Katholieke
Universiteit
Leuven

Faculty of Engineering Technology

DISTRIBUTED APPLICATIONS: FINAL PROJECT

service-oriented distributed applications on a cloud platform

Vincent Vansant (r0800542)

Academic year 2023–2024

Contents

1 Level 1: Mandatory Basic Requirements

1.1 Application Overview

Our solution is a distributed system designed to facilitate stock trading via a broker application that interacts with two external stock exchanges: the New York Stock Exchange (NYSE) and NASDAQ. The architecture involves several key components: a broker frontend, a broker backend, and two stock exchange services. The broker backend is connected to a Firestore database for persistent storage and a Firebase service for authentication and security.

The broker frontend is a web application that allows users to interact with the broker services. It provides a user interface for logging in, viewing stock and index fund details, managing orders, and accessing manager specific functionalities.

The broker backend is responsible for handling user authentication, interfacing with the NYSE and NASDAQ services, managing orders, and storing data in Firestore. It includes various API endpoints to facilitate these operations and is implemented using Spring Boot. WebClient is used for RESTful communication, while Firestore and Firebase manage data storage and authentication, respectively.

The NYSE service simulates the New York Stock Exchange. It provides APIs for listing stocks, retrieving daily closing prices, and handling buy/sell orders with either market or limit orders. This service updates stock prices every 30 seconds and supports callback URLs for order updates. It is built using Spring Boot and ScheduledExecutorService for periodic updates. Similarly, the NASDAQ service simulates the NASDAQ stock exchange with equivalent functionalities and technologies.

1.1.1 Broker Backend API Endpoints

- `/api/hello`: Initializes user data in Firestore.
- `/api/whoami`: Returns the authenticated user's information.
- `/api/getAllCustomers`: Retrieves all user data.
- `/api/getAllOrders`: Retrieves all orders across all users.
- `/NYSEstocks`: Lists all stocks available on NYSE.
- `/NASDAQstocks`: Lists all stocks available on NASDAQ.
- `/indexFunds`: Lists all index funds.
- `/NYSEstocks/dailyclosingprice/{symbol}`: Retrieves daily closing prices for a NYSE stock.
- `/NASDAQstocks/dailyclosingprice/{symbol}`: Retrieves daily closing prices for a NASDAQ stock.
- `/indexFunds/{symbol}`: Retrieves details for a specific index fund.
- `/api/NYSE/buy/{symbol}`: Places a buy order for a NYSE stock.
- `/api/NASDAQ/buy/{symbol}`: Places a buy order for a NASDAQ stock.
- `/api/indexFund/buy/{symbol}`: Places a buy order for an index fund.
- `/api/NYSE/sell/{symbol}`: Places a sell order for a NYSE stock.

- `/api/NASDAQ/sell/{symbol}`: Places a sell order for a NASDAQ stock.
- `/api/indexFund/sell/{symbol}`: Places a sell order for an index fund.
- `/api/orders`: Retrieves all orders for the authenticated user.
- `/order-callback`: Endpoint for receiving order status updates from the stock exchanges.

1.1.2 NYSE and NASDAQ Service Endpoints

- `/stockexchange/dailyclosingprice/{symbol}`: Retrieves daily closing prices for a stock.
- `/stockexchange/stocks`: Lists all stocks.
- `/stockexchange/buy/{symbol}`: Places a buy order for a stock.
- `/stockexchange/sell/{symbol}`: Places a sell order for a stock.

1.2 Access Control for Manager Methods

Access control is enforced using Firebase authentication and Spring Security. Firebase generates an ID token upon user login, which includes custom claims to indicate user roles. These roles are used in Spring Security to control access to specific endpoints.

In the `WebSecurityConfig` class, access control for manager-specific methods is enforced by adding `.antMatchers("/api/getAllOrders", "/api/getAllCustomers").hasRole("MANAGER")`.

A security filter intercepts all (starting with /api) incoming requests to verify the ID token. The filter extracts the token from the `Authorization` header, decodes it to retrieve the user's email and role claims, and assigns these to a `User` object. If the user has the "manager" role, the request proceeds; otherwise, access is denied. The decoded information is then stored in the `SecurityContext` for subsequent access control checks.

1.3 Potential Issues with ACID Properties

In our implementation, atomicity is generally maintained by only storing ETF orders if all buy/sell orders of the individual stocks are successful. However, ensuring true atomicity and consistency across multiple exchanges is challenging, a commit and rollback system should be implemented to address this, ensuring that any failures on the exchanges themselves can be rolled back, preventing partial transactions.

Isolation is managed correctly in our system. Transactions are executed independently, ensuring that operations within a transaction are not visible to other transactions until the transaction is completed. This prevents concurrent transactions from interfering with each other, thereby maintaining data integrity.

Consistency and durability are generally well-maintained through the use of Firestore for order storage and proper error handling.

1.4 Coping with External Supplier Service Failures

Our application is designed to handle failures of the external supplier services with resilience. If one of the stock exchanges (either NYSE or NASDAQ) experiences a failure or becomes unavailable, the application ensures that only the stocks and index funds associated with that particular exchange are impacted. The backend prevents any buy or sell operations for stocks and ETFs from the affected exchange, ensuring users do not encounter unexpected failures during transactions. No orders are stored in Firestore if a transaction involving an affected stock or index fund fails, maintaining the integrity of the order data. The other exchange continues to function normally, allowing users to view, buy, and sell stocks and index funds that are not affected by the outage.

While our application generally handles failures well, any pending orders on the affected exchange will no longer be able to be filled, even after the exchange is rebooted, since we store everything, including callback URLs, in memory.

2 Level 2: Optional Advanced Requirements

2.1 Data Model Structure

Our data model primarily focuses on storing ETFs and Orders in Firestore. Each ETF is composed of multiple component stocks from the NYSE and NASDAQ exchanges. We store detailed information about each ETF, including its symbol, name, component stocks, their weights, and associated exchanges. Orders, whether they are for individual stocks or ETFs, are stored with details such as the order ID, symbol, price, state, type, user email and date time.

Because of real-time stock updates, we do not store stock prices or ETF prices on the broker side. Instead, we fetch live data from the external exchanges. This approach ensures that users always receive the most current stock and ETF prices, reflecting the latest market conditions. While this means that stock and ETF prices are not persisted in our data model, it allows for real-time trading functionality, which is essential for the operation of a stock broker platform. By balancing the storage of static ETF compositions and user orders with live price data, our data model supports both efficient transaction management and real-time trading capabilities.

2.2 Google Cloud Deployment

We followed all the steps to deploy our broker application on Google Cloud. Despite our efforts, we encountered a MojoExecutionException that we could not resolve in time. This prevented us from successfully launching the broker in the cloud environment.

3 Team Member Contributions

- Vincent Vansant: everything (32 hours)
- Lotte Kesteleyn: didn't enroll in the course