UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Informatics Institute of Technology

Department of Computing

5COSC019C.1 Object Oriented Programming

Individual Coursework

Documentation & Testing Report

Real-Time Event Ticketing System with Multithreading & Concurrency Handling

Module Leader - Mr. Guhanathan Poravi

Student Name	IIT ID	UoW ID
Saajid Ahamed	20221921	w2052929

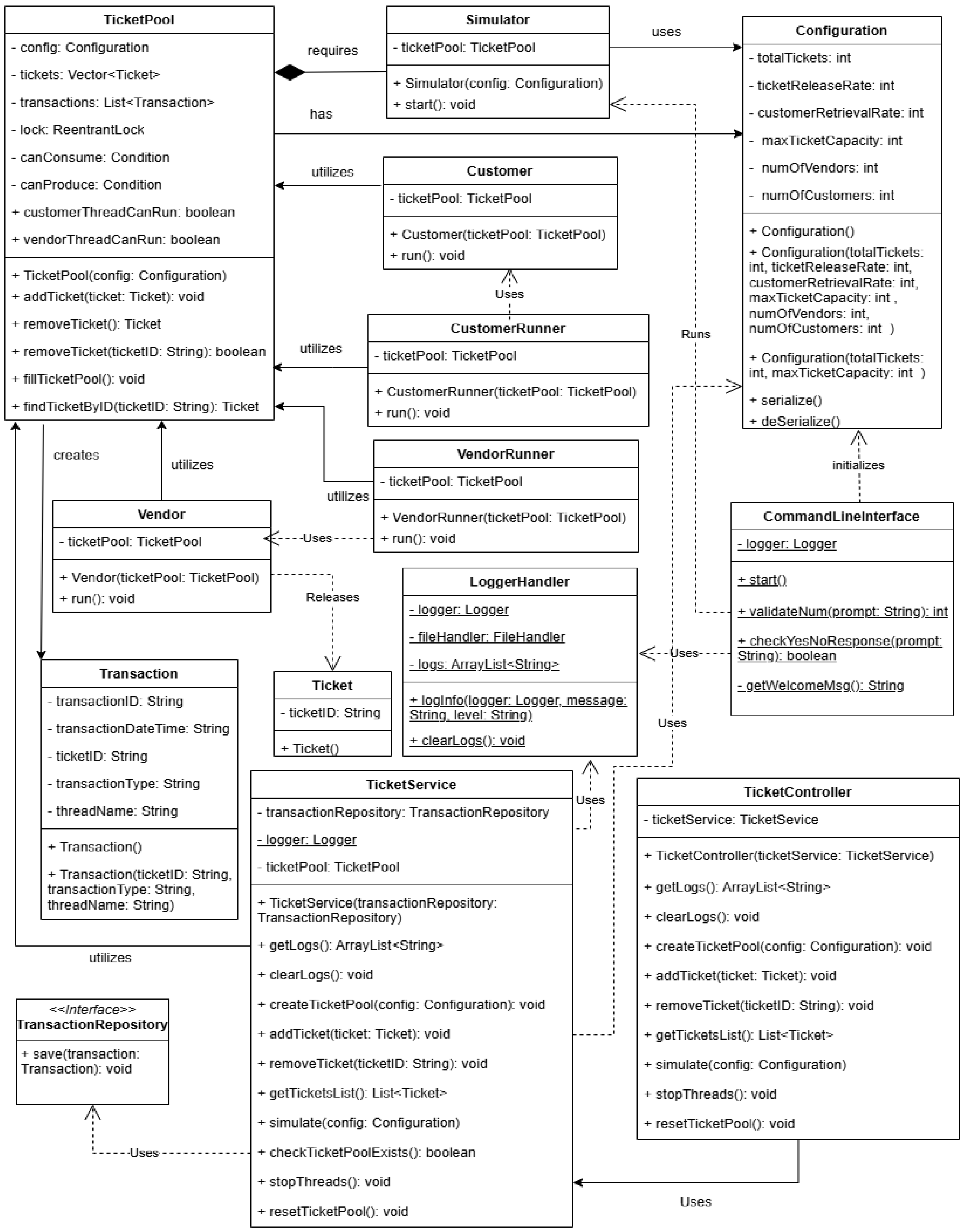
Table of Contents

List of Figures

List of Tables

FIGURE 1 CLASS DIAGRAM

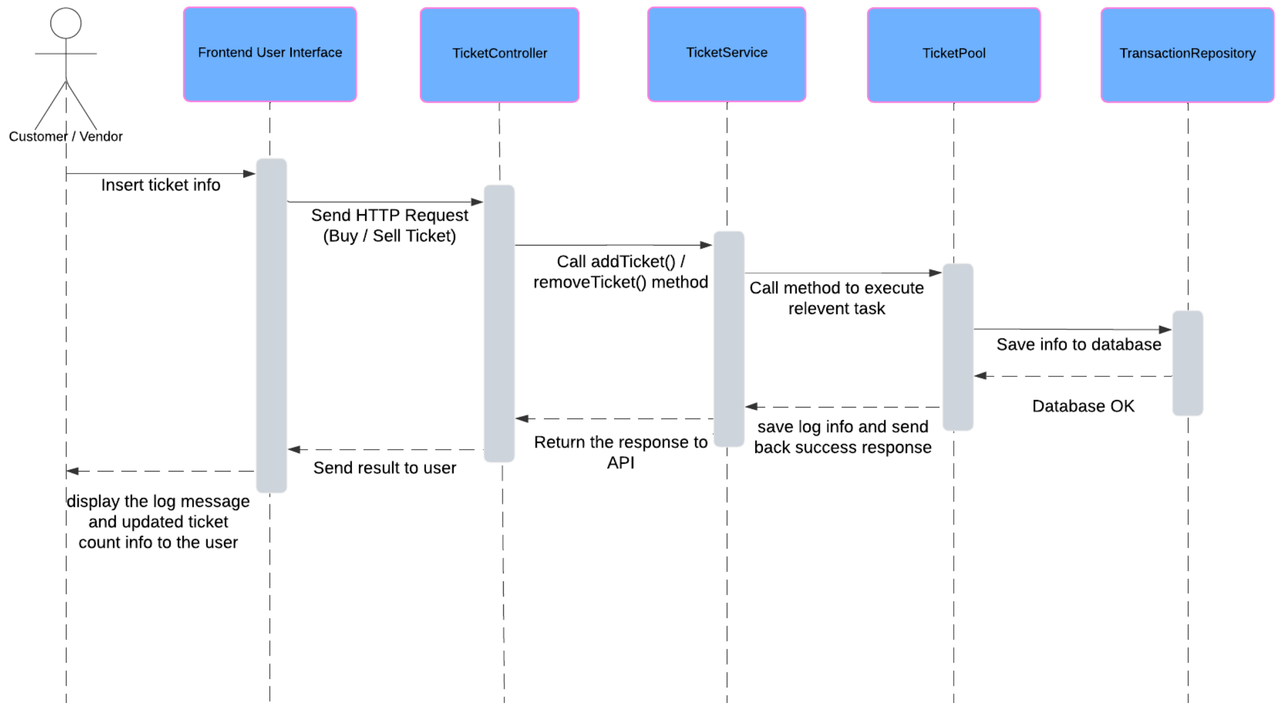
1. Class Diagram


TicketPool
- config: Configuration
- tickets: Vector<Ticket>
- transactions: List<Transaction>
- lock: ReentrantLock
- canConsume: Condition
- canProduce: Condition
+ customerThreadCanRun: boolean
+ vendorThreadCanRun: boolean
+ TicketPool(config: Configuration)
+ addTicket(ticket: Ticket): void
+ removeTicket(): Ticket
+ removeTicket(ticketID: String): boolean
+ fillTicketPool(): void
+ findTicketByID(ticketID: String): Ticket
requires
has
utilizes
Simulator
- ticketPool: TicketPool
+ Simulator(config: Configuration)
+ start(): void
uses
Configuration
- totalTickets: int
- ticketReleaseRate: int
- customerRetrievalRate: int
- maxTicketCapacity: int
- numOfVendors: int
- numOfCustomers: int
+ Configuration()
+ Configuration(totalTickets: int, ticketReleaseRate: int, customerRetrievalRate: int, maxTicketCapacity: int , numOfVendors: int, numOfCustomers: int)
+ Configuration(totalTickets: int, maxTicketCapacity: int)
+ serialize()
+ deSerialize()
Customer
- ticketPool: TicketPool
+ Customer(ticketPool: TicketPool)
+ run(): void
Uses
CustomerRunner
- ticketPool: TicketPool
+ CustomerRunner(ticketPool: TicketPool)
+ run(): void
Runs
VendorRunner
- ticketPool: TicketPool
+ VendorRunner(ticketPool: TicketPool)
+ run(): void
creates
utilizes
Vendor
- ticketPool: TicketPool
+ Vendor(ticketPool: TicketPool)
+ run(): void
Releases
initializes
CommandLineInterface
- logger: Logger
+ start()
+ validateNum(prompt: String): int
+ checkYesNoResponse(prompt: String): boolean
- getWelcomeMsg(): String
LoggerHandler
- logger: Logger
- fileHandler: FileHandler
- logs: ArrayList<String>
+ logInfo(logger: Logger, message: String, level: String)
+ clearLogs(): void
Transaction
- transactionID: String
- transactionDateTime: String
- ticketID: String
- transactionType: String
- threadName: String
+ Transaction()
+ Transaction(ticketID: String, transactionType: String, threadName: String)
Ticket
- ticketID: String
+ Ticket()
TicketService
- transactionRepository: TransactionRepository
- logger: Logger
- ticketPool: TicketPool
+ TicketService(transactionRepository: TransactionRepository)
+ getLogs(): ArrayList<String>
+ clearLogs(): void
+ createTicketPool(config: Configuration): void
+ addTicket(ticket: Ticket): void
+ removeTicket(ticketID: String): void
+ getTicketsList(): List<Ticket>
+ simulate(config: Configuration)
+ checkTicketPoolExists(): boolean
+ stopThreads(): void
+ resetTicketPool(): void
TicketController
- ticketService: TicketSevice
+ TicketController(ticketService: TicketService)
+ getLogs(): ArrayList<String>
+ clearLogs(): void
+ createTicketPool(config: Configuration): void
+ addTicket(ticket: Ticket): void
+ removeTicket(ticketID: String): void
+ getTicketsList(): List<Ticket>
+ simulate(config: Configuration)
+ stopThreads(): void
+ resetTicketPool(): void
<<Interface>>
TransactionRepository
+ save(transaction: Transaction): void
Uses
utilizes

2. Sequence Diagram

- The below sequence diagram illustrates the simple interaction for Vendor releases tickets to TicketPool and Customer retrieves tickets from TicketPool use cases.
- Only the essential classes are included to avoid any unnecessary complexities in the diagram.

FIGURE 2 SEQUENCE DIAGRAM


Customer / Vendor
Frontend User Interface
TicketController
TicketService
TicketPool
TransactionRepository
Insert ticket info
Send HTTP Request (Buy / Sell Ticket)
Call addTicket() / removeTicket() method
Call method to execute relevent task
Save info to database
Database OK
save log info and send back success response
Return the response to API
Send result to user
display the log message and updated ticket count info to the user

3. Testing Report

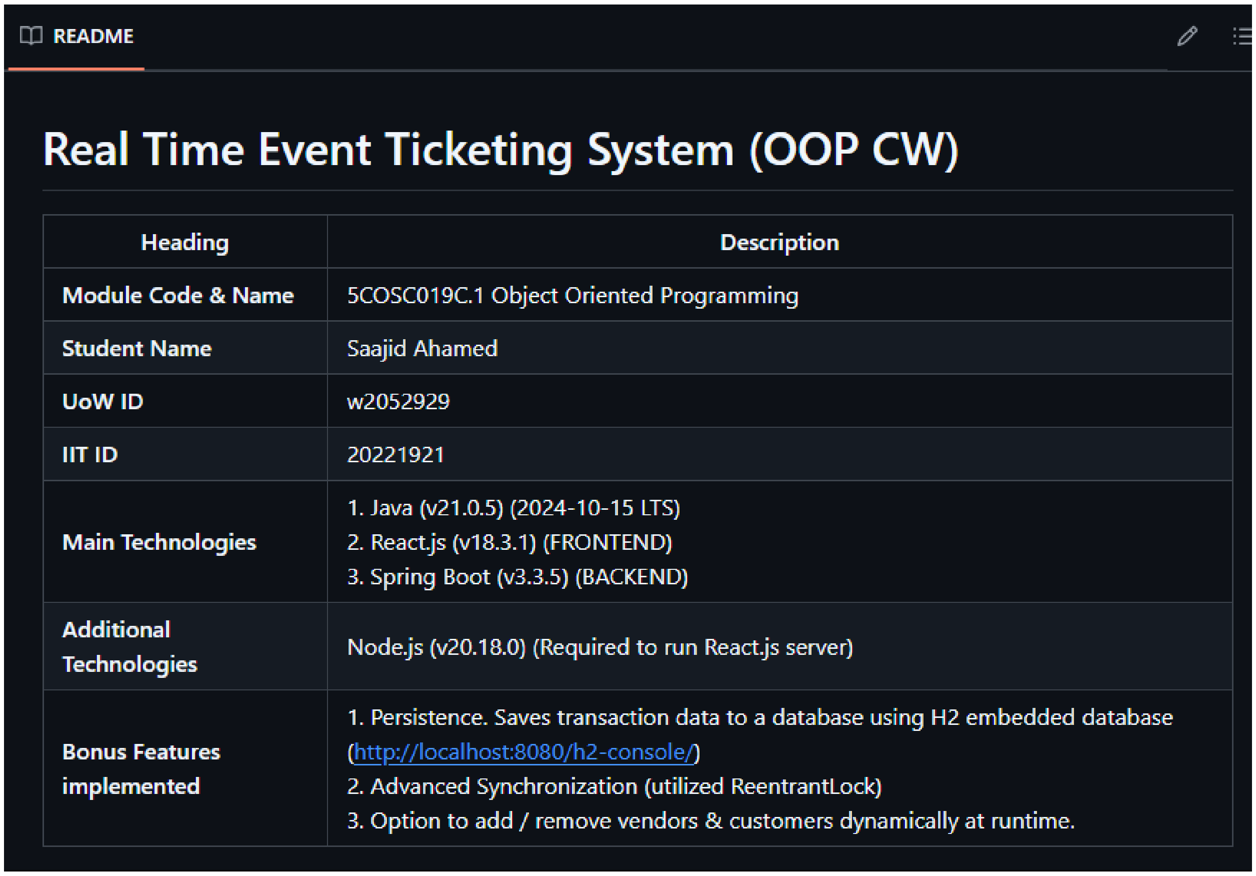
TABLE 1 TEST CASES TABLE

Test Case / Scenario	Output / Observation	Passed / Failed
Simulate multiple customers purchasing tickets (In CLI & GUI)	Multiple customer threads were created and tickets with unique ID's were purchased	Passed
Simulate multiple Vendors releasing tickets (In CLI & GUI)	Multiple Vendor threads were created and tickets with unique ID's were released	Passed
(Deadlock Test) Simulate extremely higher number of Customers than Vendors and check if the indefinite wait of Customers with deadlock is managed due to ticket pool being empty for a prolonged period. (In CLI & GUI)	Additional waiting customer threads were terminated due to no tickets being released to the ticket tool based on a timeout. Deadlock was managed.	Passed
(Deadlock Test) Simulate extremely higher number of Vendors than Customers and check if the indefinite wait of Vendors with deadlock is managed due to ticket pool being Full for a prolonged period. (In CLI & GUI)	Additional waiting vendor threads were terminated due to no tickets being purchased and the ticketpool being full based on a timeout. Deadlock was managed.	Passed
CLI & GUI inputs by user are validated and errors handled gracefully.	Repromoted when invalid inputs.	Passed
Transaction data occurring via the GUI is stored in the database.	Transactions were store in the in memory h2-database system. (SQL)	Passed
Independent Vendor/ Customer option to release / purchase tickets from a shared ticket pool in the GUI	Options were available and purchases/ releases worked correctly.	Passed
Ability to initialize a new ticket pool / reset (erase) an existing ticket pool in the GUI	Option was available. Initalization worked correctly. Reset worked correctly.	Passed

Log all system activities and display it in Real Time (In CLI & GUI) & save the logs to a file	All logs were stored in a list and were logged accurately with clear descriptions. Utilized java.util.logger framework.	Passed
Serialize Configuration to JSON file & text file format	Worked as expected.	Passed
Real time ticket data updates in the frontend GUI	Real time ticket updates were delivered to the frontend utilizing periodic polling methods using RESTful API calls at a regular interval.	Passed

4. README.md File Setup, Instructions & Assumptions

FIGURE 3 README 1

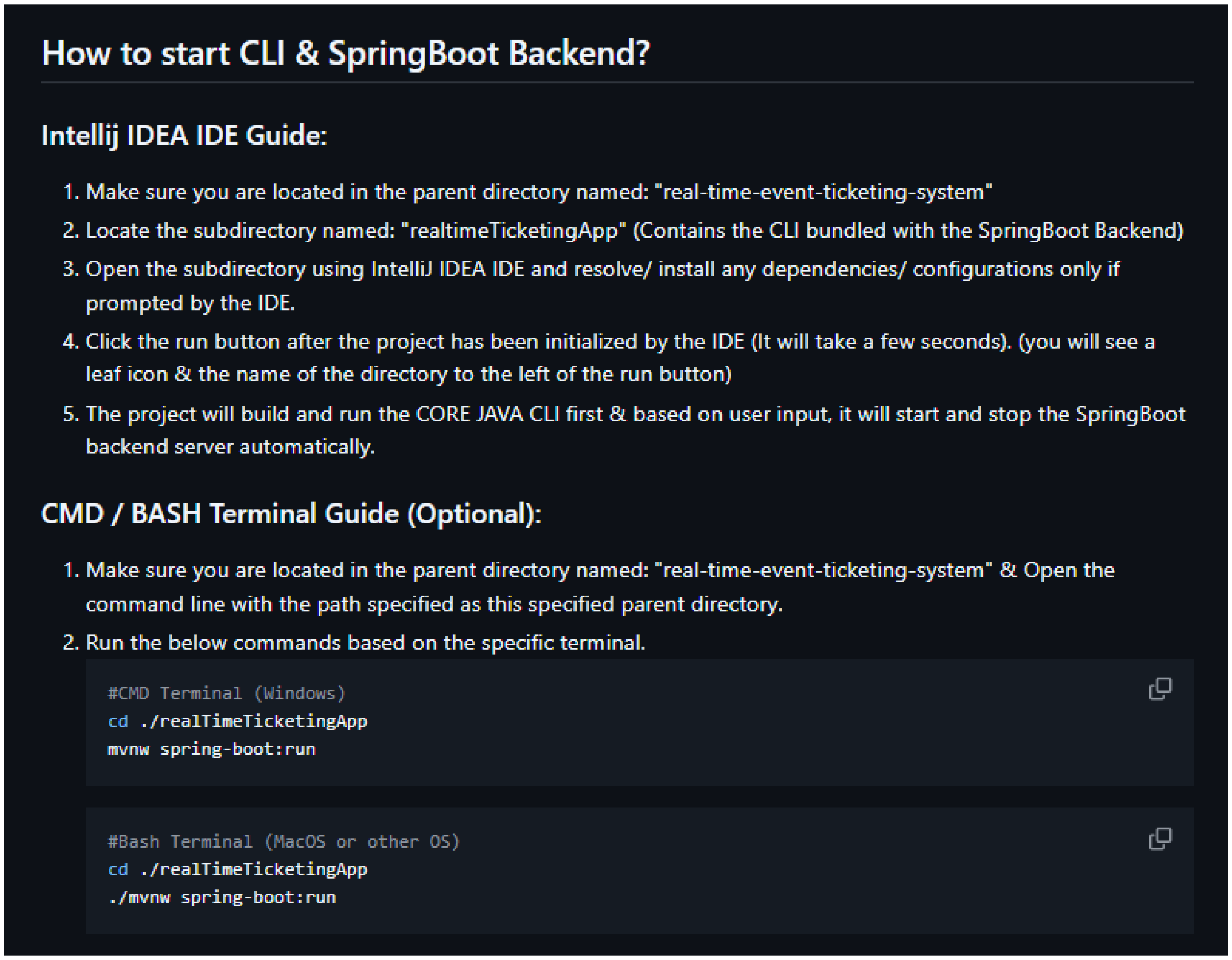


README

Real Time Event Ticketing System (OOP CW)

Heading	Description
Module Code & Name	5COSC019C.1 Object Oriented Programming
Student Name	Saajid Ahamed
UoW ID	w2052929
IIT ID	20221921
Main Technologies	1. Java (v21.0.5) (2024-10-15 LTS) 2. React.js (v18.3.1) (FRONTEND) 3. Spring Boot (v3.3.5) (BACKEND)
Additional Technologies	Node.js (v20.18.0) (Required to run React.js server)
Bonus Features implemented	1. Persistence. Saves transaction data to a database using H2 embedded database (http://localhost:8080/h2-console/) 2. Advanced Synchronization (utilized ReentrantLock) 3. Option to add / remove vendors & customers dynamically at runtime.

FIGURE 4 README 2



How to start CLI & SpringBoot Backend?

Intellij IDEA IDE Guide:

1. Make sure you are located in the parent directory named: "real-time-event-ticketing-system"
2. Locate the subdirectory named: "realtimeTicketingApp" (Contains the CLI bundled with the SpringBoot Backend)
3. Open the subdirectory using IntelliJ IDEA IDE and resolve/ install any dependencies/ configurations only if prompted by the IDE.
4. Click the run button after the project has been initialized by the IDE (It will take a few seconds). (you will see a leaf icon & the name of the directory to the left of the run button)
5. The project will build and run the CORE JAVA CLI first & based on user input, it will start and stop the SpringBoot backend server automatically.

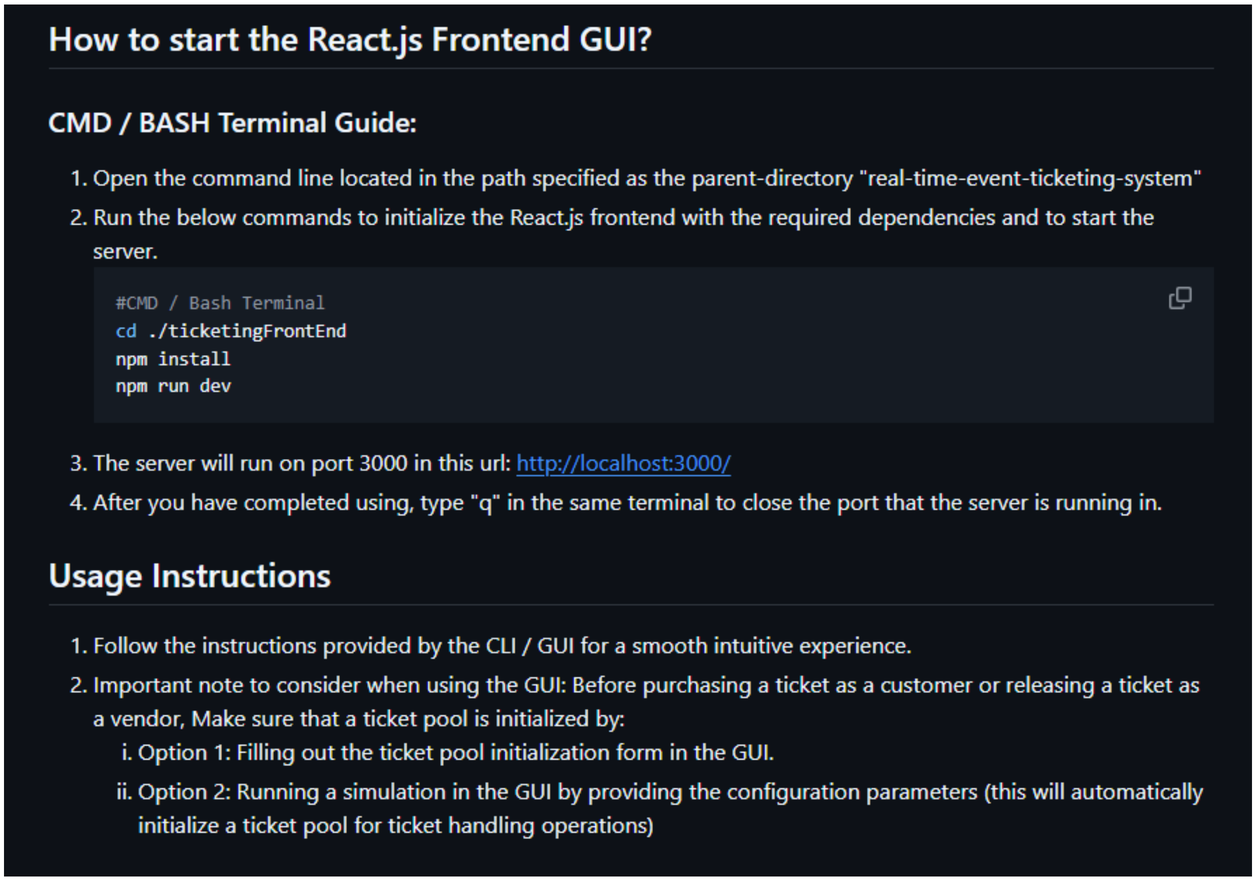
CMD / BASH Terminal Guide (Optional):

1. Make sure you are located in the parent directory named: "real-time-event-ticketing-system" & Open the command line with the path specified as this specified parent directory.
2. Run the below commands based on the specific terminal.

```
#CMD Terminal (Windows)
cd ./realTimeTicketingApp
mvnw spring-boot:run
```

```
#Bash Terminal (MacOS or other OS)
cd ./realTimeTicketingApp
./mvnw spring-boot:run
```

FIGURE 5 README 3



How to start the React.js Frontend GUI?

CMD / BASH Terminal Guide:

1. Open the command line located in the path specified as the parent-directory "real-time-event-ticketing-system"
2. Run the below commands to initialize the React.js frontend with the required dependencies and to start the server.

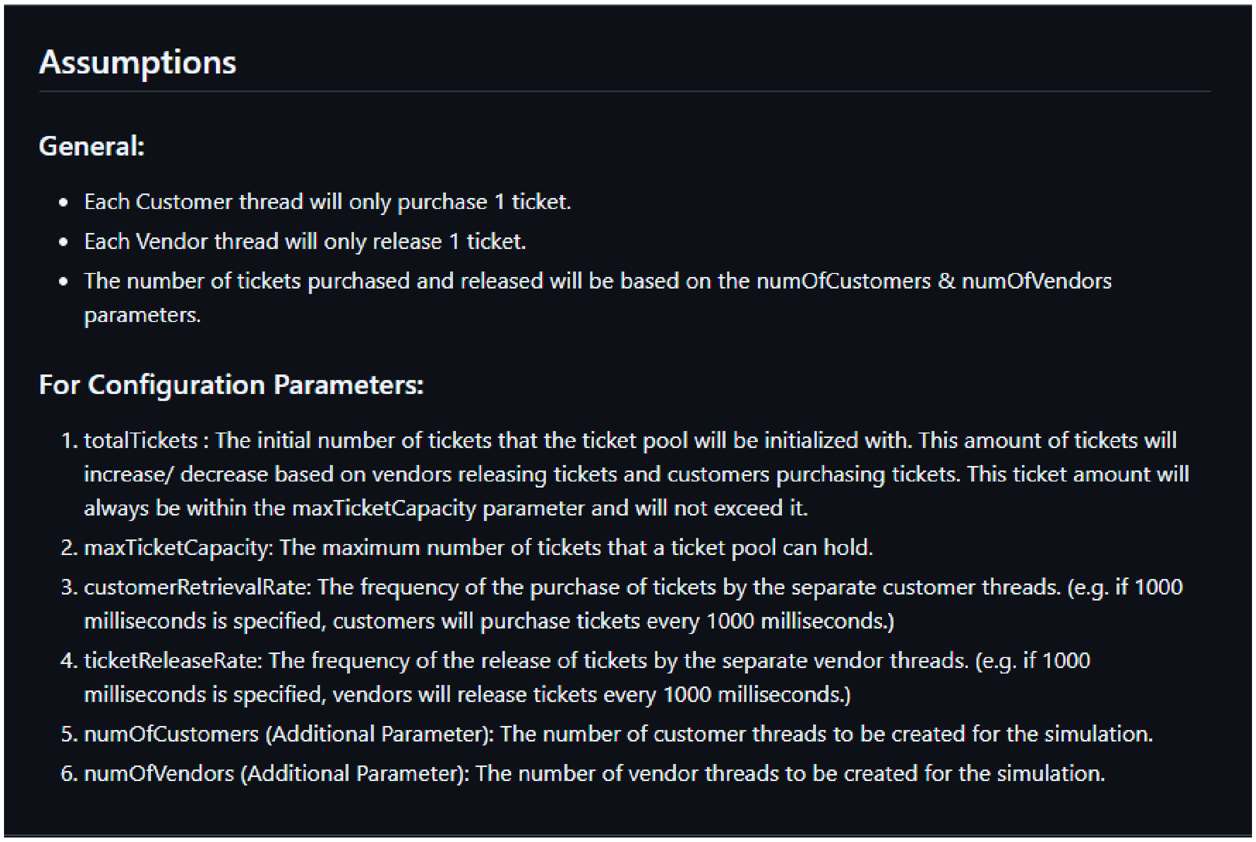
```
#CMD / Bash Terminal
cd ./ticketingFrontEnd
npm install
npm run dev
```

3. The server will run on port 3000 in this url: http://localhost:3000/
4. After you have completed using, type "q" in the same terminal to close the port that the server is running in.

Usage Instructions

1. Follow the instructions provided by the CLI / GUI for a smooth intuitive experience.
2. Important note to consider when using the GUI: Before purchasing a ticket as a customer or releasing a ticket as a vendor, Make sure that a ticket pool is initialized by:
   i. Option 1: Filling out the ticket pool initialization form in the GUI.
   ii. Option 2: Running a simulation in the GUI by providing the configuration parameters (this will automatically initialize a ticket pool for ticket handling operations)

FIGURE 6 README 4



Assumptions

General:

- Each Customer thread will only purchase 1 ticket.
- Each Vendor thread will only release 1 ticket.
- The number of tickets purchased and released will be based on the numOfCustomers & numOfVendors parameters.

For Configuration Parameters:

1. totalTickets : The initial number of tickets that the ticket pool will be initialized with. This amount of tickets will increase/ decrease based on vendors releasing tickets and customers purchasing tickets. This ticket amount will always be within the maxTicketCapacity parameter and will not exceed it.
2. maxTicketCapacity: The maximum number of tickets that a ticket pool can hold.
3. customerRetrievalRate: The frequency of the purchase of tickets by the separate customer threads. (e.g. if 1000 milliseconds is specified, customers will purchase tickets every 1000 milliseconds.)
4. ticketReleaseRate: The frequency of the release of tickets by the separate vendor threads. (e.g. if 1000 milliseconds is specified, vendors will release tickets every 1000 milliseconds.)
5. numOfCustomers (Additional Parameter): The number of customer threads to be created for the simulation.
6. numOfVendors (Additional Parameter): The number of vendor threads to be created for the simulation.