# Stock Application Design Document

## Part1 – Frontend Service

**Design choices:**
The server is designed using Flask, a lightweight and flexible Python web framework. Flask is chosen for its simplicity, ease of use, and flexibility, which make it ideal for building RESTful APIs. Flask uses multithreaded 'thread per request' model for handling concurrent requests.

**Server Components:**
- FrontendService()
  a) The class implements methods for communicating with the backend services and client.
  b) The constructor of this class creates a new Flask application instance and assigns it to the app instance variable of the class.It then calls the ***`setup***() function which extracts the address and IP of replica order servers . This function also calls the ***`find_leader()*** and ***`send_leader_info***() functions which find the leader among replicas and passes the leader information to all the replicas.
  c) The server class includes several custom routes to handle incoming http requests such as GET,POST,DELETE.

**Synchronization:**
- cache_lock: This lock is used by ***`invalidate_cache()*** and ***`lookupstock()*** functions before accessing the cache for removing data and adding data respectively.
- leaderlock: This lock is used during start up, by **`tradestock()** and **`checkstock()** functions to send information to the leader and also find the new leader and send leader information if the current leader doesn't respond.
- configlock : Used to store config data for different servers and read from it

**Datastructures/Variables:**
- servers : dictionary which stores id,port,address of all replica order servers at start up.
- catalog_cache : dictionary which stores the stock name and its details during a lookup.
- order_id : variable to store the id of the leader order server.
- orderserver_addr :variable to store leader's address
- orderserver_port:variable to store leader's port
- cache_en: variable set at startup to enable or disable caching feature.

**API Endpoints:**
- Endpoint: **/stocks/<stockname>**
  Description: Looks up for a stock
  Method: GET
  Request body: {}
  Response body:
  Success : {"data": {"name": string , "price" : float, "quantity": integer}}
  Error message : {"error": {"code": 404,"message": "stock not found"}}

- Endpoint**: /orders**
  Description: Trade a stock
  Method: POST
  Request body: {"name": string, quantity": integer, "type": string}
  Response body:
  Success : {"data":{"transaction_number":integer,"name":string,"quantity":integer,"type":string}}
  Error messages :

{"error": {"code": 404,"message": "stock not found"}}
{"error": {"code": 403,"message": "Not enough stock"}}

- Endpoint: **/invalidate/<stockname>**
  Description: Remove stock from local frontend cache
  Method: DELETE
  Request body: {}
  Response body:
  {'data': {'message': 'Successfully invalidated'}}
  {'data':{'message': 'Stock not present'}}
  {'data': {'message': 'Caching not enabled'}}

- Endpoint**: /orders/<ordernum>**
  Description: Check whether the transaction is present in order server
  Method: GET
  Request body: {}
  Response body:
  Success : {"data":{"transaction_number":integer,"name":string,"quantity":integer,"type":string}}
  Error messages :
  {"error": {"code": 409, "message": 'Order number does not exist'}}

**Interfaces:**

1) **GET: /stocks/<stockname>**
   The ***`lookupstock()*** function checks if ***`cache_en*** feature is set and accordingly takes a ***`cache_lock*** on the local cache, checks if the stock is already present. If the stock is present it returns the stock details as a json http resonse. If the feature is not set or the stock is not present in the local cache, the function sends a http GET request to the catalog server and stores the received output in the cache if **`cache_en** feature is set.
2) **POST: /orders**
   The ***`tradestock()*** function takes ***`leaderlock*** and sends a trade request to the leader order server. If the server does not respond or there is a timeout, a leader finding process is automatically initiated to check the availability of a replica server in decreasing order of their ids. The leader information is passed to all the replicas and the request is sent to the new leader.
3) **DELETE: /invalidate/<stockname>**
   The function ***`invalidate_cache()*** checks whether ***`cache_en*** feature is activated and applies a cache lock and checks if the stock name is present in the local cache. If it is present, it removes the details of the stock.
4) **GET: /orders/<ordernum>**
   The ***`check_stock()*** function takes ***`leaderlock***, sends a http get request for stock details corresponding to the transaction number sent to the frontend server.If the leader server does not respond or there is a timeout, a leader finding process is automatically initiated to check the availability of a server in decreasing order of their ids. The leader information is passed to all the replicas and the request is sent to the new leader.

## Part2 - Backend Catalog Server

### Design choices:

- The Catalog server is implemented using Flask web framework. The framework is multithreaded and uses thread per request model.

### Components:

- CatalogServer() :

  a) The constructor of this class creates a new Flask application instance and assigns it to the app instance variable of the class,calls the ***`load_data()*** function which initially loads data from stockCatalog.csv and stores it in a dictionary called ***`catalog***. This class is responsible for implementing all functions required to perform trade operations namely lookup and trade. This class also has a ***`backup_data*** function which is started as a separate thread when the object of this class is created. The function periodically backs up changes from the dictionary back to the csv.

  b) The constructor of the class also calls the `setup() method which contains the **`trade** and **`lookup** http route interfaces.

### Synchronization:

- cataloglock : This lock is used by both trade and lookup functions while updating and looking up for stocks in catalog dictionary.
- orderlock : This lock is used by backup_data() to take a lock on the stockCatalog.csv and periodically back data into it.

### Datastructures:

- catalog: dictionary which stores the stockname,quantity,volume traded and price.

### API Endpoints

- Endpoint: **/lookup/<stockname>**
  Description: Looks up for a stock Method: GET
  Request body: {}
  Response body:
  Success : {"data": {"name": stockname, "price": price, "quantity": quantity}}
  Error: {"error": {"code": 404, "message": "stock not found"}}

- Endpoint: **/update**
  Description: Updates the quantity and traded volume for the given stock
  MEHOD : POST
  Request body: {'name':stockname,'quantity':reduced_quantity,'traded':quantity}
  Response body:
  {"data": {"message": "Success"}}

### Interfaces:

1) **GET /lookup/<stockname>**
   - The ***`lookup()*** function takes a ***`cataloglock*** on the ***`catalog*** dictionary , looks up for the stock and sends a dictionary with values if the stock exists or sends an error message. The body is converted into json format and sent back as a http response.

2) **POST /update**
   - The **`*trade()*** function takes a lock on the catalog dictionary, updates the quantity and traded volume of the stock, sends an invalidation http DELETE request to the frontend server and returns a 'success' message as a dictionary. The body is converted into json format and sent back as a http response.

## Part3-Backend Order Server

### Design choices:

The Order Server is implemented using the Flask web framework. Three replicas of the Order Server are created, each with its own copy of the database file and a unique ID associated with it. The ID is appended to the database file and used in all the routes defined within that server.

### Components:

- orderserver():
  a) The class has methods for communicating with the frontendserver and catalogserver.
  b) The constructor of the class initially sets the **`*count*** variable to 0, **`*active*** variable to false,creates a new Flask application instance and assigns it to the app instance variable of the class. It then calls the **`*setup()*** function.
  c) The **`*setup()*** function initializes the server and ensures that it is in a consistent state by performing two critical tasks. Firstly, it calls the **`*get_replica_configs()*** function to retrieve the addresses, ports, and IDs of all the order replicas. Using this information, it determines which replica is currently serving as the leader, updates the leader information using **`*update_leader()*** and proceeds to synchronize the server's database with the leader's database using the **`*synchronize_database()*** function.
  d) The server class includes several custom routes to handle incoming http requests such as GET,POST.

### Synchronization:

- transactionlock**:** The **`*count*** variable is locked using the transactionlock before incrementing it after a successful trade. The **transactionlock** is also used to assign the value of **`*count*** to the last transaction number as sent by the leader during trade synchronization. This ensures that the server's **`*count*** value is consistent with the leader's value, and no transactions are lost or duplicated during the synchronization process.
- writelock: The writelock is taken for reading and writing to ordercatalog.csv file. The write lock is taken and the transaction number,stock name,quantity and type of the stock are appended to the csv file. The lock is also taken during synchronization while writing the updated data and during reading the data for sending synchronizing information.
- leaderlock: The leaderlock is taken whenever the leader contents need to be updated and whenever requests are sent to the leader order server.
- activelock: The activelock is taken for updating or accessing the **`*active*** variable.
- configlock:Used to read configurations of all the servers.

### Datastructures/Variables:

- count: variable is used to keep track of the transaction number
- active: once this variable is set to true ,the replica is considered to be fully synchronized and can safely start accepting transactions from the leader without worrying about duplicates or other issues.

- orderservers: dictionary is used to store the addr,ip and id of all the order server replicas
- leader_addr: variable stores the address of the leader server
- leader_port: variable stores the leader port
- leader: variable stores id of the leader.

**API Endpoints**

- **Endpoint: /find_leader_{id}**
  Description: Returns the ip,port and id of itself if it is active
  Method: GET
  Request body: {}
  Response body:
  {'data':{'leader_id':self.leader,'addr':self.leader_addr,'port':self.leader_port}}
  Error: {'error':{'code':400,'message':'Serve not up'}},400

- **Endpoint: /databasesync_{id}**
  Description: Sends complete synchronization data to replica which is just up
  Method:POST
  Request body: {{'transaction_number':integer}}
  Response body:
  [{'transaction_number':integer,'name':string,'quantity':integre,'type':string},{},..]
  {'data':{'message':'Nothing to sync'}}

- **Endpoint: /leader_info_{id}**
  Description: Updates the leader id,port and address
  Method: POST
  Request body: {'data': {'leader': self.order, 'addr': self.orderserver_addr,
  'port':self.orderserver_port}
  Response body: {'data':{'message':'Success'}}

- **Endpoint /synchronise_{id}**
  Description: Used by replica to add the trade response sent by the leader to its database
  Method: POST
  Request body:
  {"data":{"transaction_number":count,"name":stockname,"quantity":quantity,"type":type}}
  Response body:{'data':{'message':'Success'}}

- **Endpoint /ping_{id}**
  Description: Used by frontend to check if the server is active
  Methd: GET
  Request body:{}
  Response body:
  Success:{'data':{'message':'active'}}
  Error:{'error':{'code':400,'message':'not active'}}

- **Endpoint /checkorder_{id}/<ordernum>**
  Description:Used to check if the order details with a particular transaction number exist in the order server
  Methd: GET
  Request body:{}
  Response body:
  Success: {"data":{"transaction_number":integer,"name":string,"quantity":integer,"type":string}}
  Error:
  {'error',{'code':400,'mssage':'Server not up yet'}},400
  {"error": {"code": 409, "message": 'Order number does not exist'}}

- **Endpoint /trade_{id}**
  Description: Leader sends a lookup request and then an update request to the catalog server and also sends the trade response to the other replicas for synchronization.
  Method: POST
  Request body: {"name": stockname, quantity": quantity, "type": buy/sell}
  Response body:
  Success: {"data":{"transaction_number":integer,"name":string,"quantity":integer,"type":string}}
  Error :
  {"error": {"code": 403, "message": 'Not enough stock'}}
  {"error": {"code": 405, "message": "Unknown error,retry"}}
  {'error',{'code':400,'message':'Server not up yet'}},400

**Interfaces:**

**a) GET /find_leader_{id}**

- The ***`find_leader()*** function takes an ***`activelock***, checks if the replica is active. If it is active, it takes the ***`leaderlock*** and sends its id, port and address as a json http response. If it is not active it sends an error message with error code 400.

**b) POST /databasesync_{id}**

- The ***`databasesync()*** function takes an **`activelock** and checks if the server is active.If the server is not active it returns an error message.
- If it is active, it extracts the transaction_number from the request, takes a ***`writelock*** on orderCatalog, appends all the rows with transaction_number greater than the one sent by the request and sends the rows as a json object.

**c) POST /leader_info_{id}**

- The ***`leader_info()*** function takes a ***`leaderlock*** and updates the address,port and id of the leader.

**d) POST /synchronise_{id}**

- The ***`synchronise()*** function takes an ***`activelock*** and checks if the server is active.If it is active, it extracts information from various fileds in the trade response, takes a ***`transactionlock***, updates ***`count*** to the transaction_number sent by the leader.
- The function then takes a ***`writelock*** on the ordercatalog, writes the response to the file and returns a json success message.

**e) GET /ping_{id}**

- The function ***`ping()*** takes an ***`activelock*** ,checks if server is active.If it is active, it sends a json object with message 'active'.If the server is not active,it sends a json error response that the server is not active.

**f) GET /checkorder_{id}/<ordernum>**

- The checkorder() function takes an ***`activelock*** ,checks if the server is active.
- If the server is active, it calls ***`read_file(ordernum)*** which reads ordercatalog.csv and returns the row with **<ordernum>.**
- The server sends a json object as response depending on whether the **<ordernum >**is present or not.

**g) POST /trade_{id}**

- The ***`trade()*** function takes an ***`activelock*** and checks if the server is active or not.
- If the server is active,the function sends a http GET /lookup/<stockname> request to the catalogserver.
- If the trade is a buy request, it checks the quantity returned and sends a POST /update request to the catalogServer with a json body that contains the stockname,actual quantity to update and the traded volume only if the returned quanity is greater than 0.If the trade is a sell request it directly sends it.

- If the transaction is successful, it takes a `***transactionlock***, increments the `***count*** variable, takes the `***writelock***, updates ordercatalog.csv.
- It creates a json reply with transaction_number,name,quantity and type, sends it to all the replicas for synchronisation and returns the reply as a dictionary.

## Client

### Design choices:

This client will be designed using Python and the requests library, and will support HTTP 1.1. The value of p is initially assigned to 0.6 for sending order requests.

### Client components:

Connect()-

- The function runs a loop 1000 times. Each time it picks a random stock from the list of stocks present and sends a http compliant GET request using requests library to the frontend service.
- If the http response has an error, there is no trade request sent.
- If the lookup was successful, a trade request body is generated using create_trade_request for the given stock and a http post request is sent to the frontend service with a probability 'p'. If the received response doesn't have an error, the response is stored in an in-memory list.
- In the end, for each value in the list , the transaction number is extracted, a post request is sent to the orderserver with the transaction number. The details received is compared with the result already cached. If there is an error it is printed out.

create_trade_request()

- The function generates trade dictionary by picking random value for quantity and choosing one of 'buy' or 'sell'.

### Datastructures/Variables:

- trade_result_cache **:** The list is used to store the trade response received from the order server.