

Barista – customer - app

CE303 Advanced Programming



December 2, 2022

prince eKEWENU

2004921

Contents

[Introduction 1](#_Toc120109067)

[Customer (Client) Implementation 2](#_Toc120109068)

[Barista (Server) Implantation 2](#_Toc120109069)

[Logging 3](#_Toc120109070)

[Project review and personal reflection 3](#_Toc120109071)

[Appendix 4](#_Toc120109072)

[How to compile and run the Barista 4](#_Toc120109073)

[Compiling the Barista prgram 4](#_Toc120109074)

[Running the Barista application 4](#_Toc120109075)

[How to compile and run the Customer 4](#_Toc120109076)

[Compiling the Customer program 4](#_Toc120109077)

[Running the Customer application 4](#_Toc120109078)

[Accepted Barista command line input 4](#_Toc120109079)

[Accepted Barista command line from Customer 5](#_Toc120109080)

# 

# Introduction

This report is aimed to deliver the complete description and functionality of The ‘Virtual Café’.

This system aims to provide a solution that assist in a Barista (Server) servicing multiple Customers (Clients).The purpose of the Barista will be to manage multiple orders from different Customers whilst minimising waste by repurposing abandoned orders to new orders if the requirements meet. The purpose of the Customer is to make orders to the Barista. At any point in time the Customer can make a request to the Barista to find out their order status. The barista is responsible for returning the correct order status to the customer as required. Throughout this report we will identify the Server as Barista and the Client as Customer.

# Customer (Client) Implementation

The implementation of the Customer is quite simple, and its responsibility is to listen to messages from the Barista and send commands of the server when needed. The Customer was implemented using 2 threads. The purpose of the two threads are as follows:

1. ***Main Thread*** – This is the main application thread that runs up when the main application starts. Its purpose is to read input from the command line and send to the Barista. Reading of server input is in an infinite loop hence it was necessary to create a new thread to handle incoming messages from the server.
2. ***CustomerRunnable* *Thread*** – This is a custom class that implements Runnable. It spins up and runs asynchronous to the main thread when the Customer is connected to the Barista. Its purpose it to read all incoming communications from the Barista. This was necessary as reading incoming messages from the server is blocking and will cause the main thread to wait forever until it receives a message from the server.

# Barista (Server) Implantation

The Barista implementation wasn’t as easy as the Customer implementation. The Barista’s purpose it to accept new Customer connection. To achieve this behaviour, the Barista spins up a new Thread to handle each Customer (Client connection). In addition to this Barista spins up other threads internally to handle processing of customer orders. The Barista implementation works as follows:

1. When the application starts, it cleans the ***Server-log.json*** file by writing an empty array into it.
2. We start listening to incoming communications calling ***BaristaProcessor.getInstance().startServer().*** The Barista processor is a singleton object to ensure only one instance of the server will run throughout the lifecycle of the Barista Application.
3. Once the ***BaristaProcessor.getInstance().startServer()*** method is invoked, the processor spins up a separate thread for listening to any command send to the server from the command line (Query the server to tell the user how many orders are in a room at any given time). See [appendix](#_Accepted_Barista_command) for accepted server commands or type ***help*** in the barista’s command line. The main thread listens to any incoming connection (this is a blocking operation). The Barista implements two interfaces namely **CustomerRequestHandler** and**CustomerOrderProcessorNotification.**

The**CustomerRequestHandler**interface has 3 methods name **onRegisterCustomer, onMessageReceived**and**onDisconnect.**

* The ***onRegisterCustomer*** get called when the user has entered their name, so the Processor can keep hold and identify the Client (The processor uses the Thread’s ID to identify the customer and their name to address them. It used the thread’s id so It can accept multiple clients with the same name).
* The***onMessageReceived***gets called each time acustomersends a message. The method is called with the Customer’s Id, Customer’s name, and the message for the processor to identify the correct Customer and process their message accordingly.
* The ***onDisconnect***gets called with the Customer’s id whenever it disconnects so the processor can dispose of it and repurpose their other for another customer if there is any.

The **CustomerOrderProcessorNotification**interface has two methods***notifyCompletion***which notifies the processor when an order is complete, ***notifyStatusChange***which notified the processor when state of a customer’s order has change. The processor then sendsthese messages to the Customer.

The ***BaristaProcessor*** also has **CustomerOrderProcessor**whose responsibility is to manage items in the **Waiting**, **Brewing** and **Tray** areas. These areas objects of their own and uses a synchronized list to manage their Orders (Beverages). All methodsin these areas aresynchronized to prevent race conditions.

* The Waiting area can hold n number of individual teas and coffees orders. It only gives out a maximum of 2 teas or coffees orders when queried for next order to process.
* The Brewing area will only process a maximum of 2 teas or 2 coffees in any given time. It Uses **Executors**from the *java.util.concurrent*to create a thread pool of 2 which executes the **BeveragePreparer**which prepares the Beverages.
* The Tray area is where all orders are delivered. When a beverage is placed in this tray it checks the customers main orders to see if all of their orders are I the tray. If so the it notifies the ***BaristaProcessor*** to deliver the order to the Customer.

1. When the processor establishes a new connection, it creates a new Thread called **CustomerConnection.** The ***CustomerConnection*** holds reference to the socket created and the Barista Processor. It uses the socket connection to send messages to the and notifies the processor through the reference its holds. In its run methods it spins up another Thread to make it non-blocking. The thread is used for listening to incoming communication from the Customer.
2. Once a connection has been established and the Customer has provided their name the ***BaristaProcessor***puts the connection in a synchronized map(which is thread-safe) with the thread id as it’s key and the connection as its value. The ***BaristaProcessor***later uses the thread’s id to get the connection and send messages to the Customer.
3. If the ***BaristaProcessor***receives an on disconnect signal, it then asks the **CustomerOrderProcessor** to perform repurpose or clear all the customers orders if there isn’t any order in the Brewing and Tray area.

# Logging

The server has a dedicated class **ServerLogger**to logs all its outputs. This is to ensure logging is done in one place so we can log on console and Server-log.json file.

# Project review and personal reflection

I have thoroughly enjoyed doing this project. I have learnt a lot about TCP, Socket programming, Data Structures and Multi-threading.

The difficult part for me was serving multiple clients at the same time whilst also processing other request. There were several times where my application run into a deadlock when I had multiple clients ordering the same items. With a lot of debugging and learning from stack overflow I was able to identify the deadlock and synchronize the methods causing these issues.

I also feel I could have used Queues in Waiting, Brewing and Tray area but I wasn’t able to figure out how to replace items in a specific index when repurposing the orders.

Another thing I wish I could have done is kill the thread that is brewing a beverage when a customer has abounded the order and there is no order to repurpose their to. Currently the beverage will finish brewing then discarded beverage which is waste.

# Appendix

## How to compile and run the Barista

Open command line and navigate to the **server** folder in the barista-customer folder.

### Compiling the Barista prgram

Type ***java -cp“gson.jar;.” Barista.java***on windows or ***java -cp“gson.jar:.” Barista.java*** on mac or a Linux machine

### Running the Barista application

After that type ***java -cp“gson.jar;.” Barista***on windows or ***java -cp“gson.jar:.” Barista*** on mac or a Linux machine

## How to compile and run the Customer

Open command line and navigate to the **client** folder in the barista-customer folder.

### Compiling the Customer program

Type ***java -cp“.” Customer.java***on windows or ***java -cp“.” Customer.java*** on mac or a Linux machine

### Running the Customer application

After that type***java Customer*** on windows or ***java -cpCustomer*** on mac or a Linux machine

## Accepted Barista command line input

|  |  |  |
| --- | --- | --- |
| Command | Description | Output |
| tray area info | Tells you how many beverages are in the tray area |  |
| waiting area info | Tells you how many beverages are in the waiting area |  |
| Brewing area info | Tells you how many beverages are in the brewing area |  |
| Anything else | Anything the server does not understand |  |

## Accepted Barista command line from Customer

|  |  |  |
| --- | --- | --- |
| Command | Description | Outcome |
| Order status | Tells the customer the status of their order | Tells the customer the status of their order |
| Order n coffees and n teas | Order to be processed | Process order for customer |
| Any invalid command | Any invalid command received from the Customer | Tell the customer it does not understand the command |
| Exit | Abandon’s Customer order and tries to repurpose it if I can | Discard Customer order and repurpose the order. |