Student Name: Siqi Li

GitHub Repo Under NEU Organization:

<https://github.ccs.neu.edu/lisiqi/CS6650-DSBS-Repo>

Explanation About Client Structure and How it works:

* *Database Design:*
* One table names “Puchases” to store all purchase request as each row input
* Table contains:
* UUID as primary key, and avoid data duplication
* StoreID as Integer
* CustomerID as Integer
* Data as String -> with the given date send by client2 purchase URL
* CreateTime as Timestamp, takes the server system time while insert to DB
* PurchaseBody as Json that contains all purchase items information
* *Total Three Packages @ “SuperMarketServer”:*

*MVC pattern used*

* model: (general utils for Server)
  + PurchaseRecord:
    - the data access object for each purchase request received
    - Contains attributes match the database purchases table for each columns
* dal: (the DB connection)
  + DBCPDataSource:
    - Use HikariCP as the Database Connection Pool configuration
    - set a max pool size of 15
    - set max possible lifetime for connection in pool for 20 mins
  + PurchaseDao:
    - Setup and close Database Connection Pool (DBCP) connections
    - Insert data to purchase table with executing SQL queries
    - Enforce successful insertion, when insertion fail, keep inserting until successful, avoid failed insertion that give a failed http response back to client and cause a resend of duplicate purchase call
* tools: (the purchase servlet)
  + PurchaseServlet: the server that handles purchase requests
    - have a DBCPDatasource pool attribute for DB connections
    - valid http request if contains required URL path as header information
    - valid http request body is not empty
    - Get information from the URL path and request body to insert to DB, and get DBCP (Database Connection Pool) connections for each post request and close DBCP connection after done insertion
    - write captured information to response body, and send back
* *Total two Packages @ “SuperMarketClient”:*
* model: (general utils for both client2)
  + Record: the record object used to store all latency results for Client2
  + LogStderr: the error logger for write out system error to err.txt file
  + HttpMethod: Enum for POST and GET
  + CmdParser: set up required and optional command line input options to run client2
* part2: (package for client 2 to generate requested outputs)
  + Clinet2:
    - Main for execute all threads by calling SingleThread in package part2
    - Check if command line arguments are valid
    - Generate output results for part 2, print out to console
  + SingleThread:
    - the single runnable object represents as one thread to make purchase requests with remote server
    - when send purchase request failed, retry maximum 3 times to make purchase request send successfully
    - Use AtomicInteger to count the total successful/unsuccessful requests
    - Use CountDownLatch to initiate phase 2 and phase 3 threads
    - designed for Client 2 to capture start and end time of each request, in order to get latency time
    - use LogStderr to log errors into txt file
  + RecordWriter:
    - Runnable object used to generate Records object and write to csv file
    - Using BlockingQueue, and work as a Consumer, when consider each SingleThread are all producers of the queue
  + ReportGenerator:
    - Generate output results for part 2, print out to console
    - Read in all Records stored in CSV and get all Latency in to a list and sort list
    - Analysis all latency list, and get mean, median, max and p99 response time

Below are the output results for Client2 and analysis Charts:

* Results based on run Server on EC2, and run client on EC2 and database as RDS
* *With 32 Threads Single Server:*

Text

Description automatically generated

* *With 32 Threads Load Balanced Server:*

Text

Description automatically generated

* *With 64 Threads Single Server:*

Text

Description automatically generated

* *With 64 Threads Load Balanced Server:*

Text

Description automatically generated

* *With 128 Threads Single Server:*

Text

Description automatically generated

* *With 128 Threads Load Balanced Server:*

Text

Description automatically generated

* *With 256 Threads Single Server:*

Text

Description automatically generated

* *With 256 Threads Load Balanced Server:*

Text

Description automatically generated

* *With 512 Threads Load Balanced Server:*

Text

Description automatically generated with medium confidence

* *Analysis Charts:*