ITSA Project Report Team

Git Name: Round-Robin

Team Members:
Hong Yang
Ng Yujin Benjamin
Lo Huifen Ferne
Ong Chin Hao
Muhammad Yazid
Jasky Ong Qing Hao

Background & Business Needs

With the evolving need to cater to an increasing digitalized audience, stock exchanges must embrace technology to ensure that it will remain relevant in today's day and age. Therefore, Round-Robin was approached to create a scalable, secure IT application that provides an interface for traders to view, buy and sell shares without the need to engage a broker.

Currently, the business is using a 3rd party API to access the ask and bid prices for some listed stocks and keeps a database of historical transactions for its clients to be able to request them and conduct their own technical analysis of the underlying stock. The proposed application must cater to the high availability of at least 100 transactions every second with limited downtime. Furthermore, it must be able to provide its users with a resolution of at most 1 second for all the supported stocks.

Stakeholders

Business Stakeholder	**Description**
Trader	- End-user of the application - The bulk of the users for this application
Government and Regulatory Authorities	- Auditing body - Occasionally requires information in order to audit the company or individual

IT Stakeholder	**Description**
Database Administrator	- The person-in-charge of maintaining the database - Ensure high availability of database elements of the application
Backend Developer	- The person-in-charge of exposing and integrating APIs to the application - Develop and build new interfaces for the end-users

Key Use Cases

Account Creation	
ID	001
Description	This function allows new users to sign up for the service.
Actors	1. Traders
Main Flow of Events	1. User wishes to create account 2. User clicks on sign-up button on the login page 3. User is redirected to a user form 4. User enters valid information into the form 5. Information is validated on the account server 6. User is redirected to the main navigation page, logged in.
Alternative FoE	NIL
Pre-Conditions	NIL
Post-Conditions	User is redirected to navigation page

User Authentication	
ID	002
Description	Due to different users on the system, proper access control must be determined during login. Furthermore, users expect that their account information are kept confidential. Hence this function must allow proper access rights to be provided to the users
Actors	1. Traders 2. Authentication Server

Main Flow of Events	1. User keys in credentials on log-in page 2. Credentials are sent to the authentication server 3. Server validates credentials and assign specific access rights back to the user 4. Upon obtaining valid credentials, a session object containing the credentials is returned and user is redirected to the main navigation page
Alternative FoE	1. User keys in wrong credentials on log-in page 2. Server returns that wrong credentials 3. Upon several successive retries, the account is locked, and the information is sent to the user that successive unsuccessful login attempts have been made
Pre-Conditions	Account creation function Proper routing and credentials database User has an account in the system
Post-Conditions	User is logged in with proper access rights given

Submitting a Purchase Request	
ID	003
Description	A fundamental part of the functionality of the application is allowing traders to be able to submit bids for the purchase on any of the listed financial instruments on the platform.
Actors	1. Traders 2. Bidding Server 3. 3rd Party API for live data on stock prices 4. Notification Server

Main Flow of Events	1. User is presented with the ask and bid prices on the buy page 2. User keys in the exact amount of stock he wants to purchase 3. User submits confirmation 4. Trading application validates that the user has sufficient balance to purchase that amount of stock and passes all compliance checks 5. Purchase order is then stored on a Priority Queue, based on asking price and then time. 6. Upon a sell order that is below the purchase order asking price the trade is complete. 7. Stock will be credited to the buyer account and funds will be credited to the sellers account. 8. A notification is sent to the buyer
Alternative FoE	- The event that the buyer is looking to remove the order submitted
Pre-Conditions	Ensuring that the queue does not lose any information on a server failure Enough balance in the buyers account Enough stock in the sellers account
Post-Conditions	A completed trade

Submitting a Sell Request	
ID	004
Description	A fundamental part of the functionality of the application is allowing traders to be able to submit sell orders on any of the listed financial instruments on the platform.
Actors	1. Traders 2. Bidding Server 3. 3rd Party API for live data on stock prices 4. Notification Server

Main Flow of Events	1. User is presented with the ask and bid prices on the sell page 2. User keys in the exact amount of stock he wants to sell 3. User submits confirmation 4. Trading application validates that the user has enough stock to sell and passes all compliance checks 5. Sell order is then stored on a Priority Queue, based on asking price and then time. 6. Upon a purchase order that is above the sell order asking price the trade is complete. 7. Stock will be credited to the buyer account and funds will be credited to the sellers account.
Alternative FoE	NIL
Pre-Conditions	Ensuring that the queue does not lose any information on a server failure Enough balance in the buyers account Enough stock in the sellers account
Post-Conditions	A completed trade

Submitting a Transfer Request	
ID	005
Description	An additional feature for the application would be for 2 individual traders to submit a transfer of shares between the 2 parties. This however is subjected to IRAS for payment of stamp duties and a record must be present in the database
Actors	1. Traders 2. Notification Server

Main Flow of Events	1. Upon entering the transfer page, trading application gets owned stocks of current user (sender) from the Purchased database, returns a response and displays the data. 2. Sender enters the receiver's id. 3. Trading application validates the receiver's id with the database and displays the corresponding name. 4. Sender selects the stock, keys in the quantity of each selected stock to transfer to the receiver. 5. Upon confirmation of details by the sender, trading application stores transaction details in the database, bypassing the typical bidding and payment stage of buying and selling stocks. 6. Once the sender's stock has been successfully transferred, the sender will receive an email notification on the successful transfer from the notification server. The receiver will also receive a notification of which stocks and their quantities have been transferred over from the sender.
Alternative FoE	1. Sender enters an invalid receiver id 2. Trading application validates the receiver's id with the database and displays an error message. 3. Sender is unable to proceed with selecting the stocks to transfer until a valid receiver id is entered.
Pre-Conditions	User (sender) has already logged in to the Trading application. User (sender) must have already owned stocks. User (sender) knows the receiving user’s id.
Post-Conditions	NIL

Account Crediting Function API	
ID	006
Description	The application must allow a simple API for users to add balance to their accounts. The proposed methodology is to use Stripe to allow users to tag their trading account to their credit card to support easy transfer
Actors	1. Traders 2. Notification Server 3. Stripe API

Main Flow of Events	1. On account creation function (001), the user receives an email tied to his account to add a stripe account. 2. User is navigated to stripe in order to link his trading account with a stripe account. 3. User would then be able to top-up any amount of money to his trading account 4. Stripe server processes the payment and returns a successful response if the amount is successfully credited 5. User receives a notification on that his account has been successfully credited
Alternative FoE	NIL
Pre-Conditions	User has a Stripe Account
Post-Conditions	User's trading account is successfully credited

Key Architectural Decisions

Architectural Decision - Reverse Proxy	
ID	001
Issue	1. To route messages to the correct destination 2. Potential attacks to the backend servers
Architectural Decision	By implementing a reverse proxy, our application is able to use content-based routing to route the client's request to the correct backend server. It is also able to hide its identity and acts as an additional defense against security attacks.
Assumptions	NIL
Alternatives	Connecting Straight to the Servers
Justification	Better Security, No CORS Problem, Centralised point for request logging than a straight connection to the servers

Architectural Decision - Failover (Active-Passive)	
ID	002
Issue	1. Key components of application to be available during operation hours 2. Failure of active server
Architectural Decision	Upon failover, the redundant/passive server will takeover and all requests coming from the client will go to that server. By implementing failover, the system will be able to continue to operate without any disturbance to the client even when one of its servers is down.
Assumptions	NIL
Alternatives	NIL
Justification	NIL

Architectural Decision - Separate Authentication Server	
ID	003
Issue	1. Any small change to app requires the entire app to be built and deployed 2. Potential security attacks
Architectural Decision	Achieved separation of concern by separating authentication server and backend server as this allows changes to be made on either ends without impacting one another. As authentication server will be having sensitive information, it will be placed in the lowest level of the network (internal zone 2) to achieve better security
Assumptions	NIL
Alternatives	Monolithic Application
Justification	User Authentication is usually requires a higher level of security due to user privacy issues. By decoupling we can manage the needed compliance better.

Architectural Decision - Load Balancing Backend	
ID	004
Issue	1. Too many requests coming in at once to the backend server which degrades performance 2. Requests that are going to the backend server requires more computation time
Architectural Decision	As the frontend server is not computationally heavy, the request coming from the frontend to the backend requires more computation time thus our team has decided to implement a load balancer at the frontend server. By implementing a load balancer at the frontend, the application is able to route requests to the two instances of the backend server via round robin which will ensures that no one server is overworked and requests are distributed evenly.
Assumptions	NIL
Alternatives	NIL
Justification	NIL

Architectural Decision - Master-Slave Database Cluster	
ID	005
Issue	1. Failure of database
Architectural Decision	Amazon's Aurora relational database is selected for its automatic failover mechanism, ensuring high availability of the backend server's data. This is achieved through database replication, where the slave takes over requests in the short period that the master is down (and automatically recovers).
Assumptions	NIL
Alternatives	NIL (This is the only available cluster in AWS)
Justification	NIL

Development View

Development Workflow

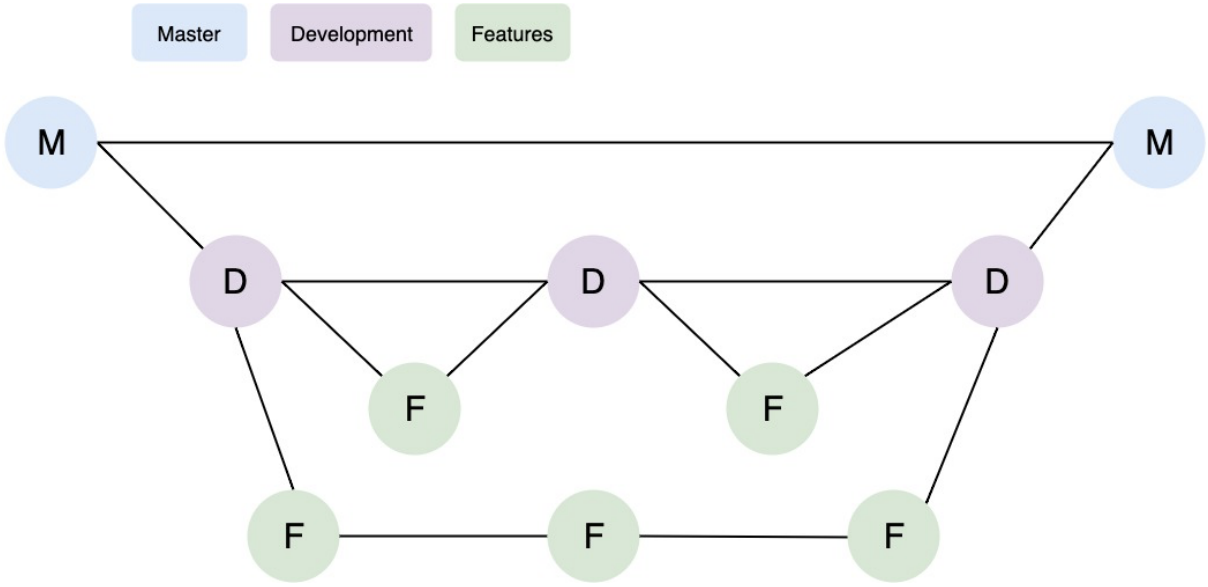

Figure 1.1: Development Workflow

The development workflow that our team will be using is shown in the diagram above. From the master branch, we have branch out to a development branch where all of the features that are needed will be compiled and tested. Multiple feature branches will branch out from the development branch where the developers will focus on developing the specific function required. Once a feature has been completed on a feature branch, the completed feature branch creates a pull request to the development branch to merge the changes, Travis CI will trigger at this point. Once the development branch is production-ready and bug free, it will be pushed to the master branch.

Travis CI Integration

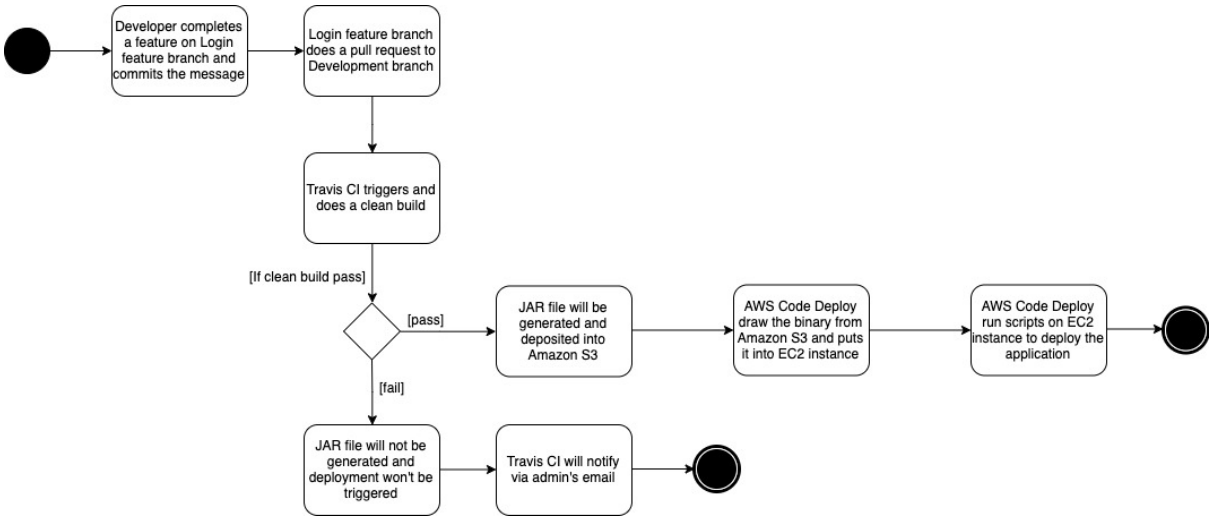

Figure 1.2: Travis CI Integration Diagram

Our team has implemented Travis CI inside the Development workflow. Travis CI will trigger when a feature branch creates a pull request or when there is a push to the development branch.

The flow of how Travis CI will trigger has been shown in the activity diagram shown above where the scenario will be that the developer completes a feature on Login feature branch and is ready to create a pull request to the development branch.

Development Strategy

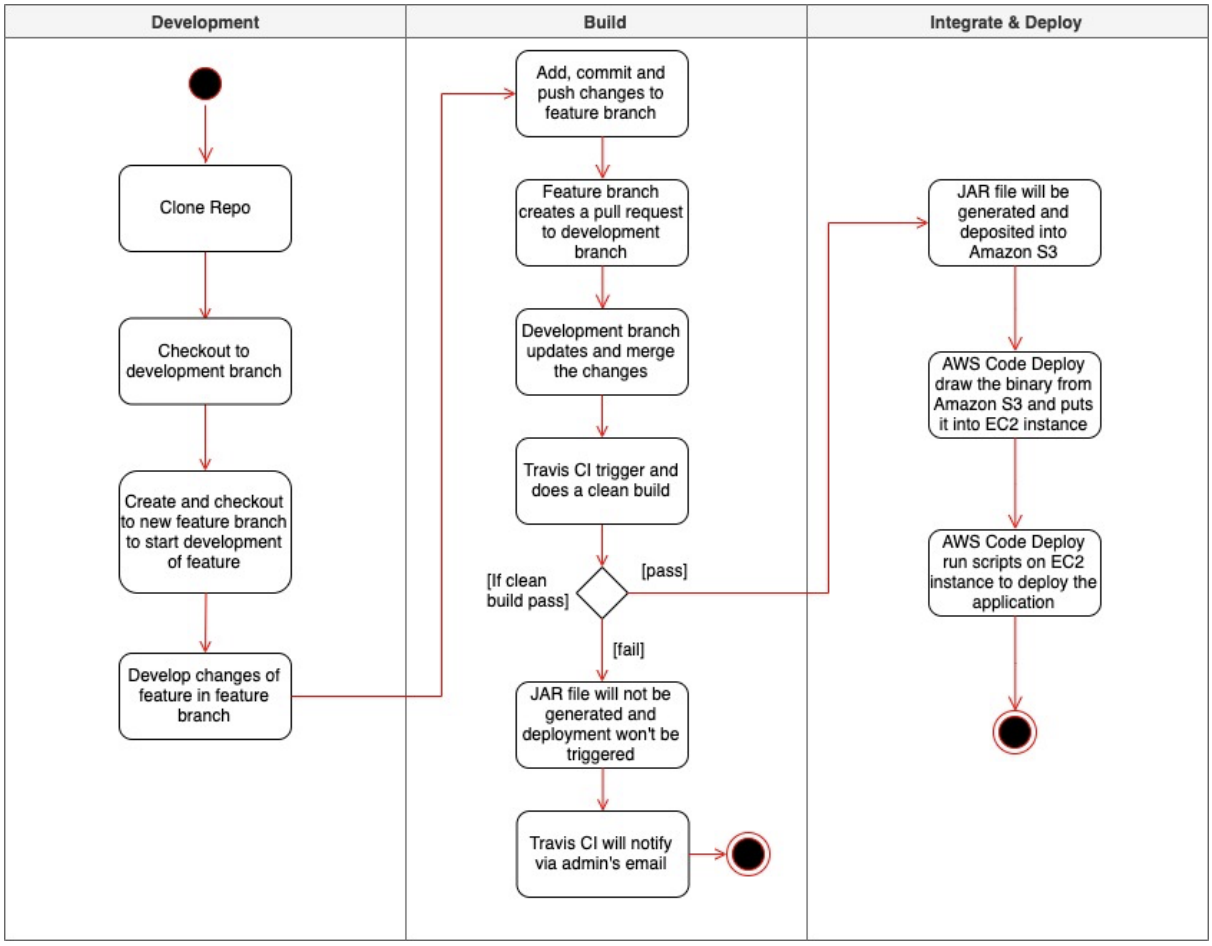

Figure 1.3: Development Strategy Diagram

Solution View

Ease of Maintainability

Our team decouples our trading application for easier maintainability. This ensures that it is easier to target and diagnose any integration problems that we may face along the way. We also incorporate software design patterns and integration patterns into our application. As only one logger in the trading server is required, the singleton design pattern is implemented. APIs are also used to expose the separate functionalities of both the trading server and authentication server, allowing the trading application to call the endpoints as required. The Amazon Relational Database Service (RDS) instance that the trading server uses implements database replication where it replicates the data to a standby instance that will takeover handling requests in the event of the database failing.

Deployment Diagram

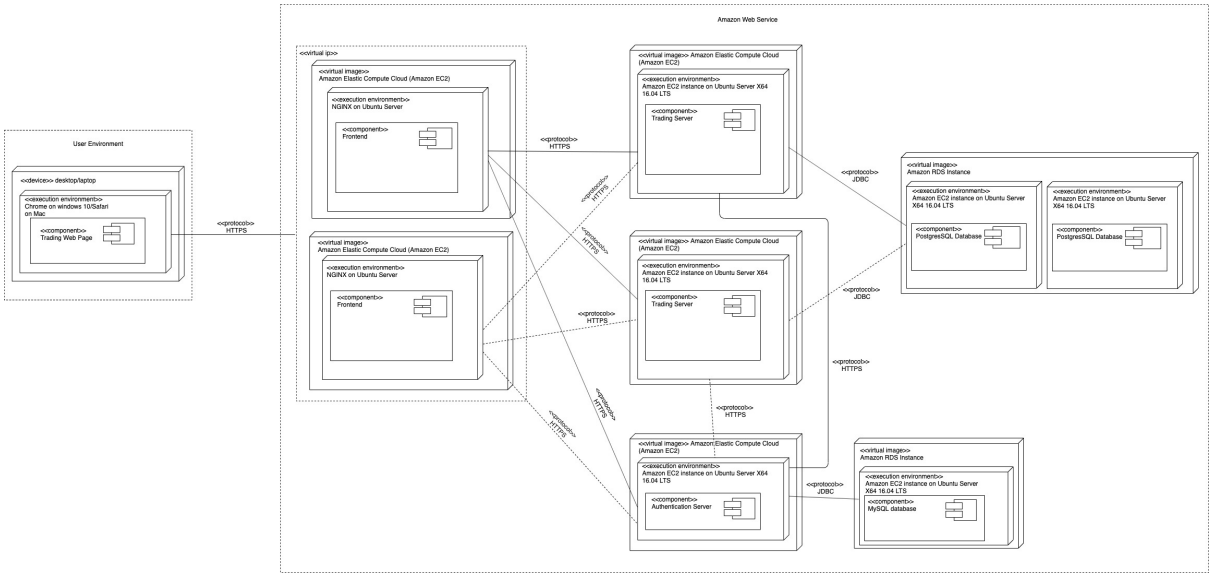

Figure 1.4: Deployment Diagram

Network Diagram

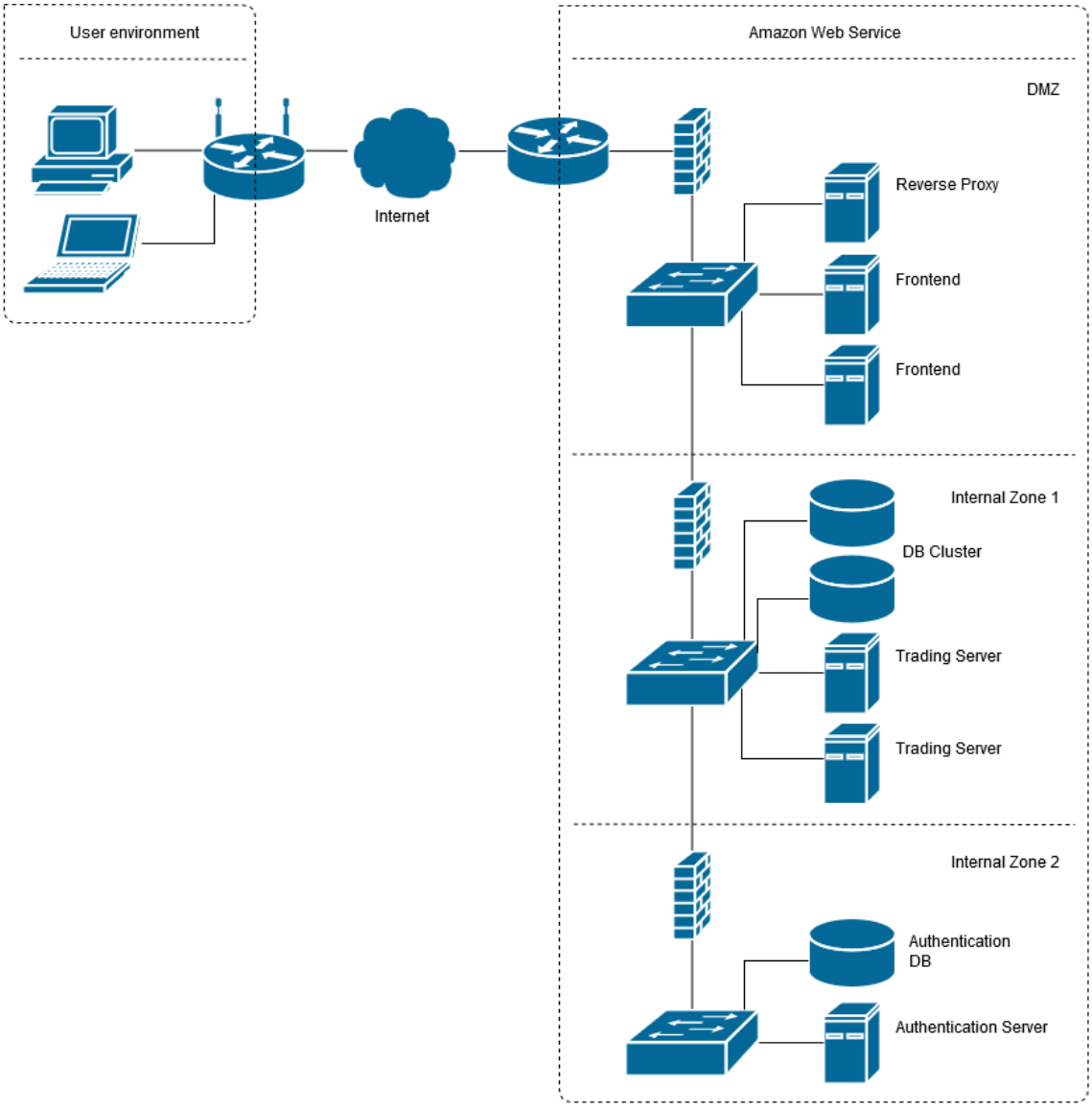

Figure 1.5: Network Diagram

Integration Endpoints

Source System	Destination System	Protocol	Format	Communication Mode
Frontend	Auth (/api/authenticate)	HTTPS (POST)	JSON	Synchronous
Frontend	Auth (/api/signup)	HTTPS (POST)	JSON	Synchronous
TradingServer	Auth (/api/validate)	HTTPS (POST)	JSON	Synchronous
Frontend	TradingServer	HTTPS	JSON	Asynchronous

	(/api/stock)	(GET)		
Frontend	TradingServer (/api/stock/{id})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/stockholding)	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/stockholding/account/{accountId}/stock/{stockId})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/stockholding/stock/{stockId}/account/{accountId})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/stockholding/account/{accountId})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/trade)	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/trade/{id})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/tradequeue)	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/tradequeue/{id})	HTTPS (GET)	JSON	Asynchronous
Frontend	TradingServer (/api/tradequeue)	HTTPS (POST)	JSON	Asynchronous
Frontend	TradingServer (/api/tradequeue/{id})	HTTPS (PUT)	JSON	Asynchronous
Frontend	TradingServer (/api/tradequeue/{id})	HTTPS (DELETE)	JSON	Asynchronous

Hardware/Software/Services required

No	Item	Quantity	License	Buy/Lease	Cost (Optional)
1	AWS EC2 Instance	7	Proprietary	Free-tier	NIL
2	AWS Code Deploy	1	Proprietary	Free-tier	NIL
3	Amazon S3 Bucket	1	Proprietary	Free-tier	NIL
4	Amazon RDS	1	Proprietary	Free-tier	NIL
5	Amazon Aurora	1 (2DB)	Proprietary	Free-tier	NIL
6	Travis CI	1	Open-sourced	NIL	NIL
7	Tomcat	1	Open-sourced	NIL	NIL
8	PostgresSQL	1	Open-sourced	NIL	NIL
9	MySQL	1	Open-sourced	NIL	NIL

* AWS credits used

Availability View

Node	Redundancy	Clustering			Replication (if applicable)			
		Node Config	Failure Detection	Failover	Repl. Type	Session State Storage	DB Repl. Config	Repl. Mode
Amazon Aurora DB Cluster	Horizontal / Vertical	Active-Active	Managed by Amazon	Managed by Amazon	DB	Database	Master-Slave	Asynchronous
Backend	Horizontal	Active-Active	Ping	Load-balancer				
Reverse Proxy	Horizontal	Active-Passive	Ping	Virtual IP				
Frontend	Horizontal	Active-Passive	Ping	Virtual IP				
Authentication Server	Horizontal	Active-Active	Ping	Load-balancer				

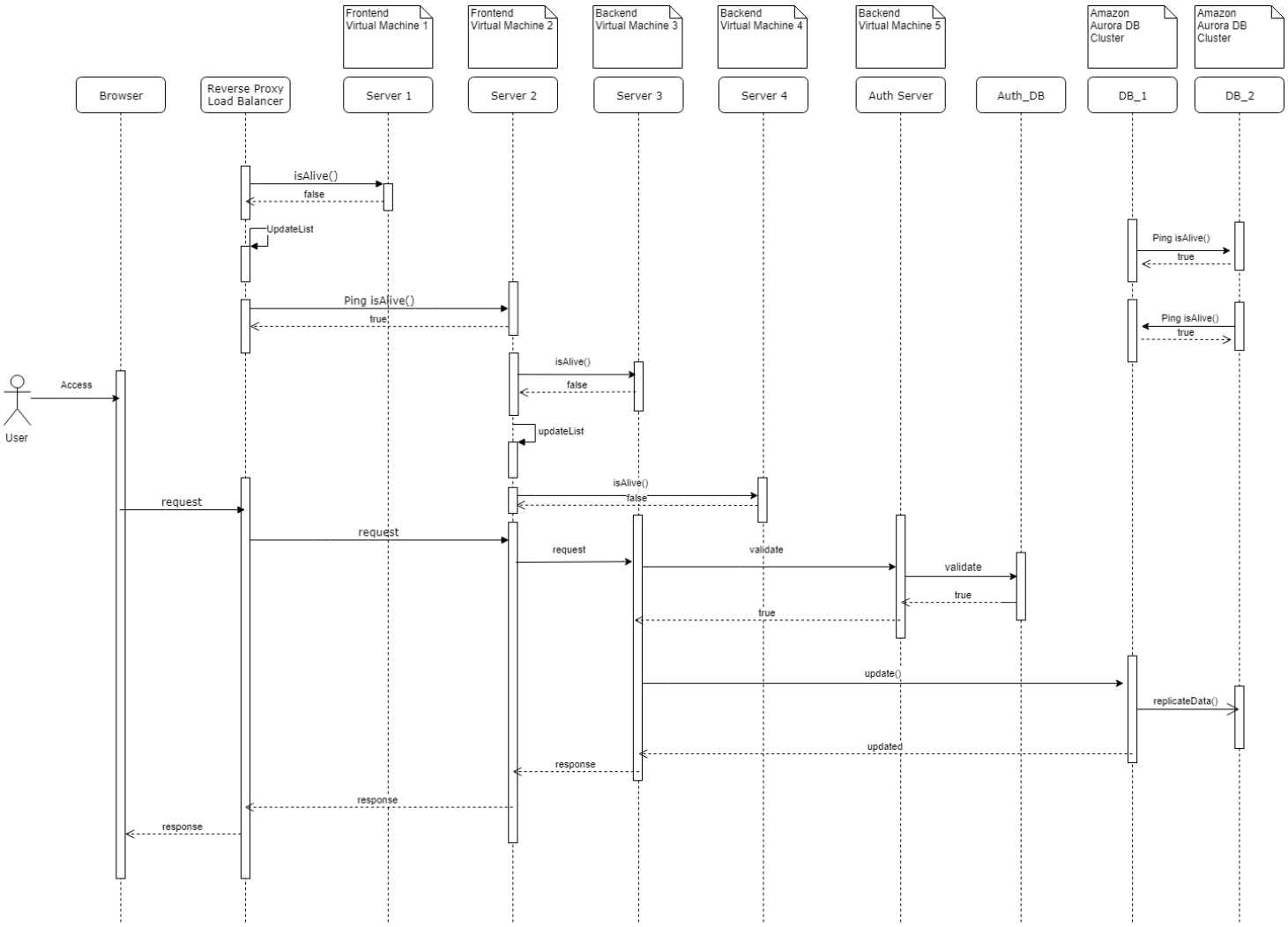

Figure 1.6: Sequence Diagram with Detect Failures and Failovers

Security View

No	Asset/Asset Group	Potential Threat/Vulnerability Pair	Possible Mitigation Controls
1	Server Access	Intrusion	Firewall, proper ssh key, Tunneling, Proper Access Control, usergroups, remove default root user access, editing visudo
2	Database Access	Running queries that leak data	Firewall, tunneling, proper user management, table access management
3	HTTP protocol leak	Leaking of user data	Not using HTTP -> HTTPS

4	Password Management	Leaking of password data	Hashing password, salting, stopping replay attacks via client side salt

Performance View

No	Description of the Strategy	Justification	Performance Testing (Optional)
1	Database Locking Optimisation	Application only requires row locking, taking a simplistic row locking attitude would result in a reduced parallelism of write statements	Improvement of average time for 200 concurrent purchase, sell stock request by ~300%.
2	Caching	Following SPA paradigm all frontend content are served static. Database select queries are constantly required to validate user accounts. Rather keep a memcached copy in the application and validate via there.	Frontend serve increase by ~50%. Database select queries almost no effect.
3	Removing a Frontend Server to static file server from nginx	Following SPA paradigm all frontend content are served static. No need for a spring boot application to serve it.	~200% increase from spring boot serve to nginx static.
4	Round-Robin Backend Server	Serving api read request should be doubled in processing request per second.	