Here are some steps you can follow to architect the backend of a trading app:

1. **Determine the requirements:**

Identify the key features and functionality that the trading app will need to support. This may include real-time market data, trade execution, account management, and compliance with laws and regulations.

You can find above, the main Features that our application should have, as well as a technical Overview on how things should work.

* 1. **Real-time market data**

1. **General Overview**

A trading app typically includes real-time market data, such as quotes, charts, and news, to help users make informed trading decisions.

There are several ways to retrieve real-time market data for a trading app:

* **Use an API:** Many exchanges and financial institutions offer APIs that allow developers to access real-time market data. You can use these APIs to retrieve data such as quotes, charts, news, and other market information.
* **Use a data feed:** Some vendors offer data feeds that provide real-time market data. These data feeds can be delivered through a variety of means, such as web sockets, message queues, or HTTP requests.
* **Use web scraping:** Some websites provide real-time market data on their websites, which can be accessed by scraping the data from the website. This approach can be less reliable and may be subject to change if the website changes its structure or blocks the scraping.
* **Use a market data provider**: There are various market data providers that offer real-time market data. These providers may offer data through APIs, data feeds, or other means, and may cover a wide range of markets and instruments. Ex:Redline

Regardless of the approach you choose, it is important to ensure that the market data is reliable and accurate, and that you have the necessary permissions to access and use the data.

1. **Technical Choice**

*I’ll go with Alpha Vantage: Alpha Vantage is a financial data provider that offers a wide range of APIs for accessing real-time market data. The APIs cover a variety of markets and instruments, including stocks, options, futures, currencies, and more. Alpha Vantage offers both free and paid plans, with the free plan providing access to a limited number of API calls per day.*

***C. Simplified Technical Diagram***

Diagram

Description automatically generated

In this diagram, the Stock Price API is the interface that clients (e.g., mobile apps, websites, etc.) use to request real-time stock prices. The API communicates with a Stock Market Data Provider (e.g., Alpha Vantage in Our case) to retrieve the latest stock prices and then returns the data to the client. The Stock Market Data Provider fetches the stock prices from the stock exchange(s) and stores them in a database. We will also have a kafka queue where we would be having a Pub/Sub model that the user subscribes to for getting the stock. On searching for a specific stock, it will check the cache if its not there in the cache tyhen it will query the database. The API can then query the database to retrieve the requested stock price data and return it to the client in the appropriate format (e.g., JSON).

Diagram

Description automatically generated

In this diagram, the Stock Price API retrieves real-time stock prices from the Stock Market Data Provider and returns them to the client (e.g., a mobile app or website). The client then displays the stock prices to the user on their device (e.g., a smartphone or computer).

There are many ways that the stock prices can be displayed to the user, but this is more a Front-end work, depending on the specific requirements of the application. For example, the stock prices could be displayed in a table, chart, or graph, and the user could have the ability to customize which stocks and data points are displayed. The user could also have the option to receive notifications or alerts when certain stock prices reach a certain threshold.

Diagram, text

Description automatically generated

You can find above a more detailed version of the Diagram, in this diagram, the Stock Price API is a server-side application that runs on a web server (e.g. Apache or Nginx). The API communicates with the Stock Market Data Provider to retrieve real-time stock prices and stores the data in a database (e.g., MySQL) and publishes the data to a Kafka queue. The database can be hosted on the same server as the API or on a separate server.

When a client (e.g., a mobile app or website), or the final user sends a request to the API for stock prices (Example: a user wants to display Euros Prices), the API retrieves the data from the database and returns it to the client in the appropriate format (e.g., JSON), this Json file, will be parsed in the backend, The client then displays the stock prices to the user on their device. For the watchlist the stock is subscribed to will be there in frontend engine lookup table and would be receiving the real-time stock updates.

**How to avoid deadlock(Use paritioning) and ensure data we get is in order as there is parallelism involved and also multi-threading where there are different workers working simultaneousli on same topic in different partition in kafka. So while consuming how do we préserver their order ?**

We would need to put timestamps in orders as the orders coming through same partitions will be sorted but when we are consuming different partition we can get random orders so hence storing by timestamp is important to get proper numbering

**What if the broker node fails ? i.e. Messages are being published but cant be consumed as worker is down ?**

Restart the queue or maintain an active and a passive worker that the user can take the data from in case a worker is down. Always maintain an excess of worker and machines. Use horizontal scaling to prevent this. Use data sink from where you can retrieve the latest snapshot/state of the data.

GTT

* 1. **Trade execution**

A trading app should allow users to place orders and execute trades in real-time. This may include features such as limit orders, stop orders, and market orders.

* **API layer**

The API layer is the interface between the front end of the trading app and the backend. It exposes a set of APIs that the front end can use to place orders and execute trades. The API layer may be implemented using a framework such as Express.js or Flask.

**Differences between Express.js and Flask and which one to use?**

Express.js is a JavaScript framework, while Flask is a Python framework. This means that you will use different programming languages to build applications with these frameworks.

* **Size and complexity:** Express.js are a lightweight framework that is easy to learn and use. It is well-suited for building simple and fast web applications. Flask, on the other hand, is a more full-featured framework that is more complex to learn and use but can be more flexible and powerful.
* **Routing**: Both frameworks provide routing capabilities, which allow you to define the URLs that the application will respond to. However, the syntax and capabilities of the routing systems differ between the two frameworks.
* **Middleware:** Both frameworks provide middleware capabilities, which allow you to define functions that are executed before or after a request is handled. However, the syntax and capabilities of the middleware systems differ between the two frameworks.

*In our app we will use, Flask because it’s more featured enhanced.*

*Here’s a simplified code in Flask, to create an API layer for trade execution in a trading app:*

*Text

Description automatically generated*

In this example, the execute\_trade\_api function is a Flask route that handles POST requests to the /trade endpoint. The function extracts the trade parameters (symbol, quantity, price, and side) from the request data and passes them to the execute\_trade function, which is responsible for executing the trade.

The execute\_trade function could be implemented in a variety of ways, depending on the specific requirements of the trading app. There can be a class for trade object from which the data is being taken. This is for informative purposes only, it might involve interacting with a brokerage API to place the trade, or it might involve executing the trade on a simulated trading platform.

Once the trade has been executed, the execute\_trade\_api function returns the result as a Fix/avro/JSON response to the client. We assume a JSON. The client could be a mobile app, a website, or any other application that can send Websocket requests and process JSON responses.

There are different methods of establishing connection between client and service provides:

1. Grpc
2. **HTTP Polling-SSL certification,**
3. TCP
4. UDP
5. **Websocket**
6. **SSE**

The connection will be established using Webhooks and SSE and http Polling in our case.

* **Business logic layer**

The business logic layer contains the code that handles the placement and execution of trades. It may include logic for validating orders, checking account balances, and communicating with exchanges and other financial institutions to execute trades.

Text

Description automatically generated

In this example, the execute\_trade function is responsible for validating the trade parameters and executing the trade.

The would be under the Risk Management module where it starts by checking that the symbol, quantity, price, and side parameters are all present and valid. If any of the parameters are invalid, it returns an error message.

Next, the function checks if the user has sufficient funds or assets to execute the trade. If the trade is a buy trade, it checks if the user has enough funds to cover the cost of the trade. If the trade is a sell trade, it checks if the user has enough assets to sell. If the user does not have sufficient funds or assets, it returns an error message.

Finally, if all the parameters are valid and the user has sufficient funds or assets, the function executes the trade by calling the execute\_buy\_trade or execute\_sell\_trade functions (which could be implemented in a variety of ways, depending on the specific requirements of the trading app). If the trade is successfully executed, it returns a success message.

* **Data storage layer**

The data storage layer is responsible for storing and managing data related to the trading app, such as trade data, market data, and customer data. It uses a database, to store this data.

There are many ways to architect a data storage layer, and the best approach will depend on the specific requirements of your application. Here are some general considerations to keep in mind when designing a data storage layer:

*Data model:* One of the first things to consider when designing a data storage layer is the data model, which determines the structure of the data that will be stored and how it will be organized. This might include decisions about the data types, relationships between entities, and how data will be queried and retrieved.

Here is a simple data model for a trading app that allows users to buy and sell stock

Text

Description automatically generated

In this data model, the User entity represents a user of the trading app. It includes information about the user's name, email, password (hashed for security), and balance (the amount of money they have available to trade).

The Stock entity represents a stock that can be traded on the app. It includes information about the stock's symbol, name, and current price.

The Trade entity represents a trade that has been executed by a user. It includes information about the user who executed the trade (referenced by the user\_id foreign key), the stock that was traded (referenced by the symbol foreign key), the quantity of shares that were traded, the price at which the trade was executed, the side of the trade (buy or sell), and the timestamp of the trade.

This is just one example of a data model for a trading app, and there are many other possible variations and additional entities that could be included. For example, you might want to include additional information about the user (e.g. address, phone number), or additional details about the stocks (e.g. industry, market capitalization).

*Data storage technology:* There are many different technologies that can be used to store data, including relational databases (e.g. MySQL, Postgres-AuroraQL), NoSQL databases (e.g. MongoDB, Cassandra), and in-memory data stores (e.g. Redis). Each technology has its own strengths and weaknesses, and it's important to choose the right one based on the needs of your application.

We use MongoDB, Postgres-AuroraQL and Redis

The data access pattern in a trading app will depend on the specific requirements of the application. Here are a few examples of the types of data access patterns that might be used in a trading app:

* **Real-time data:** A trading app might need to access real-time data about stock prices, trade volumes, and other market conditions to make informed trading decisions. This might involve making frequent requests to a stock market data API or subscribing to a real-time data feed.

Here is a simple diagram that illustrates how a real-time data access pattern might work in a trading app

[Trading App] <--subscribe-- [Real-time Data Feed/API]

[Trading App] <--store data-- [Database/Cache]

[Trading App] <--display data-- [User Interface]

In this diagram, the trading app is connected to a real-time data feed or API and subscribes to receive updates. As updates are received, the trading app processes the data and stores it in a database or cache. The trading app then displays the real-time data to users via the user interface.

* **Trade execution**: When a user places a trade, the trading app will need to access data about the user's account balance, the current price of the stock, and any other relevant details in order to execute the trade. This might involve making a single write request to the database to record the trade and update the user's balance or assets.

In this diagram, the user initiates a trade by interacting with the trading app. The trading app then accesses the cache/DB in case cache has failed to retrieve relevant data about the user's account balance, the current price of the stock from DB, and any other relevant details. Based on this data, the trading app determines whether the trade can be executed and, if so, updates the user's account balance and records the trade in the database.

[User] --> [Trading App] --> [Cache]/[Database]

This is a very simplified diagram, and the actual process of trade execution in a trading app may be more complex and involve additional steps or components. For example, the trading app might need to send the trade request to a brokerage or exchange in order to execute the trade, or it might need to perform additional checks and validation to ensure the trade is valid.

* **Trade history**: Users might want to view their past trades, which will require the trading app to access historical trade data from the database. This might involve running queries to retrieve specific trades or generating reports on the user's trading activity.

Create a database to store information about each trade, such as the trade date, the instruments traded (e.g. stocks, options, etc.), the quantity traded, and the price. We use Postgres-Aurora for storing this joined using userID as Foreign key OrderHistory table is there and User Table is there

You can find in the Following, steps needed to architecture the Trade History Backend

1. Set up a server-side application to handle requests for trade history data. This could be a web server running a language like Python or Java, or a serverless function running on a cloud platform like AWS Lambda. We will use a cloud platform AWS Lambda
2. The server-side application should include a set of APIs (Application Programming Interfaces) that allow clients (e.g. mobile apps or web apps) to retrieve trade history data from the database.(They can be get trade history for 1 year half year, get trade history for day, Get rejected order history etc..) The APIs should handle tasks such as authenticating requests, parsing parameters, and formatting the response data.
3. To improve performance and scalability, you may want to consider using a cache (e.g., Redis) to store frequently accessed trade history data. This will allow you to reduce the number of requests made to the database and can help to offload some of the load from the database server.
4. To ensure the security and privacy of trade history data, you should implement measures such as SSL/TLS encryption for all data transmissions and use secure authentication methods like OAuth2. You may also want to consider implementing role-based access controls to restrict access to trade history data based on the user's permissions.

Here is a diagram illustrating the data flow in the backend architecture for a trading app with a trade history feature:

[Client] <-- Websocket request --> [API Server] <-- Database query --> [Database]

In this diagram:

The client represents a mobile app or web app that is making a request to the API server for trade history data.

The API server is a server-side application that handles requests from clients and retrieves data from the database as needed.

The database is a storage system Postgres-AuroraQLthat contains the trade history data.

The API server and the database communicate via database queries.

Note: This diagram does not include a cache, but you could add one as an additional layer between the API server and the database:

[Client] <-- Webscoket request --> [API Server] <-- Cache query --> [Cache] <-- Database query --> [Database]

* **Stock research**: Users might want to research stocks and their historical performance, which will require the trading app to access data about stock prices, financial statements, and other relevant details. This might involve making requests to a stock market data API or running queries against a database of historical stock data.

Create a database to store information about the stocks being tracked by the app, such as the ticker symbol, the company name, and financial data such as earnings, revenue, and market capitalization. We use PostgreesSQL.

**Daily stock price will be in cache once market over cache dumped and then the session renews and there is only a write query to the data base to write the opening and closing price of the day.**

Set up a server-side application to handle requests for stock research data. Use server side application hosted on AWS could.

The server-side application should include a set of APIs (Application Programming Interfaces) that allow client to retrieve stock research data from the database. We use cache for performance and scalability

To keep the stock research data up to date, We set up a process for regularly importing new data from external sources such as financial news feeds or stock market data feeds. We use a real-time streaming data platform like Apache Kafka for getting the latest price.

These are just a few examples of the types of data access patterns that might be used in a trading app. The specific patterns

We can merge AWS Amazon S3, for more sensitive Data, Amazon S3, is a scalable, durable, and secure object storage service, we can also use Amazon Redshift for parts of the program that requires more speed.

*Things we considered when deciding the Solutions*

*Data access patterns:* Another important consideration is the data access patterns that will be used by the application. This includes things like the types of queries that will be run, the frequency of reads and writes, and the amount of data that will be accessed at once. These patterns will help determine the appropriate data storage technology and design patterns to use.

*We assumed that* ***half a million users*** *will be there and there be* ***40,000 queries per second at max load****. The write will be* ***1000ticktime=1 ms****. Read will be every seconds and also real-time. So our system is write heavy and read heavy because of which we used Pub/Sub and real time-streaming service like Kafka. The data will have* ***5000 symbols*** *and at max a user can access all the* ***5000 symbols*** *so there can be in total* ***50 million symbols*** *that would need to be accessed assuming every 1 lakh user subscribes to 5000 symbols*

*Data security:* It's important to consider the security of the data that is being stored, especially if it is sensitive or personal information. This might include things like encryption, access control, and backup and recovery strategies.

*We use multiple Backups, AWS has option for replications, We maintain active and passive queue and databases and backup our data in case of failure.* There are multiple nodes and workers and machines that we can use, and a Master-Slave architecture is followed. So that availability is not compromised.

*Scalability:* As the data storage needs of the application grow, it's important to design the data storage layer in a way that can scale with the needs of the application. This might include things like sharing, replication, and load balancing.

*We use Haproxy for load balancing and AWS for sharing and replication across multiple datacenters in multiple availability zones.*

* **Integration layer:** The integration layer is responsible for connecting the backend to external systems, such as exchanges and financial institutions. It may use APIs or other integration techniques to access real-time market data and execute trades.

Here are some steps we follow to build a backend integration layer for a trading app:

* Determine the types of integrations that are required by the app. This could include integrations with brokerage firms, exchanges, and financial data providers.
* Identify the API interfaces and protocols that will be used to communicate with each integration partner.
* Design the backend integration layer to handle the required API interfaces and protocols. This may involve creating custom code or using existing libraries or frameworks.
* Implement the backend integration layer, including error handling and retry logic to ensure robustness and reliability.
* Test the backend integration layer thoroughly, including simulating failures and other edge cases.
* Deploy the backend integration layer to a staging or production environment and monitor it for performance and stability.
* Maintain the backend integration layer over time, including updating it as necessary to support new features or changes to the integrations.
* Use version control in API’s to avoid failures. Discard API’s with multiple versions.
* **Monitoring and management layer:** The monitoring and management layer is responsible for monitoring the backend and ensuring that it is performing well. It may use tools such as CloudWatch or **Prometheus** and **Grafana** and Sentry and ELK stack to monitor the backend and identify any issues that may arise. ELK Stack Logs are important as we need the logs of the past few years for regulatory and compliance related issues.
  1. **Account management**

A trading app should provide users with the ability to manage their accounts, including depositing and withdrawing funds, viewing account balances and activity, and updating personal information.

We will divide the Account Management into several chunks, each chunk corresponding to a specific function or feature that the application should Include.

* + 1. **Login and authentication**

*Use Oauth2.0 for login and OTP Based-Login using twilio.*

Alternately, Users should be able to log in to the app using a secure authentication method, such as a username and password or biometric authentication.

First, you would need to create a database to store user accounts, including their login credentials (e.g., username and password). Use Postgres-Aurora

Next, you would need to create a server-side application to handle login requests from the client. This application could include an API endpoint that receives login requests and authenticates them using the credentials stored in the database. Use Salt Hashing for storing the passwords.

Here is an example of how this might look using the Flask framework in Python:

Text

Description automatically generated

* + 1. Account information

Here is an example of how you might retrieve account information for a trading app using a server-side language like Python:

First, you would need to create a database in postgres-Aurora to store user accounts, including their account information (e.g. contact details, account preferences, etc.).

Next, you would need to create a server-side application to handle requests for account information from the clientThis application could include an API endpoint that retrieves the account information from the database and returns it to the client.

Here is an example of how this might look using the Flask framework in Python:

Text

Description automatically generated

This code defines an API endpoint at the URL '/account/<user\_id>' that accepts GET requests and retrieves the account information for the user with the specified ID. It queries the database to retrieve the account information and returns it to the client as a JSON object. If the user is not found in the database, it returns an error message USER IS NOT FOUND and he would be redirected to the Registration page.

* + 1. **Security settings**

Here are some steps you could follow to properly set up security parameters for users in a trading app:

* Use secure authentication methods: To ensure that user accounts are secure, you should use strong and secure authentication methods, such as password hashing and salting, and possibly biometric authentication methods like fingerprint scanning or facial recognition.
* Implement two-factor authentication: You should consider offering two-factor authentication (2FA) as an additional layer of security for user accounts. This could involve using methods such as SMS or email-based verification or using a third-party authenticator app like Google Authenticator.
* Use SSL/TLS encryption: To ensure that data transmitted between the app and the server is secure, you should use SSL/TLS encryption for all data transmissions. This will help to protect against man-in-the-middle attacks and other types of cyber threats.
* Implement role-based access controls: Depending on the needs of your app, you may want to implement role-based access controls to restrict access to certain features or data based on the user's permissions. This could involve creating different user roles (e.g. admin, trader, etc.) and assigning specific permissions to each role. For Our version of the app we are not using role based access control but probably for the future iterations we could implement this. But for now we have not given priority to this feature.
* Regularly update and patch the app: To ensure that the app remains secure, you should regularly update and patch the app to fix any vulnerabilities or security issues that are discovered.
* Educate users about security: It is important to educate users about security best practices, such as using strong and unique passwords, enabling 2FA, and avoiding suspicious links or downloads. You could include security tips in the app or send out regular security notifications to users.
* Monitor for security threats: You should set up monitoring systems to detect and alert you to any security threats or breaches that occur. This could involve using tools like log analysis or intrusion detection systems.
  + 1. **Account balances and positions**

Users should be able to view and update their account information, such as their contact details and account preferences.

* + 1. **Trade history**

Users should be able to view their trade history, including details such as the date and time of each trade, the instruments traded, and the quantity and price.

* + 1. **Order history**

Users should be able to view their order history, including details such as the status of each order (e.g., filled, cancelled, etc.), the instruments ordered, and the quantity and price.

* + 1. **Deposits and withdrawals**

Users should be able to make deposits and withdrawals to and from their account and view the transaction history for these actions.

* + 1. **Alerts and notifications**

Users should be able to set up alerts and notifications for events such as changes in their account balance or positions, or the execution of a trade.

* + 1. **Customer support/Help/FAQ**

Users should have access to customer support, either through in-app chat or through other channels such as email or phone.

* + 1. **Compliance and regulatory requirements**

The account management feature should adhere to any relevant compliance and regulatory requirements, such as Know Your Customer (KYC) and anti-money laundering (AML) laws. There will also be batch processing where post-trade Batch Processing would be done and all the transaction will be written in the order book and the order book will be written and closed before the trading session begins the next day

Here are the steps for the same:

* + - Data from the exchanges is pulled using SFTP to (Amazon S3) bucket.
    - An application running on an (Amazon EC2) instances performs processing before the data is persisted. This processing comprises charge calculations, generation of contract notes, etc.
    - The processed data is persisted in MySQL databases running on Amazon EC2 instances.
    - An AWS Batch job is then triggered to extract the preprocessed data from the databases and populate the data into a Redis cluster to ensure faster I/O during processing.
    - The AWS Batch job then starts processing the records and persists the processed results back to the Redis cluster
    - An application running on an Amazon EC2 cluster then extracts the data from the Redis cluster, and performs post-processing such as removing unwanted data and adding customer-related data before persisting the data back into Emint’s databases.
  1. **Mobile access**

A trading app should be accessible from mobile devices, such as smartphones and tablets, to allow users to trade on the go.

Here are some steps you can follow to make the backend of a trading app work on multiple devices:

* + 1. **Design the backend to be device-agnostic:** The backend should be designed to work with any device that can make API requests, regardless of the operating system or hardware. This could involve using standards-based APIs and protocols, such as REST , JSON, FIX, FAST.
    2. **Use a cloud-based architecture:** By using a cloud-based architecture, you can ensure that the backend is accessible from any device with an internet connection. We use AWS Aurora for the same Also it frees us from the headache of managing the servers. And ensure replication, reliability and availability through auto scaling, horizontal scaling-More machine and vertical scaling-more RAM/CPU and also by providing resources on demand.
    3. **Implement a load balancer:** To ensure that the backend can handle a high volume of requests from multiple devices, you should consider implementing a load balancer to distribute the load across multiple servers. This can help to improve performance and scalability. We use Haproxy. We don’t use Nginx as we don’t have and specific rules that are required for load balancing or routing reverse proxies.
    4. **Implement caching and other performance optimization techniques:** To improve the performance of the backend for multiple devices, we implement caching and other optimization techniques, such as using a CDN (Content Delivery Network) to serve static content and using a database with good scalability and performance characteristics.
    5. **Monitor and test the backend**: To ensure that the backend is working correctly on multiple devices, we regularly monitor and test it to detect any issues or performance bottlenecks. We use chaos engineering/Stress testing to ensure our systems don’t crash at failures or bugs and proper error handling and log management is maintained
  1. **Educational resources**

A trading app may include educational resources, such as tutorials, webinars, and market analysis, to help users improve their trading skills.

Use streaming Service-Hyperlink to Youtube videos and display static content and add the quizzes etc, from the Postgres-Aurora database and display on the educational resources page.

* 1. **Research tools**

A trading app may include research tools, such as stock screener and fundamental analysis, to help users research and analyze potential trades. Use Money control API to fetch the resea4rch page tools

* 1. **Social Media**

Get followers list like, comment share, follow subscribe and most importantly learn. We will use a graph database and use ranking to rank the post relevant to the users based on graph connections and using graph algorithms recommend relevant people and content to follow.

* 1. **News Feed**

User will subscribe to the news feed of their choice or will get a default list of their recommended news related to their stock portfolios personalized based on the information gathered during the onboarding and the user's habits. For the stock market insights, we can use some APIs like Yahoo Finance, Bloomberg market, Zirra and Xignite

* 1. **Compliance with laws and regulations**

A trading app should be compliant with relevant laws and regulations, such as anti-money laundering (AML) and know-your-customer (KYC) requirements.. Upload document to amazon S3 and do manual screening or use a third party api to use ml to auto-verify. I suggest manual verification.

* 1. **Customization**

A trading app should allow users to customize their experience, such as by setting up alerts and notifications or creating watchlists of favorite securities.

1. **Choose a technology stack**

Select the technologies that you will use to build the backend of the trading app. This may include programming languages, frameworks, databases, and cloud services.

* 1. **Server-side language:**

This could be a language like Python, Java, or Node.js, which is used to build the backend of the app.

* 1. **Database:**

This could be a relational database management system (RDBMS) like MySQL or a NoSQL database like MongoDB, which is used to store data such as user accounts, trade history, and market data.

* 1. **Web server:**

This could be a web server like Apache or Nginx, which is used to serve the backend APIs and static assets (e.g., HTML, CSS, JavaScript) to the client.

* 1. **API framework:**

This could be a framework like Flask (Python) or Express (Node.js), which is used to build the APIs that the backend exposes to the client.

* 1. **Client-side language:**

This could be a language like HTML, CSS, and JavaScript, which is used to build the frontend of the app (e.g., the user interface).

* 1. **Mobile development framework:**

If the app is a native mobile app, this could be a framework like React Native (cross-platform) or Android Studio (Android) or Xcode (iOS), which is used to build the app.

* 1. **Charting library:**

This could be a library like Highcharts or D3.js, which is used to render charts and graphs in the app.

* 1. **Messaging and streaming platform:**

This could be a platform like Apache Kafka or RabbitMQ, which is used to handle real-time messaging and data streaming between the backend and the client.

* 1. **Cloud platform**:

If the app is deployed to the cloud, this could be a platform like AWS, Azure, or Google Cloud

1. **Implement the backend:**

Use the technology stack and system architecture design to build the backend of the trading app. This may involve creating APIs, implementing business logic, and integrating with external systems such as exchanges and financial institutions.

1. **Test and deploy the backend**:

Test the backend thoroughly to ensure that it is reliable and performs well. Once testing is complete, deploy the backend to a production environment, such as a cloud platform, to make it available to users.

* Test and deploy the backend: Once you have implemented the backend, you should test it thoroughly to ensure that it is working as expected. You may want to set up staging and production environments for testing and deployment. Once the backend is tested and ready for production, you can deploy it to the chosen hosting environment (e.g. on-premises, cloud, hybrid).

By following these steps, you can implement the backend for a trading app that is scalable, performant, and secure.

1. **Monitor and maintain the backend:**

Monitor the backend to ensure that it is performing well and identify any issues that may arise. Regularly update and maintain the backend to fix bugs, add new features, and improve performance.

**API’s to integrate**

Detailed KYC Verification API-Veriff

Help/Support Chatbot-Dialogflow

Basic monlith at start can expand to microservices later.

Razorpay/Plaid Api-Connecting to Bank accounts

**References** :

* [*https://www.moneycontrol.com/stock-price-widget/apis-india/AI65*](https://www.moneycontrol.com/stock-price-widget/apis-india/AI65)
* [*https://dltlabs.medium.com/demystifying-forward-and-reverse-proxies-e52eea330e85*](https://dltlabs.medium.com/demystifying-forward-and-reverse-proxies-e52eea330e85)
* [*https://www.linkedin.com/pulse/kafka-optimization-how-many-partitions-needed-maria-hatfield-phd/*](https://www.linkedin.com/pulse/kafka-optimization-how-many-partitions-needed-maria-hatfield-phd/)
* [*https://www.geeksforgeeks.org/design-scalable-system-like-instagram/*](https://www.geeksforgeeks.org/design-scalable-system-like-instagram/)
* [*https://www.investopedia.com/terms/o/oms.asp*](https://www.investopedia.com/terms/o/oms.asp)
* [*https://welpmagazine.com/the-complete-guide-to-trade-order-management-system/*](https://welpmagazine.com/the-complete-guide-to-trade-order-management-system/)
* [*https://zerodha.tech/blog/*](https://zerodha.tech/blog/)
* [*https://stackoverflow.com/questions/21173496/haproxy-vs-nginx*](https://stackoverflow.com/questions/21173496/haproxy-vs-nginx)
* [*https://github.com/hackingthemarkets/fullstack-trading-app*](https://github.com/hackingthemarkets/fullstack-trading-app)
* [*https://github.com/hackingthemarkets?tab=repositories*](https://github.com/hackingthemarkets?tab=repositories)
* [*https://hackingthemarkets.com/*](https://hackingthemarkets.com/)
* [*https://www.youtube.com/watch?v=XyvsiZsUetU&list=RDCMUCY2ifv8iH1Dsgjrz-h3lWLQ&start\_radio=1&rv=XyvsiZsUetU&t=27*](https://www.youtube.com/watch?v=XyvsiZsUetU&list=RDCMUCY2ifv8iH1Dsgjrz-h3lWLQ&start_radio=1&rv=XyvsiZsUetU&t=27)
* [*https://www.youtube.com/watch?v=Ut6dbvlRIAU*](https://www.youtube.com/watch?v=Ut6dbvlRIAU)
* [*https://www.youtube.com/watch?time\_continue=500&v=gQfaWHOrITI&feature=emb\_logo*](https://www.youtube.com/watch?time_continue=500&v=gQfaWHOrITI&feature=emb_logo)
* [*https://stackoverflow.com/questions/41701516/is-there-a-way-to-create-a-fingerprint-scanner-for-an-app-native-or-hybrid*](https://stackoverflow.com/questions/41701516/is-there-a-way-to-create-a-fingerprint-scanner-for-an-app-native-or-hybrid)
* [*https://www.quora.com/How-is-moneycontrol-com-getting-its-stock-data*](https://www.quora.com/How-is-moneycontrol-com-getting-its-stock-data)
* [*https://www.geeksforgeeks.org/two-factor-authentication-implementation-methods-and-bypasses/*](https://www.geeksforgeeks.org/two-factor-authentication-implementation-methods-and-bypasses/)
* [*https://bootcamp.uxdesign.cc/how-to-build-a-stock-trading-platform-like-a-pro-ddcd7160e684*](https://bootcamp.uxdesign.cc/how-to-build-a-stock-trading-platform-like-a-pro-ddcd7160e684)
* [*https://www.softwareatscale.dev/p/software-at-scale-37-building-zerodha#details*](https://www.softwareatscale.dev/p/software-at-scale-37-building-zerodha#details)
* [*https://github.com/zerodha*](https://github.com/zerodha)
* [*https://aws.amazon.com/blogs/industries/how-zerodha-reduced-their-backend-processing-time-from-hours-to-minutes-using-aws-batch/*](https://aws.amazon.com/blogs/industries/how-zerodha-reduced-their-backend-processing-time-from-hours-to-minutes-using-aws-batch/)
* [*https://www.softwareatscale.dev/p/software-at-scale-37-building-zerodha*](https://www.softwareatscale.dev/p/software-at-scale-37-building-zerodha)
* [*https://ajinasokan.com/posts/flutter-zero-to-production/*](https://ajinasokan.com/posts/flutter-zero-to-production/)
* [*https://screenroot.com/stock-trading-capital-market-app.php*](https://screenroot.com/stock-trading-capital-market-app.php)
* [*https://thinksoftware.medium.com/robinhood-backend-system-design-how-to-receive-realtime-stock-updates-56cd0009bd0*](https://thinksoftware.medium.com/robinhood-backend-system-design-how-to-receive-realtime-stock-updates-56cd0009bd0)
* [*https://www.youtube.com/watch?v=dUMWMZmMsVE*](https://www.youtube.com/watch?v=dUMWMZmMsVE)
* [*https://www.youtube.com/watch?v=eNDADqa9858*](https://www.youtube.com/watch?v=eNDADqa9858)
* [*https://www.youtube.com/watch?v=AOD2qPiMmH8*](https://www.youtube.com/watch?v=AOD2qPiMmH8)
* [*https://www.devteam.space/blog/how-to-build-a-trading-platform/*](https://www.devteam.space/blog/how-to-build-a-trading-platform/)
* [*https://topflightapps.com/ideas/how-to-develop-a-trading-and-investment-app/*](https://topflightapps.com/ideas/how-to-develop-a-trading-and-investment-app/)
* [*https://easternpeak.com/blog/stock-market-apps-how-to-build-a-trading-platform/*](https://easternpeak.com/blog/stock-market-apps-how-to-build-a-trading-platform/)
* [*https://www.merixstudio.com/blog/build-stock-trading-app-like-robinhood/*](https://www.merixstudio.com/blog/build-stock-trading-app-like-robinhood/)