**Design Document: Two-Tiered Stock Bazaar Microservices**

**Overview:** The Two-Tiered Stock Bazaar consists of three microservices: front-end service, catalog service, and order service. The front-end service communicates with clients using HTTP socket. The catalog and order services are accessed by the front-end service, with the order service interacting with the catalog service to complete an order request. Each service is implemented as a server that listens to requests using sockets. The catalog and order services persist data to disk using a CSV file. Clients interact with the front-end service using HTTP sockets.

**Front-end Service:** The front-end service consists of a front-end service program and an HTTP service handler. The front-end service receives HTTP GET and POST requests on a socket port, parses the requests, and processes them using a thread pool. The service supports two REST APIs: Lookup and Trade. The Lookup API retrieves stock details. If the lookup request is successful, the server returns a JSON response with a data object containing the stock name, price, and quantity. If an error occurs, the front-end service returns a JSON response with an error object containing an error code and message.

The Trade API places an order for a stock by attaching a JSON body to the POST request. If the order is placed successfully, the front-end service returns a JSON object with a top-level data object containing a transaction number. If an error occurs, the front-end service returns a JSON response with a top-level error object containing an error code and message.

Catalog Service:

The catalogService includes a static stocksCatalog ConcurrentHashMap that stores the mapping of stock names to Stock objects, and a static tradeCatalog ConcurrentMap that stores the mapping of trade numbers to Trade objects. It also has a static tradeNumber integer that represents the next trade number.The catalog service maintains a list of all stocks traded in the stock market. It also maintains the trading volume of each stock and the number of stocks available for sale. When the front-end service receives a Lookup request, it forwards the request to the catalog service. The catalog service reads the catalog data from a CSV file on disk and returns it to the front-end service. When the order service receives a Trade request, it contacts the catalog service to update the trading volume and the number of stocks available for sale. These updates are written to the CSV file on disk. The catalog service is implemented using sockets and a thread pool to handle concurrent requests. The service uses stamp locks to synchronize access to the catalog data to prevent concurrent modifications.

**Order Service:** The order service consists of an order service and a poster service program. The order service receives Trade requests from the front-end service. It interacts with the catalog service to complete the order. If the order is successful, the order service generates a unique transaction number and returns it to the front-end service. The order service maintains an order log containing the transaction number, stock name, order type, and quantity in a CSV file on disk. The order service is implemented using sockets and a thread pool to handle concurrent requests. The service uses synchronization to prevent concurrent modifications to the order log.

The code implements an order service by creating a ServerSocket and listening for incoming client connections on a specified port. It uses ExecutorService and the PosterProcessor class to handle client connections and requests. Specifically, it creates an ExecutorService object to handle client connections, and in an infinite loop, uses the ServerSocket to accept client connections and submit a PosterProcessor task to handle the request. The task passes the received client socket to the constructor of the HttpServiceProcessor class and calls the run() method of the HttpServiceProcessor class to handle the request at runtime. At the end of the main() method, the try-with-resources statement ensures that the ServerSocket is properly closed when the program ends.

**Interfaces**: The catalog service exposes an internal interface to the front-end and order services. The interface includes functions to read and write catalog data to a CSV file. The order service exposes an internal interface to the front-end service. The interface includes functions to complete an order request and write order log data to a CSV file on disk.

**Communication:** The front-end service communicates with clients using HTTP-based REST APIs. The catalog and order services are accessed by the front-end service using sockets.

**Concurrency:** Each microservice employs sockets and thread pools to handle concurrent requests. The catalog and order services use read-write locks and synchronization respectively to prevent concurrent modification of catalog data and order logs. The thread pool size is set sufficiently large to handle all active clients and their sessions without starving.

**Conclusion:** The two-tier stock marketplace microservice is designed to be concurrent, efficient, and scalable. The frontend service employs a thread pool to handle HTTP-based REST APIs. The catalog and order services persist data to CSV files on disk. The entire system has been carefully designed with regard to communication, concurrency, and interface design among various services to achieve a high-performance, scalable stock trading service.