CS 4287 Group 7: Investment Portfolio Tracker App

Ronni Tong
Computer Science
Vanderbilt University
Nashville, USA
ronni.tong@vanderbilt.edu

Xueyuan Li
Data Science
Vanderbilt University
Nashville, USA
xueyuan.li@vanderbilt.edu

Xishan Deng
Data Science
Vanderbilt University
Nashville, USA
xishan.deng@vanderbilt.edu

***Abstract*—We developed a cloud-based platform that simplifies investment management by allowing users to input various investment products and custom API sources. We've also leveraged several cloud technologies, such as Vagrant and Ansible for hassle-free deployment, Kafka for efficient data streaming, MongoDB for data storage, Docker for creating containerized services, and finally AWS and Chameleon for hosting different parts of the service.**

***Index Terms*—cloud computing, CRUD, react project, investment portfolio, Node JS**

I. Problem Statement

Today's investors encounter a notable challenge: the absence of a centralized platform for effectively overseeing and evaluating their array of investment holdings. Despite the multitude of investment opportunities available, there exists a notable gap in the market for a singular hub offering real-time tracking and analysis of diverse investment products. Without this centralized solution, investors are left navigating fragmented data sources, impeding their ability to gain holistic insights and make informed decisions regarding their investments.

II. Solution

A. Overview

To solve this problem, we're developing an investment portfolio tracker app. Investors can easily input details about their investments, including type, quantity, and value. They can also use customized API sources to ensure the app displays the desired output.

The project has two parts: the frontend and the backend. We used the React framework and the Material UI specifically for frontend development. For the backend part, we used Node JS and Express JS to create several API endpoints to support CRUD operations. For consistent and reliable data storage, we utilized MongoDB for document-style databases.

For data storage, MongoDB serves as the backbone, storing application data and supporting essential CRUD operations for smooth user interactions.

We also employed several cloud technologies during the development process, such as using Kafka to set up data streaming pipelines and Docker for creating the MongoDB container.

Finally, we deployed the finished app on AWS EC2 Virtual Machines using automation tools like Ansible and Vagrant.

B. Backend

After careful evaluation of different backend frameworks, we decided on Node JS to match the frontend language choice. Given that the app aims to support CRUD operations of investment products, we created several endpoints for investment products, including stocks, futures, and CDs (certificates of deposits). Each endpoint supports the creation, read, deletion, and update of the investment product subscriptions.

Using ExpressJS, we created the APIs using the URL path of */investments/product*

Below are the API endpoint designs for this investment tracker app:

1) **Stock APIs:**

a) *GET /stocks/* - Get all stock data

b) *POST /stocks/* - Create new stock data entry with given payload data

c) *GET /stocks/:symbol/* - Get all stock data for the given stock symbol

d) *DELETE /stocks/:id/* - Delete the given ID's stock data entry

e) *PUT /stocks/:id/* - Updathe the given ID's stock data entry with the payload data

f) *GET /stockvalue/* - Get the live value data for all stocks

g) *GET /stockvalue/:symbol/* - Get the live value data for the given stock using rhe symbol as the query

2) **Future APIs:**

a) *GET /futures/* - Get all future data

b) *POST /futures/* - Create new future data entry with given payload data

c) *GET /futures/:symbol/* - Get all future data for the given future symbol

d) *DELETE /futures/:id/* - Delete the given ID's future data entry

e) *PUT /futures/:id/* - Update the given ID's future data entry with the payload data

f) *GET /futurevalue/* - Get the live value data for all futures

g) *GET /futurevalue/:symbol/* - Get the live value data for the given future using the symbol as the query

3) **CD APIs:**

a) *GET /cds/* - Get all cd data

b) *POST /cds/* - Create a new cd data entry with given payload data

c) *GET /cds/:name/* - Get all cd data for the given cd name

d) *DELETE /cds/:id/* - Delete the given ID's cd data entry

e) *PUT /cds/:id/* - Update the given name's cd data entry with the payload data

f) *GET /cdvalue/* - Get the live value data for all cds

g) *GET /cdvalue/:name/* - Get the live value data for the given cd using the name as the query

C. Frontend

As we didn't have much experience with frontend engineering, we chose the easy and popular React framework. Also, to unify the page design style, we used the Material UI library to create the dashboard components.

We based the design of the frontend UI on popular stock apps such as Fidelity and Robinhood. The dashboard page includes two pie charts, one for the portfolio breakdown and one for each investment product. Users can select the investment product they want to view using a dropdown menu.

Additionally, they can view all the investment product entries in the dropdown table below the pie charts. Each field shows its value in an editable text box with the save button below it. The users can modify the name, symbol, and other fields of the investment products as they wish and click the save button to save the changes. They can also delete entries by clicking the delete button at the end of each table row.

Finally, the app has three forms for adding new stock, future, or CD data entries. The users can find these forms by choosing from the dropdown menu from the add icon on the top right corner of the page. They would then put in custom values for each field, such as the name, symbol, purchase time, amount, etc. After clicking the submit button at the end of the form, they can see the latest values of the newly added products on the dashboard page.

D. DevOps

For the DevOps part of the project, we used Chameleon, EC2, Docker, Ansible, Vagrant and Kafka.

Firstly, we created three new EC2 instances and reused an old Chameleon instance for hosting the app.

Then, we wrote several Ansible playbooks for initializing the virtual machine environments, such as installing Java, Kafka, and Docker on all machines. We ran the playbooks using the Vagrant provision command.

Next, we used the Docker container service for composing and hosting the MongoDB database that stores all the backend data for this app.

For the last step, we set up the Kafka data pipeline by creating Python producer and consumer that stream and dump the latest stock and future price data in the database.

E. Database

Since the app requires a lot of data streaming and changes on the database, we chose MongoDB, the NoSQL document-based database service. We connected the backend part of the project to our MongoDB instance on EC2 using the Node JS driver.

We created two databases for the app, one for investment product subscriptions and one for the latest price data. The subscription table has three collections for the three investment products and stores the user inputs about what, how, and when they purchased each entry. The live data table receives the Yahoo Finance stock and future price data from the data pipeline. The only exception is the CD live prices, as we used the CD value formula for calculating the actual market values.

III. Cloud Technologies

Our project incorporated the following cloud technologies:

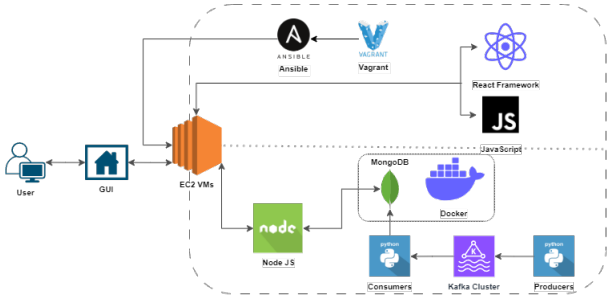

Fig. 1. This figure shows the overall framework of our project.

1) **Docker:** Deploying the messaging, processing, and analytics components.

2) **MongoDB:** Storing investment product subscriptions and live data in different databases and collections.

3) **Docker:** Creating and hosting containers for project components such as MongoDB servers.

4) **AWS & Chameleon Cloud:** Hosting the docker container and deploying the frontend and backend of the application for external access from different devices and browsers.

5) **Ansible:** Automating the installation of Docker and Kafka on the remote virtual machines.

6) **Vagrant:** Creating the Vagrant Virtual Machine and executing the Ansible playbooks for DevOps automation.

7) **Kafka:** Creating the data pipeline for streaming the live prices of stock and futures to the MongoDB Docker container.

IV. Architecture

Our work Fig. 1 shows the framework of our investment portfolio tracker app.

V. UI Screenshots

The Figure Fig. 2 displays a user interface for an Investment Portfolio App, which is divided into various sections for a comprehensive overview of the user's investment distribution and details. The top section features a pie chart labeled "Investment Portfolio," illustrating the percentage distribution among three investment types: stocks (62.8 percent), futures (20.1 percent), and certificates of deposit (CDs) (17.1 percent). Adjacent to this chart, there are two infoboxes: the first displays the total investment worth of 204,957.29 dollars with a timestamp of the value's last update, and the second box shows the total worth of stocks alone amounting to 41,1663 dollars with its respective timestamp.

Below the overview charts, detailed tables 3 are provided for each investment type. The stock table lists individual stocks like Amazon, Meta, Nvidia, and Apple with corresponding symbols, the amount owned, unit value, total value, and purchase time, giving a clear transaction history and current valuation for each stock. Similarly, the future investments table lists items such as Micro Gold Futures and E-Mini 500 Jun 24 with relevant details, and the CD investments table lists entries from institutions like Capital One and American Express, showing deposit amounts, terms in months, interest rates, current values, and purchase dates, thereby offering an in-depth financial snapshot for each CD held.

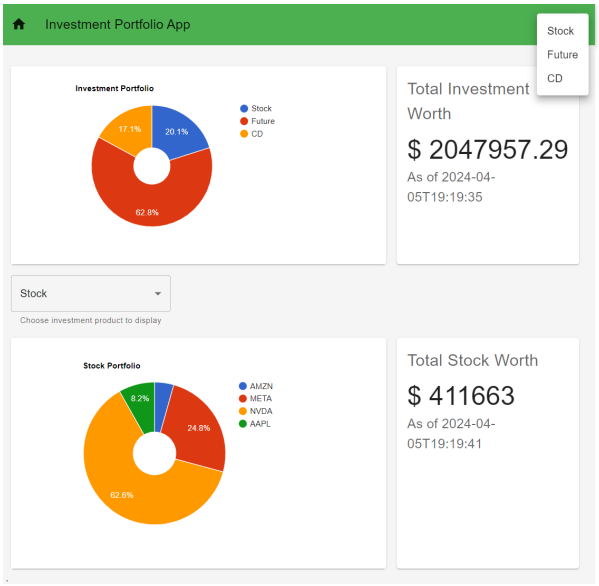

Fig. 2. This figure shows overview charts.

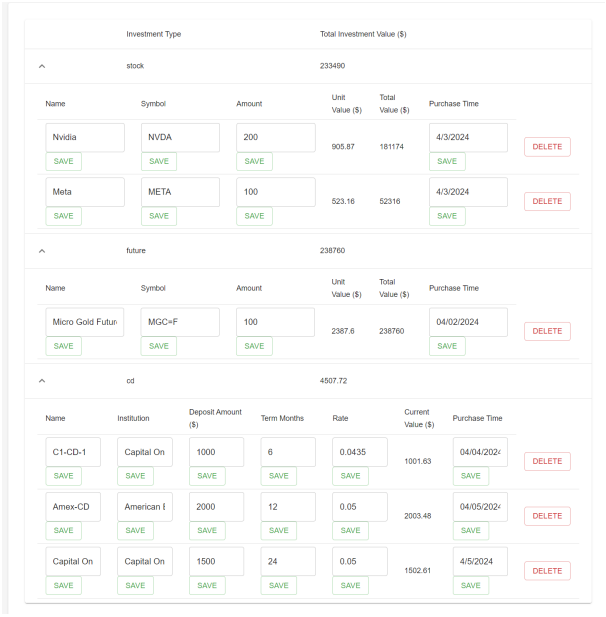

Fig. 3. This figure shows detailed tables.

VI. Future Work

A. Account Management

Currently, the app only supports one instance of the centralized tracker app. The desired future state, however, should allow each user to create their user profiles and have their own version of the investment tracker app so that they can track their own investment portfolio.

B. Support for More Investment Products

Given the time limit, we could support stocks, futures, and CDs in this app. The ideal situation would be that the app allows users to track almost all investment types, such as bonds, options, cryptocurrency, etc.

C. Distributed System

As for now, the app is deployed on one remote virtual machine, which saves a lot of time and effort. But to ensure better scalability and availability, we should aim to distribute the app in more regions on scalable web service platforms. In this way, the app would be more fault-tolerant and faster for all levels of workloads from all geographical areas.

VII. Acknowledgements

This work is supported by Vanderbilt University and the School of Engineering. The project is supervised by Professor Singh and serves as the final deliverable for CS 4287: Principles of Cloud Computing in the Spring 2024 semester.

References

[1] ChatGPT, response to author query. OpenAI [Online]. https://chat.openai.com/ (accessed April 16th, 2024).