Klear Trade Settlement System

This System facilitates stock market trading by allowing clients to register, create trade orders, and execute trades. The application is integrated with a Kafka messaging system for real-time processing and includes validation and monitoring mechanisms.

Note:

While developing this system, all the components have been consolidated into a single service. This decision was made to improve ease of use and make the review process easier.

Since this application utilizes Kafka for real-time messaging and schedulers for automated processes, separating these functionalities into multiple services would have introduced unnecessary complexity for reviewers to review and this this in their local machine.

I have mimicked the behaviour of a multi-service architecture within a single system, this approach allows reviewers to easily navigate and assess the application without the added overhead of managing multiple services.

Technologies Used: **Java, Spring webflux, RESTful APIs, Kafka Messaging, Spring Scheduler, r2dbc, Mysql Database, Spring Security-JWT** .

I have opted reactive programming with R2DBC and Spring WebFlux to enable asynchronous request processing, resulting in faster response times and enhanced performance. This approach allows the application to handle multiple requests concurrently without blocking, improving resource utilization.

API Endpoints

1. Client Registration

Endpoint:

POST http://localhost:8889/api/v1/clients

Request Body:

```
{
  "clientId": "241013327180513",
  "name": "Ria Das",
  "email": "riadas1911@gmail.com",
  "availableBalance": 10000
}
```

Responses:

201 Created: Successful registration.

400 Bad Request: If the request is malformed.

Error Message: "Column cannot be null" if any required property is missing.

Table:

```
CREATE TABLE clients (
   client_id VARCHAR(255) PRIMARY KEY,
   name VARCHAR(100) NOT NULL,
   email VARCHAR(100) NOT NULL,
   available_balance DECIMAL(10, 2) NOT NULL
);
```

2. Trade Order Creation

Endpoint:

POST http://localhost:8889/api/v1/orders

Request Body:

```
{
  "clientId": "241026296368892",
  "stockSymbol": "UPL",
  "quantity": 100,
  "price": 30,
  "sellerId": "241026296368892"
}
```

In the **TradeOrderController**, several critical validation checks are performed during the trade order creation process to ensure the integrity of the data and the execution of valid trades.

Validation Checks:

1. **Valid Client ID:**
    - The system verifies that the provided client ID exists in the database. This ensures that only registered clients can place trade orders. If the client ID is invalid or does not correspond to any existing client, an exception is thrown indicating that the client is not found.
2. **Sufficient Available Balance:**
    - The application checks whether the client has enough available balance to cover the total cost of the trade. This is calculated by multiplying the order quantity by the price per share. If the available balance is insufficient, an exception is thrown with a message indicating that the client does not have enough funds to complete the order.
3. **Non-Negative Price:**
    - The price per share provided in the trade order must be a non-negative value. The system checks that the price is greater than or equal to zero. If a negative price is submitted, an exception is thrown, indicating that the price cannot be negative.
4. **Positive Quantity:**
    - The quantity of shares requested in the trade order must be a positive integer. The system ensures that the quantity is greater than zero. If the

quantity is zero or negative, an exception is thrown, indicating that the quantity must be a positive value.

Exception Handling: If any of these validation checks fail, the system throws an appropriate exception describing the error.

Response:

```
{
  "id": "2610243888114",
  "clientId": "241026296368892",
  "stockSymbol": "UPL",
  "quantity": 1,
  "price": 1,
  "status": "PENDING",
  "exchangeId": null,
  "deviatedBalance": null,
  "sellerId": "241026296368892"
}
```

Table:

```
CREATE TABLE trade_orders (
  id VARCHAR(255) PRIMARY KEY,
  client_id VARCHAR(100) NOT NULL,
  stock_symbol VARCHAR(10) NOT NULL,
  quantity INT NOT NULL CHECK (quantity > 0),
  price DECIMAL(10, 2) NOT NULL CHECK (price > 0),
  status ENUM('PENDING', 'EXECUTED', 'CLEARED', 'SETTLED') NOT NULL,
  exchange_id VARCHAR(255),
  deviated_balance DECIMAL(10, 2) DEFAULT 0.00,
  seller_id VARCHAR(255),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  FOREIGN KEY (seller_id) REFERENCES client_shares(client_id)
);

CREATE TABLE client_shares (
  client_id VARCHAR(100) NOT NULL,
  stock_symbol VARCHAR(10) NOT NULL,
  quantity INT NOT NULL CHECK (quantity >= 0),
        credited_amount INT DEFAULT 0,
  PRIMARY KEY (client_id, stock_symbol),
  FOREIGN KEY (client_id) REFERENCES clients(client_id)
    ON DELETE CASCADE
);
```

3. Get Order Status

Endpoint:

GET http://localhost:8889/api/v1/order/status/{id}

Path Variable:

id: Trade ID (e.g., 2610244660774)

Responses:

200 OK: Returns order status.

404 Not Found: If the order does not exist.

4. Trade Execution

In this application, a Kafka listener is integrated to facilitate real-time processing of trade orders. After a trade order is successfully created and saved to the database, a message is published to the Kafka topic designated for trade events.

Configuration Details:

- **Topic:** trade-events
- **Consumer Group ID:** trade-order-consumer-group

Functionality: The **TradeExecuteController** is designed to listen for messages on the specified Kafka topic. Upon receiving a message, the following actions are triggered:

1. **Random Exchange Selection:** The listener randomly selects an exchange from a pre-defined list, simulating real-world trading scenarios. These lists of exchanges are stored in a table.
   Table: exchanges
2. **Dynamic Price Adjustment:** The system applies a dynamic pricing logic, allowing for price fluctuations, dynamic price logic has been implemented by ±10% fluctuation. This mimics the volatility of stock prices.
3. **Deviated Price Calculation:** The deviated price, any fluctuations during the execution phase, is calculated and stored in the database.
4. **Amount and Order Status Updates:** The application deducts the trade value amount from the buyer's account.
   Finally, the status of the trade order is updated to "EXECUTED," along with any relevant details such as the selected exchange ID and the calculated deviated balance. This ensures that the trading history is accurately recorded and retrievable.

Table:

```
CREATE TABLE exchanges (
    exchange_id VARCHAR(255) PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    liquidity DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP --
Last updated timestamp
);
```

Sample Inserted Records:

```
INSERT INTO exchanges (exchange_id, name, liquidity) VALUES
('NYSE', 'New York Stock Exchange', 300000.00),
('NASDAQ', 'NASDAQ Stock Market', 250000.00),
('LSE', 'London Stock Exchange', 100000.00),
('JPX', 'Japan Exchange Group', 80000.00),
('HKEX', 'Hong Kong Stock Exchange', 90000.00),
('TSX', 'Toronto Stock Exchange', 60000.00),
('SSE', 'Shanghai Stock Exchange', 70000.00),
('BSE', 'Bombay Stock Exchange', 50000.00);
```

5. Monitoring for Clearing

A dedicated scheduler, named **ClearingScheduler**, has been integrated into the application to manage the clearing process of executed trade orders. This scheduler runs at regular intervals of every 5 minutes.

Functionality:

1. **Scheduled Execution:**
   - The **ClearingScheduler** is configured to trigger every 5 minutes. This periodic execution allows for continuous monitoring of the trade order table, scheduler queries the trade order database to retrieve all records with an EXECUTED status.

2. **Deducting Share Quantities:**
   - For each retrieved trade order, the scheduler deducts the specified share quantities from the seller's account. This process involves updating the seller's remaining shares based on the quantity of shares sold in the executed orders. This ensures that the seller's inventory reflects the most current state post-trade.
3. **Changing Order Status to CLEARED:**
   - Once the share quantities have been adjusted, the scheduler updates the status of each processed trade order from EXECUTED to CLEARED.

6. Monitoring for Settlement

There is one more scheduler integrated at the settlement phase. This schedular is automating the payment settlement process for trade orders that have reached the CLEARED status.

Functionality:

1. **Scheduled Execution:**
   - The **SettlementScheduler** is configured to run every 5 minutes. This frequent execution allows the system to continuously monitor trade orders and efficiently manage the payment settlements.

    - Scheduler queries the trade order table to retrieve all trades with a CLEARED status. These orders indicate that the necessary share transfers have been completed and are now ready for payment processing.

2. **Processing Payments:**
    - For each trade order identified as CLEARED, the scheduler calculates the total payment amount due to the seller, and finally the amount is credited to the respective seller's account.

3. **Updating Order Status to SETTLED:**
    - After successfully processing the payment, the scheduler updates the status of each processed trade order from CLEARED to SETTLED. This status change signifies that the financial transaction has been completed and the order is fully processed.

7. Security | JWT

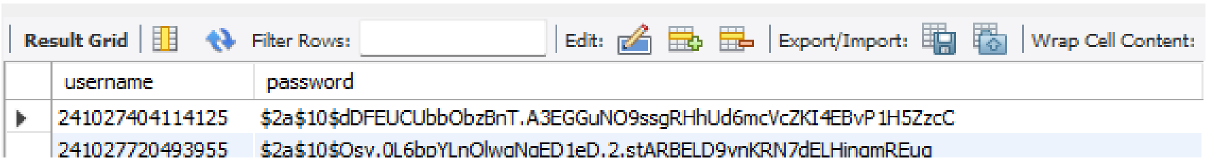
Technologies Used

- **JWT** (JSON Web Token) for authorization.
- **Spring Security** for securing endpoints.
- **BCrypt** for password encryption.

User Storage Table

```
CREATE TABLE user_secrets (
   username VARCHAR(255) PRIMARY KEY,  -- Username as the primary key, must be unique
   password VARCHAR(255) NOT NULL      -- Hashed password
);
```

Users are stored in client_secret. Each user has a username and an encrypted password. The passwords are stored securely using the **BCryptPasswordEncoder**.



username	password
241027404114125	$2a$10$dDFEUCUbbObzBnT.A3EGGuNO9ssgRHhUd6mcVcZKI4EBvP1H5ZzcC
241027720493955	$2a$10$Osv.0L6bpYLnOlwqNqED1eD.2.stARBELD9vnKRN7dELHinqmREuq

Endpoint:

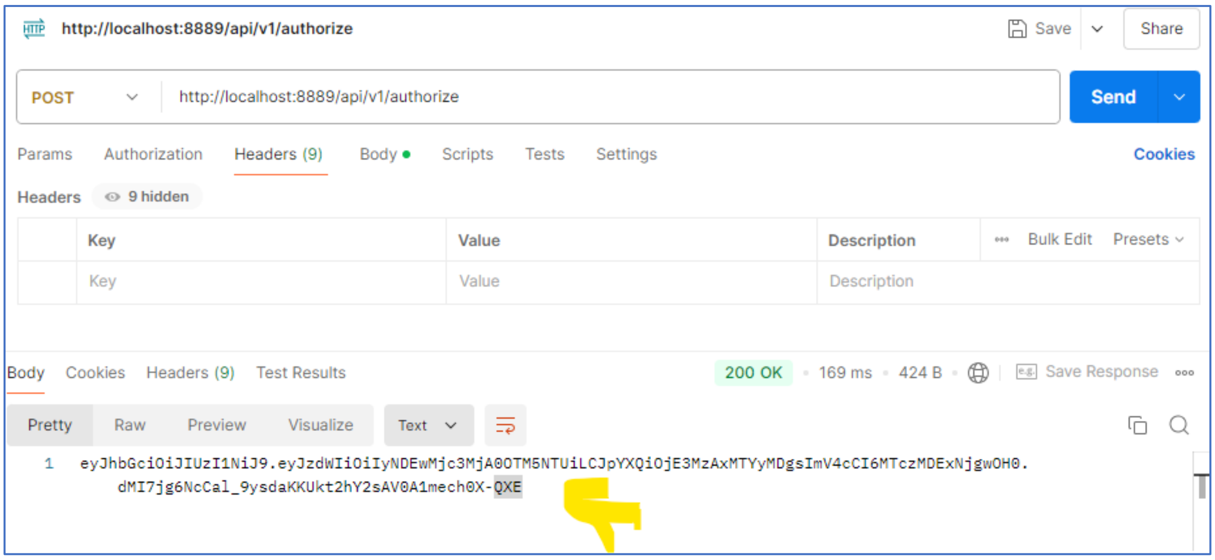
POST http://localhost:8889/api/v1/authorize

Request Body:

```
{
  "username": "xyz",
  "password": "xyz"
}
```

Response :

Status Code: 200 OK



http://localhost:8889/api/v1/authorize

POST http://localhost:8889/api/v1/authorize

Headers 9 hidden

Key	Value	Description
Key	Value	Description

200 OK · 169 ms · 424 B

```
eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiIyNDEwMjc3MjA0OTM5NTUiLCJpYXQiOjE3MzAxMTYyMDgsImV4cCI6MTczMDExNjgwOH0.
    dMI7jg6NcCal_9ysdaKKUkt2hY2sAV0A1mech0X-QXE
```

Status Code: 401 Unauthorized

{ "error": "Invalid username or password" }

Explanation:

1-The client sends a POST request to the /authorize endpoint with the username and password in the body.
2-The server checks if the user exists in the client_secret table.
3-If the user exists, the server verifies the provided password against the stored hashed password using BCryptPasswordEncoder.
4-If the password is valid, the server generates a JWT, signs it, and sends it back to the client.
5-The client can then use this token for subsequent requests to access protected resources.

This is the end of the document. Thank you!