Section 1: Overview

Each of the microservices implemented for this project utilized the Flask library for handling HTTP requests and routing. Each microservice therefore exposes multiple HTTP routes that other microservices and clients can utilize to perform actions such as trading stocks, looking up trades, looking up previous orders by their unique ID, as well as for performing caching, maintaining consistency across local databases, and leader selection in the event that one of the replica services crashes.

Section 2: Front End

In terms of the functional part of the application, the front-end microservice exposes three HTTP routes for handling the lookup of stocks and orders as well as the trading of stocks.

The route implemented for looking up stocks is the GET /stocks/<stockName> route. Whenever the client application requests to lookup a stock, it will issue a GET request using this route, and the front-end service will respond with a JSON file that contains a top-level "data" object that within itself contains the name, number of shares, and price of the specified stock. Similarly, the route implemented for looking up orders by their ID is the GET /orders/<orderNum> route, and in response to receiving a request on this route the front end will send a JSON file with a top-level "data" object that contains the order ID of the requested order as well as the name of the stock traded, the number of shares traded, and whether the stock was bought or sold.

The route implemented for trading stocks is the POST /orders route. This route requires that the client application attach a JSON file specifying the name of the stock to trade, the quantity of shares to trade, as well as whether the stock should be sold or bought. This route will return a JSON response with a top-level "data" object that simply contains the order number of the transaction in the event of a success, and in the event of a failure it will return a JSON response with a top level "error" object specifying the HTTP error code and a message pertaining to the reason for the error.

The front end microservice implements a cache that stores the information received from previous stock lookup requests. In the event a stock in the cache is traded, the current entry in the front end's cache needs to be invalidated and removed to prevent consistency issues with regards to the state of the stock. To this end, the front end exposes the POST /invalidate/<stockName> route to the catalog service in order to handle removing stocks that have been updated with new information, and the catalog service will issue requests to this route whenever a trade causes an update to the catalog's database.

Section 3: Catalog Service

For the functional part of the catalog service, it exposes the GET /lookup/<stockName> route to both the front end and order services. Whenever the catalog service receives a request on this route, it will look into its local database to retrieve information on the requested stock. In the event the requested stock is in the database, the catalog service will send a JSON response that contains the name, price, and current number of shares for the specified stock. In the event the stock does not exist in the database, this route returns a JSON response with a top-level “error” object that contains the fields “code” and “message” that specify the HTTP error code and the reason for the error respectively.

In addition, the catalog service also exposes the POST /update route to the order service. This route requires that all requests have an attached JSON object specifying the name of the stock, the number of shares being added or removed, and a field named “type” that specifies whether a stock is being bought or sold; for example if “type” = “buy”, the catalog service will attempt to decrement the number of shares of the stock in the database by the specified amount, and if “type” = “sell” the catalog service will attempt to increment the number of shares. In the event of a successful update, this route will return a JSON object with a top-level “success” object that contains the fields “code” and “message”: “code” is initialized to HTTP success code 200, and “message” is initialized to “successfully updated stock”. However, in the event a stock is not in the catalog database, this route will return a JSON object with a top-level “error” object that specifies the code and message to attach to the error.

Section 4: Order Service

For the functional part of the order service, it exposes the POST /buy and POST /sell routes to the front end service. Both routes require that the order service send with them a JSON object specifying the name of the stock to trade and the quantity of shares to trade; /buy and /sell each handle the case where a stock is bought and sold respectively. In the event of a successful trade, each route returns a JSON object specifying the order ID associated with the transaction. In the event of a failure, this route returns a JSON object specifying the HTTP error code as well as a message pertaining to the reason for the failure.

In addition, the order service also exposes the GET /orders/<orderNum> to the front end service. When the order service receives a request on this route, it will attempt to look within its database for the transaction specified by the orderNum parameter. In the event of a success, this route will return a JSON object specifying the order number, the name of the stock, the number of shares traded, and the type of transaction for the order. In the event the order service can not find the specified order or some other error occurs, a JSON object with a top-level “error” object will be returned specifying the HTTP error code and a message explaining the reason for the error.

To maintain consistency among the local databases for each replica, the order service implements the POST /push and POST /sync routes. The POST /push route is used by the lead order service replica to push updates such as the next ID to use for a new transaction as well as the information for a transaction that has recently occurred, and it requires that the lead order service attaches a JSON object specifying the ID of the transaction as well as information about the transaction such as the name of the stock being traded, the quantity being traded, as well as whether the stock was being bought or sold. Similarly, the POST /sync route is used by the order service replicas as each is starting or recovering from a crash. This route requires that the replica issuing a request attach the ID of the last known transaction in its database, and the order replicas will respond by sending both information about the current lead replica as well as the transactions the replica has missed when it was offline.

For leader selection, the order service exposes the GET /ping route to the front end service. When the front end service starts, it will issue a request to the /ping route of each order service in descending order of their server IDs. If an order service receives a request on this route, it will send a JSON response with a top level “success” object that contains the fields “code” and “message”: the “code” field is initialized to 200 to signify a success, and an appropriate success message is returned.

To further facilitate leader selection, the order service also exposes the POST /leader-broadcast route to the front end service. This route is specifically used by the front end to notify the other order replicas that a leader has been chosen whenever it receives a successful message from the GET /ping route from any of the order service replicas. When an order service receives a request on the POST /leader-broadcast route, it will update its local information about the order service leader, and it will send a JSON response that contains a top-level “success” with the fields “code” and “message” that represent the HTTP status code and a message from the replica acknowledging that it is aware of the presence of a new leader.