow to authenticate the user and allow to save data. To access /preferences and /user endpoints the user should have a valid token that is returned from the /login endpoint, otherwise, the data will not be provided and 403 HTTP code will be returned. UserService model instance is used by the controller for the user's data manipulation

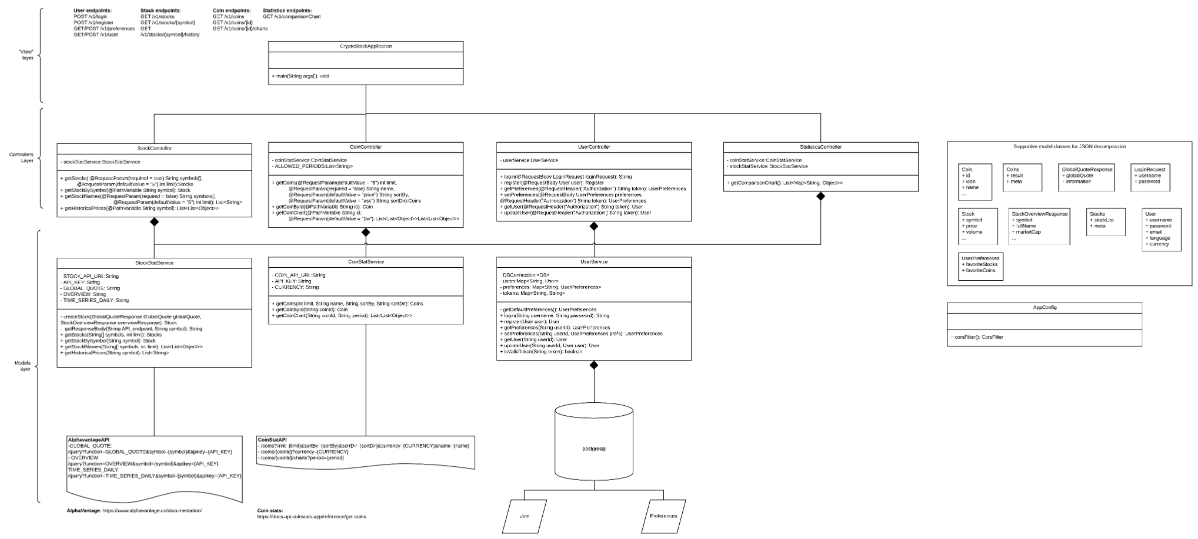
2. **Services Layer (Model)**
   - Communicates with the database (which is not implemented for the MVP) and external APIs, handling data fetching, storage, and manipulation. It processes stock, cryptocurrency, and user data before passing it to the Controller. Currently, no database instance is created and all the data is stored inside JSON data structures in the app itself however for the final version of the app the database will be implemented. The app has 4 main models:
     i. CoinStatService. The service handles cryptocurrency API fetching and its data manipulation
     ii. MockDataService. The service handles mock data requested by StatisticsController that is not fully implemented for the MVP phase yet

iii. StockStatService. The service handles stock API fetching and its data manipulation

iv. UserService. The service handles user data manipulation.

3. **Supportive services:**
    - Data structures classes. The classes that use the Jackson utility to handle JSON structures for incoming and upcoming requests
    - AppConfig class. The singleton class that specifies Spring Boot server configs such as Cross-origin resource sharing policies (CORS).

Here is the detailed schema of the backend architecture which shows in more details how information flows between components:


"View" layer
Controllers Layer
Models layer
User endpoints:
POST /v1/login
POST /v1/register
GET/POST /v1/preferences
GET/POST /v1/user
Stock endpoints:
GET /v1/stocks
GET /v1/stocks/{symbol}
GET /v1/stocks/{symbol}/history
Coin endpoints:
GET /v1/coins
GET /v1/coins/{id}
GET /v1/coins/{id}/charts
Statistics endpoints:
GET /v1/comparisonChart
CryptoStockApplication
+ main(String args[]): void
StockController
- stockStatService:StockStatService
+ getStocks(@RequestParam(required = true) String symbols[], @RequestParam(defaultValue = "5") int limit):Stocks
+ getStockBySymbol(@PathVariable String symbol): Stock
+ getStockNames(@RequestParam(required = false) String symbols[], @RequestParam(defaultValue = "5") int limit): List<String>
+ getHistoricalPrices(@PathVariable String symbol): List<List<Object>>
CoinController
- coinStatService:CoinStatService
- ALLOWED_PERIODS:List<String>
+ getCoins(@RequestParam(defaultValue = "5") int limit, @RequestParam(required = false) String name, @RequestParam(defaultValue = "price") String sortBy, @RequestParam(defaultValue = "asc") String sortDir):Coins
+ getCoinById(@PathVariable String id): Coin
+ getCoinChart(@PathVariable String id, @RequestParam(defaultValue = "1w"): List<List<Object>>List<List<Object>>
UserController
- userService:UserService
+ login(@RequestBody LoginRequest loginRequest): String
+ register(@RequestBody User user): Register
+ getPreferences(@RequestHeader('Authorization') String token): UserPreferences
+ setPreferences(@RequestBody UserPreferences preferences, @RequestHeader("Authorization") String token): UserPreferences
+ getUser(@RequestHeader("Authorization") String token): User
+ updateUser(@RequestHeader("Authorization") String token): User
StatisticsController
- coinStatService:CoinStatService
- stockStatService: StockStatService
+ getComparisonChart(): List<Map<String, Object>>
Supportive model classes for JSON decomposition
Coin
+ id
+ icon
+ name
...
Coins
+ result
+ meta
GlobalQuoteResponse
+ globalQuote
+ information
LoginRequest
+ username
+ password
Stock
+ symbol
+ price
+ volume
...
StockOverviewResponse
+ symbol
+ fullName
+ marketCap
...
Stocks
+ stockList
+ meta
User
+ username
+ password
+ email
+ language
+ currency
UserPreferences
+ favoriteStocks
+ favoriteCoins
StockStatService
- STOCK_API_URL:String
- API_KEY: String
- GLOBAL_QUOTE: String
- OVERVIEW: String
- TIME_SERIES_DAILY: String
- createStock(GlobalQuoteResponse.GlobalQuote globalQuote, StockOverviewResponse overviewResponse): Stock
- getResponseBody(String API_endpoint, String symbol): String
+ getStocks(String[] symbols, int limit): Stocks
+ getStockBySymbol(String symbol): Stock
+ getStockNames(String[] symbols, int limit): List<List<Object>>
+ getHistoricalPrices(String symbol): List<String>
CoinStatService
- COIN_API_URL:String
- API_KEY: String
- CURRENCY: String
+ getCoins(int limit, String name, String sortBy, String sortDir): Coins
+ getCoinById(String coinId): Coin
+ getCoinChart(String coinId, String period): List<List<Object>>
UserService
- DBConnection:<DB>
- users:Map<String, User>
- preferences: Map<String, UserPreferences>
- tokens: Map<String, String>
- getDefaultPreferences(): UserPreferences
+ login(String username, String password): String
+ register(User user): User
+ getPreferences(String userId): UserPreferences
+ setPreferences(String userId, UserPreferences prefs): UserPreferences
+ getUser(String userId): User
+ updateUser(String userId, User user): User
+ isValidToken(String token): boolean
AppConfig
- corsFilter(): CorsFilter
AlphavantageAPI
-GLOBAL_QUOTE: /query?function=GLOBAL_QUOTE&symbol={symbol}&apikey={API_KEY}
- OVERVIEW: /query?function=OVERVIEW&symbol={symbol}&apikey={API_KEY}
TIME_SERIES_DAILY: /query?function=TIME_SERIES_DAILY&symbol={symbol}&apikey={API_KEY}
CoinStatAPI
- /coins?limit={limit}&sortBy={sortBy}&sortDir={sortDir}&currency={CURRENCY}&name={name}
- /coins/{coinId}?currency={CURRENCY}
- /coins/{coinId}/charts?period={period}
postgresql
User
Preferences
AlphaVantage: https://www.alphavantage.co/documentation/
Coin stats: https://docs.api.coinstats.app/reference/get-coins

Patterns used in the backend

The following patterns and design solutions are used in the backend part of the application

Model-View-Controller (MVC) Pattern:

MVC architecture separates the application's concerns, keeping the logic (model), data handling, and user interactions (controller) separate from the presentation layer (view).

The pattern is implemented by creating 3 logical layers: Controllers, Models (Services), and View. Controllers are responsible for managing specific types of data and API endpoints, like handling user data, cryptocurrency, stocks, and comparison chart data. Models interact with external APIs (such as AlphaVantage and CoinAPI) and processes data, which it then passes to the controllers for API response handling. Views are managed by the React frontend, which consumes the backend APIs and displays data to the user. This separation makes the system more scalable and easier to modify.

The model was selected due to its simple realization, separation of concerns, and scalability. For example, in the current state of the app, there is no database for the user data, but the UserController can utilize UserService without considering how the data is handled inside and the service can mock the database behavior. Later a real database will be added but that will not require changing controllers.

API Versioning (/v1/ pattern):

API versioning helps manage changes over time, allowing the application to upgrade without disrupting existing clients.

In this backend, versioned URL structures (e.g., /v1/stocks, /v1/coins) are used for each API endpoint. This approach supports future growth, allowing new features or changes to be added in a new version, while existing endpoints remain stable for older clients.

API versioning was selected to ensure backward compatibility, making it easier to introduce new functionality without breaking existing integrations.

Singleton Pattern for Configurations:

The Singleton pattern is applied for configurations that need a single global instance, ensuring consistent access across the application. This pattern was selected to provide a consistent control over the server from one place

Composition and Dependency Injection in Controller-Model Relationship:

Composition and Dependency Injection (DI) allows controllers to utilize services in a coupled way, making the codebase more modular and testable.

In this backend, UserController and CoinController controllers depend on services such as UserService and CoinStatService, that are injected via the dependency injection. This approach helps to easily replace services during testing (e.g., using mock services instead of real API calls), and flexibility to switch implementations if needed.

Composition and DI were chosen to provide testability and flexibility. For example, unit tests can use mock services in place of actual services, isolating the controller logic without requiring real data sources.

Testing plan

Extensive testing probably wouldn't(and shouldn't) be done. We can write simple unit tests with JUnit for the java backend and we could use selenium to automate the frontend testing. We can also do manual testing of the front end. In either case, all testing will be documented.

Technologies Used

Frontend Testing

- **Selenium:** for automated testing of the frontend. We will also do a fair bit of manual testing.

Backend Testing

- **JUnit:** for unit testing the backend.

Project work plan of action

Scrum

We decided to use scrum because we never know what will change during the project work, maybe a TA will require something to be changed, maybe we'll decide to change an API, etc. It'd bring more order to our work and everyone will carry responsibility for the tasks assigned to be carried out in the agreed time frame.

Self-evaluation

The original design comprehensively covered the main points of the application to be implemented. It allowed us to make an abstraction of the whole idea of the program which separated the field of development into clear areas of corresponding patterns of usage and labor division. This made the development easy, high-quality, and predictable.

Adherence to Original Design

We've been able to stick closely to the original design. A lot of parts of the application were implemented:

- Stocks and cryptocurrencies lists and individual pages
- Stock and cryptocurrency charts
- Basic comparison view
- User login, registration

We are planning to continue using the plan for the implementation of the rest of the features.

Changes for Remaining Features

Despite the overall success of the initial design, the core feature of the application, "compare coin and stock," was only partially implemented. As we start developing it we expect to do some changes to the planned interfaces, particularly when combining data types (like stock and coin data) into a unified comparison chart. This also includes updating to a paid version of stock API or finding a better alternative to it due to its low daily request limit (only 25 per day) which is not enough for our users

Documented Changes to the Original Design

While the general structure has remained consistent with the initial design, we made a minor adjustment to the comparison functionality of the app. For the MVP purposes we implemented MockDataService to simulate responses for the StatisticsController, allowing us to test and demonstrate functionalities without full data integration. This mock service will be replaced with data in the final version of the app.

Conclusion

We are confident that the design we made will help us in the implementation of the remaining features. The overall structure of the MVC model and modular approach that we were following has provided a solid foundation making it easy to build plans and add new things to the existing codebase.