Equity Trading System

Members: Eitan Leitner, Yisroel Newmark, Sam Orenbakh, Mati Salem, Yoni Stern, Yoseph Teitelbaum

Project Goals

Produce an Equity Trading System to support:

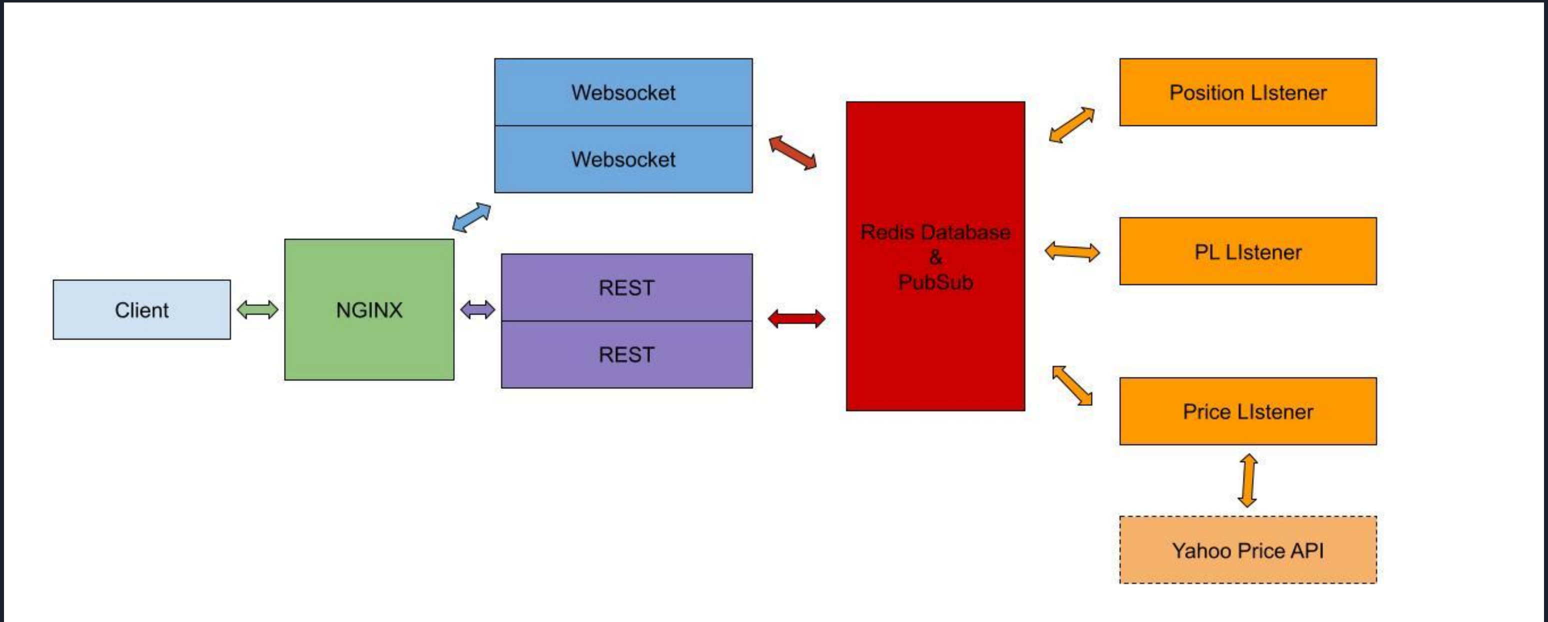
- Position management
- Trade capture
- Live P&L for Equity Products

With a fault tolerant and scalable backend solution.

Architectural Goals

- Loose coupling between components allows for:
  - Scalability
    - Able to spin up more services with specific functionalities as they are needed
  - Resiliency
    - If one component goes down, the other components can still run
  - Recovery
    - If a component runs into an error it can be restarted and rebuild its data

Architectural Diagram



Infrastructure Components

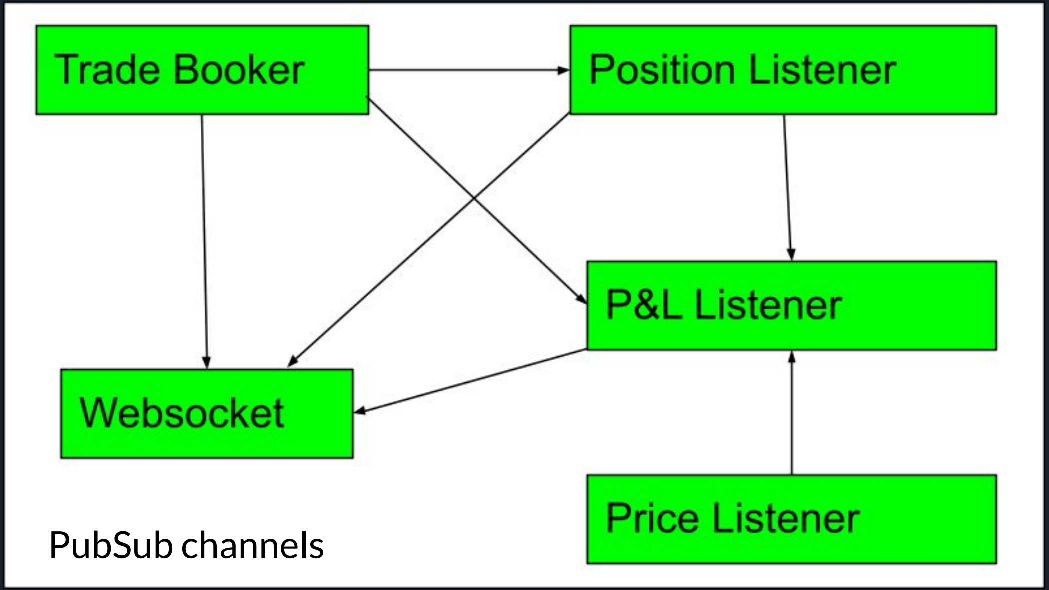
Backend Services

Redis: High performance way of storing data

Rest API: Standard request response protocol

Listeners: Loose coupling between components

Websockets: For real time data updating



Redis

- What is Redis?
    - Redis is an open-source, in-memory data structure store that we use as a database.
- Why Redis?
    - We use redis for its fast in-memory storage, allowing quick access to manage trades and positions data, and providing us with low latency updates.
    - Redis includes notification services (pubsub) for loose coupling between components.
- How do we use it?
    - Storing and Retrieving Positions
    - Managing Trades and Profit & Loss (P&L)
    - Real-time Price Handling
    - Publishing Real-time Trade and Position Updates
    - Query and Analysis
    - Startup/Recovery Configuration

Data Model

- **Trade:** Id, account, buy or sell, ticker, quantity, time booked, user, price, version (Key: Id+Account)
  - **Profit Loss:** trade pl, price when calculated, time last updated
- **Trade History** (for auditing)**:** list of trades, current version
- **Position:** account, ticker, quantity, time last updated (Key: account + ticker)
  - **Profit Loss:** trade pnl, position pnl
  - **Position Snapshot:** (stored with date)
- **Live Price:** price, ticker, time of price

Get Trade

```
key = f"trades:{account}:{date.isoformat()}"
history_json = client.hget(key, id)
if history_json == None:
    return default
return History.parse_raw(history_json).get_current_trade()
```

Set Position

```
key = "positions:"+account
client.hset(key, ticker, position.json())
```

Get Position

```
key = "positions:"+account
    json_position = client.hget(key, ticker)
    if json_position == None:
        return None
    return Position.parse_raw(json_position)
```

Set Price

```
key = "livePrices:" + stock_ticker
client.hset(key, date.isoformat(), price.json())
```

Get Price

```
key = "livePrices:"+ticker
price_json = client.hget(key, date.isoformat())
if price_json is None:
    return None
return Price.parse_raw(price_json)
```

REST API

- What is a REST API?
  - REST (Representational State Transfer) is a set of guidelines for how to design and build applications that interact over HTTP.
  - RESTful APIs use standard HTTP methods like GET, POST, PUT, DELETE, etc., to perform operations and transfer data between the client and server.
  - It treats server-side resources (data objects) as entities that can be created, read, updated, and deleted.
- FastAPI
  - FastAPI is a modern, easy-to-use, fast (high-performance), web framework for building APIs with Python.
  - It is highly scalable and comes with automatic interactive API documentation.
  - It is fast, easy of use, and has the ability to rapidly build and iterate on the API due to the automatic request and response serialization.
- Async and await
  - We make use of async and await keywords for asynchronous programming, allowing our server to handle many requests at the same time, improving the overall performance.
- APIRouter
  - We use the APIRouter class to create modular, mountable groups of routes, providing clear and maintainable code structure.

Trade Booker

- Trade booking process:
  - Receive trade booking request
  - Trade validation
  - Booking (save to data store)
  - History management
  - Confirmation
- Bulk Trade Booking and Updates:
  - Built-in
  - CSV

```python
def booktrade(client: redis.Redis, trade: Trade, tickers: ValidTickers):
    if not tickers.is_valid_ticker(trade.stock_ticker):
        raise HTTPException(status_code= 400, detail= "invalid stock ticker")
    history= History()
    history.trades.append(trade)
    redis_utils.set_history(client, trade.account, trade.date, trade.id, history)
    trade_amount = trade.get_amount()
    redis_utils.publish_trade_info(client, trade.account, trade.stock_ticker, trade_amount,
    trade.date, trade.time)
    redis_utils.publish_trade_update(client, trade.id, trade.account, trade.stock_ticker,
    trade_amount, trade.price, trade.date)
    client.publish("tradeUpdatesWS", f"create: {trade.json()}")
    redis_utils.add_to_stocks(client, trade.account, trade.stock_ticker)
    return {"message" : "trade booked successfully", "id" : trade.id}
```

```python
def set_history(client: Redis, account: str, date: date_obj, id: str, history: History):
    key = f"trades:{account}:{date.isoformat()}"
    json_data= history.json()
    client.hset(key, id, json_data)
```

Price Listener

- Updates prices of all stocks in S&P 500 every minute when the markets open
- Stores prices by date so we can get prices of stocks from previous days
- Sets closing prices once the market closes (if not all closing prices are calculated rerun up to 5 times)
- Recover:
  - Rebuild: set closing prices for every day since startup
  - Recover: set closing prices for last 5 days

```
"livePrices:{stock_ticker}", date
```

```python
def schedule_jobs(scheduler: BlockingScheduler, start_date: date_obj):
    now= datetime.now()
    end_date= now.date() - timedelta(1) if now.time() < market_calendar.closing_time else now.date()
    price_updates_trigger= OrTrigger(triggers= [CronTrigger(day_of_week= "0-4", hour= "10-15", minute= "*"), CronTrigger(day_of_week= "0-4", hour= 9, minute= "30-59")])
    closing_price_updates_trigger= OrTrigger(triggers= [CronTrigger(day_of_week= "0-4", hour= "16"), CronTrigger(day_of_week= "0-4", hour= 23, minute= 50)])
    scheduler.add_job(func= fill_in_closing_prices, args= [start_date, end_date])
    scheduler.add_job(func= update_stock_prices, trigger= price_updates_trigger)
    scheduler.add_job(func= fill_in_closing_prices, trigger= closing_price_updates_trigger)
```

Listener Base Class

- Extended by Position Listener and P&L Listener
- Threads:
  - Subscriber Thread: Receives JSON from pubsub then adds function and parameters (received from pubsub) to queue
  - Queue Processor Thread: gets function from queue and runs it
  - Main Thread: Preforms startup/recovery

```python
class listener_base(ABC):
    def __init__(self, client= get_redis_client()):
        self.queue= Queue()
        self.client= client
        self.sub= self.client.pubsub(ignore_subscribe_messages= True)
        self.sub.subscribe(**self.get_handlers())
        self.queue_processor_thread= Thread(target= self.process_queue, daemon= True)

    def start(self):
        self.subscriber_thread= self.sub.run_in_thread()
        self.startup()
        self.queue_processor_thread.start()
        signal.signal(signal.SIGTERM, self.termination_handler)

    def process_queue(self):
        while True:
            func, args= self.queue.get()
            func(**args)
            self.queue.task_done()

    def termination_handler(self, signum, frame):
        self.subscriber_thread.stop()
        self.subscriber_thread.join()
        self.queue.join()
        self.sub.close()
        self.client.close()

    def startup(self):
        mode= os.getenv("RECOVERY_MODE")
        if mode == "rebuild":
            self.rebuild()
        elif mode == "recover":
            self.recover()

    @abstractmethod
    def rebuild():
        pass
```

Position Listener

- Position: how many shares of a stock is owned
- Updates positions every time a new trade is booked or a trade is updated
- Creates position snapshot every trading day at 4:00 PM
- Uses a cache to keep track of positions so it doesn't have to retrieve position from the database every time its needed

- Recovery:
  - Rebuild: deletes all positions + position snapshots and recalculates them using trades in the system
  - Recover: uses last aggregation time to only recover trades booked when the position listener was down

```python
def update_position(self, account: str, stock_ticker: str, amount_added: int, now: datetime):
    stock_tickers= self.cache.get(account, dict())
    self.cache[account]= stock_tickers
    old_amount= stock_tickers.get(stock_ticker)
    if old_amount == None:
        old_position= redis_utils.get_position(self.client, account, stock_ticker)
        old_amount= old_position.amount if old_position != None else 0
    amount= old_amount + amount_added

    stock_tickers[stock_ticker]= amount
    position= Position(account= account, stock_ticker= stock_ticker, amount= amount,
                        last_aggregation_time= now, last_aggregation_host= "host")

    redis_utils.set_position(self.client, account, stock_ticker, position)
    self.client.publish("positionUpdatesWS", f"position:{position.json()}")
```

P&L Listener

Trade P&L: (closing price - trade price) * trade quantity
Position P&L: (live price - closing price) * position
Total P&L: Trade P&L + Position P&L

- Trade P&L (Realized P&L): money you gained or lost based off stocks you bought or sold (money from selling stock - money spent buying stock)
- Position P&L (Unrealized P&L): money you would make if you sold all your stocks at their current prices
- Calculates P&L by day
- Depends on position, trade and price data to calculate P&L
- Recovery:
  - Rebuild: delete all P&L data and performs a complete rebuild
  - Recover: deletes todays P&L data then recalculates it

Websocket

- Websocket
  - Used for real time updates to frontend
- Endpoints for trades and positions
  - Receives data from pubsub channels
  - Merges base data with p&l data
  - Filters data sent to clients by accounts

```python
self.pubsub.subscribe(**{
  key: self._handler(value) for key, value in channels.items()
})

def _handler(self, func):
    def handle(data):
        account = data["account"]
        if func(self.client, data) != None:
            data = data | func(self.client, data).dict()
            data["pnl_valid"] = True
        else:
            data["pnl_valid"] = False
        for connection in self.connections.values():
            if connection.has_account(account):
                connection.add_message(
                    {"type": type, "payload": data}
                )
    return handle
```

Frontend Services

Svelte:

- Emerging web framework
- merges concepts from other popular frameworks in order to create compact clean code.

Nginx:

- Internet scale reverse proxy and serves static html content

Svelte

- Why Svelte?
  - Svelte allows for creating reusable components.
  - These components are written in HTML, CSS, and JavaScripts blocks.
  - Svelte allows for simple Higher order 2 way binding within and between components.

```
<script>
    import {Button} from "flowbite-svelte";

    let responseData;

    async function fetchData() {
        try {
        const response = await fetch('/api/hello');

        if (response.ok) {
            responseData = await response.json();
        } else {
            console.error('Error:', response.status);
        }
        } catch (error) {
        console.error('Error:', error);
        }
    }
</script>

<div>
  <Button on:click={fetchData}>hello</Button>
</div>

{#if responseData !== undefined}
  <h1>{responseData.hello}</h1>
{/if}

<style>

</style>
```

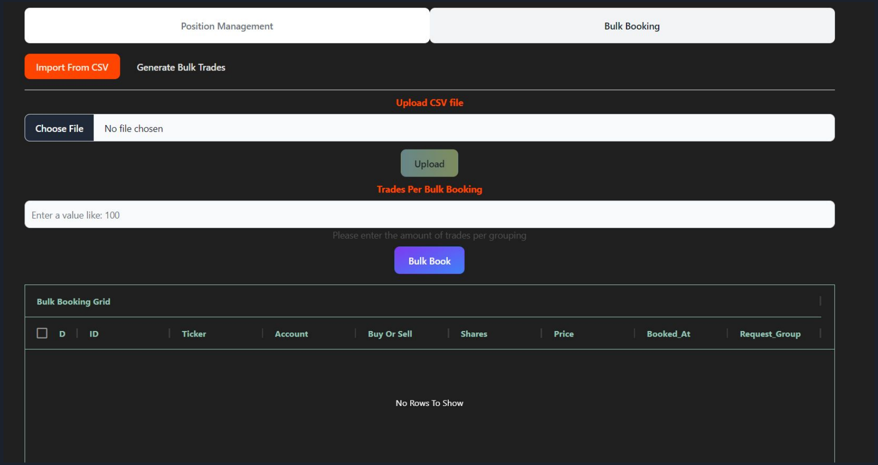
```
export let tradeData = [];

export let deleteCall = false;

export let buttonName;

export let submitTrade = false;

export let amountOfTradesPerGrouping;
```

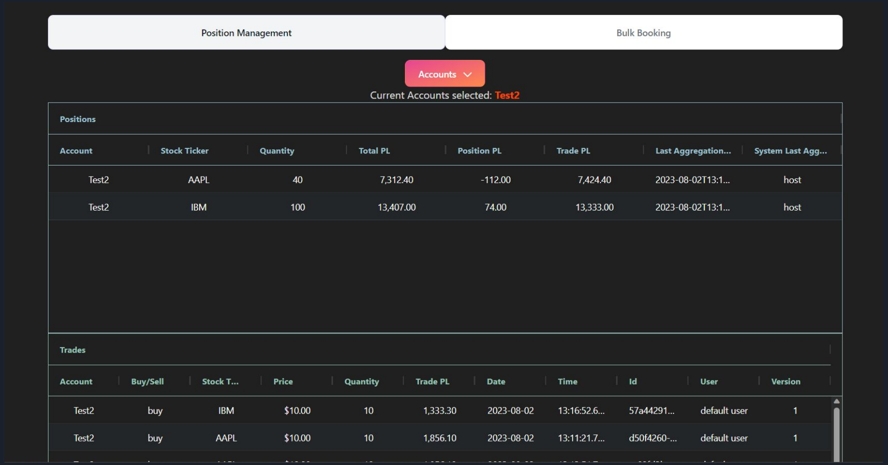
2 way binding between our Bulk Booking Form and our Bulk Booking Grid.

```
bind:tradeData = {tradeData} bind:deleteCall = {deleteCall} bind:buttonName = {buttonName} bind:submitTrade = {submitTrade} bind:amountOfTradesPerGrouping = {amountOfTradesPerGrouping}
```

UI Components

- Booking Protocol
    - Upload or generate data -> send Rest API call -> Return GUID and display it.





Data is generated or uploaded through the Bulk Booking Portal and displayed In our Trade and Positions grids

NGINX

- NGINX?
  - “**NGINX** is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more.” (https://www.nginx.com/resources/glossary/nginx/)
  - Translation: NGINX allows us to run servers, facilitate communication between servers, and balance server requests for optimal performance

```
upstream api_group {
  least_conn;
  ${API_GROUP}
}
upstream ws_group {
  least_conn;
  ${WS_GROUP}
}
```

These lines control the load balancing. Routes the request to the least used server in the appropriate network group created by the docker compose

```
server {
  listen 80;
  location / {
    root /usr/share/nginx/html;
    try_files $uri /index.html;
  }
```

These lines find the code for the frontend web page and displays that page when the server is accessed on the root endpoint

```
location /api/ {
  proxy_set_header X-Real-IP $remote_addr;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-NginX-Proxy true;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection "upgrade";
  proxy_pass http://api_group/;
  proxy_ssl_session_reuse off;
  proxy_set_header Host $http_host;
  proxy_cache_bypass $http_upgrade;
  proxy_redirect off;
  proxy_http_version 1.1;
}
```

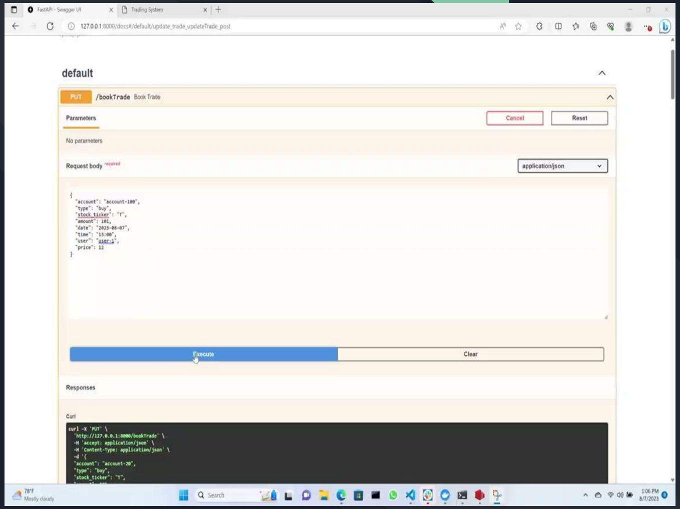
All websocket and REST requests from the front end go to frontend/api/{request} which NGINX connects to the correct server server/{request}

AWS Infrastructure

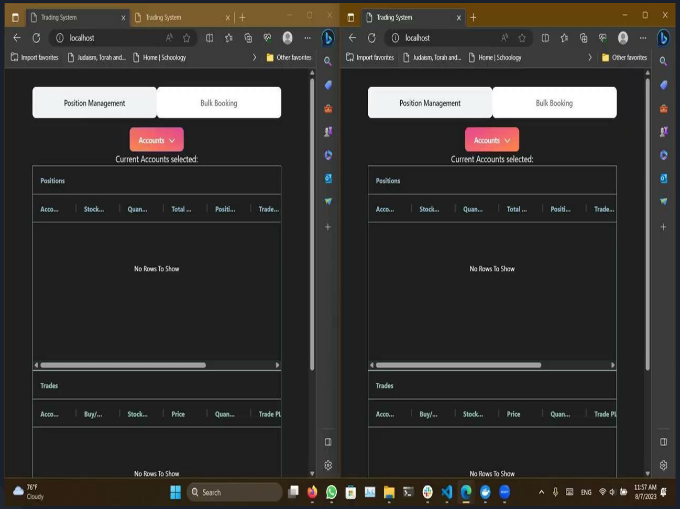
- Made up of two ECS clusters:
    - ECS allows us to rapidly spin up and down docker images
    - General cluster with t2.micro
        - 1 GiB RAM, 1 vCPU
    - Redis cluster with t2.medium
        - 4 GiB RAM, 2 vCPUs
- Service Connect for internal communication
    - Maps container ports to a subnet of the Virtual Private Cloud
- Application load balancer for external access
    - Uses a secondary nginx instance if first down
- Changes to repository automatically redeployed if automated tests pass
    - This is achieved through Github Actions (Github's CI/CD tool)

Demo

Recovery + Update Trade Demo



UI Demo



Conclusions

- **Reflections:**
    - Learned how to build modern N-tier application
    - Efficient teamwork
    - There is a cost for building for resiliency
    - Cloud infrastructure
    - Industry practices
- **Next Steps:**
    - Additional components
        - Improve performance for larger data sets.
        - Sharding Listeners
    - Create a more cohesive UI
    - Live price display