UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

University Of Westminster

Informatics Institute of Technology

Object Oriented Programming

5COSC019C.1

Real-Time Ticketing System

Tutorial Group 11 (SE)

K.P. Seniru Sonal Pathirana

w2051618

20230283

Table of Contents

Introduction

Ticket Management System is an academic project which aims to simulate a real world ticket booking system using a sophisticated software application. The purpose of this project is to show various important concepts of software engineering such as object oriented programming (OOP), multi threading and database management. The system is built on a solid technology stack, with Spring Boot for the backend, React for the interface, and H2 Database for in memory database for testing and persistence. Thanks to this combination of technologies, a dynamic, responsive, and efficient application, which is a good example of modern software development practices, is created.

The main features of the system are that vendors can add tickets and customers can purchase these tickets concurrently. In addition to showing concurrent programming, this feature set shows efficient design system principles. The Ticket Management System provides an opportunity to simulate real world operations to gain insights into the development of scalable and reliable enterprise applications.

Project Overview

The Ticket Management System is designed to replicate the operational dynamics of ticket sales and purchases. It aims to provide a seamless user experience while incorporating critical backend processes that ensure data integrity and system performance. The key functionalities of the system include:

1. **Adding Tickets:** Vendors can add new tickets to the system, simulating the availability of tickets for various events or services.
2. **Purchasing Tickets:** Customers can purchase tickets concurrently, showcasing the system's ability to handle multiple transactions simultaneously.
3. **Database Integration:** The system tracks all ticket addition and purchase operations using an H2 database, ensuring that data is managed efficiently and accurately.
4. **Graphical User Interface:** A React-based graphical user interface allows users to interact with the system, view current status, and configure settings.
5. **Concurrency:** The system demonstrates multi-threading by allowing multiple vendors and customers to interact with it concurrently, thereby testing the application's performance under load.

Primary Objectives:

- **Implement a scalable and modular system using OOP principles.**
- **Showcase efficient use of multi-threading for handling concurrent operations.**
- **Integrate a robust backend system with a dynamic frontend interface.**
- **Utilize an in-memory database for effective data management and rapid testing.**

Main Classes

The core functionality of the Ticket Management System is built around four key classes: Configuration, Vendor, Customer, and TicketPool. Each class plays a crucial role in maintaining the system's integrity and performance.

1. Configuration Class

The Configuration class is responsible for defining and managing the system's parameters. These parameters include:

- **Total Tickets:** The total number of tickets available in the system at any given time.
- **Ticket Release Rate:** The rate at which new tickets are added to the system by vendors.
- **Customer Retrieval Rate:** The rate at which customers purchase tickets from the system.
- **Maximum Ticket Capacity:** The maximum number of tickets that the system can hold at any given time.
- **Number of Vendors:** Specifies the number of vendor threads that will be run concurrently.
- **Number of Customers:** Specifies the number of customer threads that will be run concurrently.

java

```
public class Configuration {
    private int totalTickets;
    private int ticketReleaseRate;
    private int customerRetrievalRate;
    private int maxTicketCapacity;
    private int noOfVendors;
    private int noOfCustomers;

    // Getters and Setters
}
```

This class ensures that all operations within the system adhere to the defined constraints, promoting consistency and preventing potential issues such as overflows.

2. TicketPool Class

The TicketPool class serves as the central repository for all tickets. It provides synchronized methods for adding and removing tickets, which is essential for maintaining thread safety in a multi-threaded environment.

Key Responsibilities:

- **addTickets:** Adds tickets to the pool while adhering to system constraints to prevent exceeding capacity.
- **removeTickets:** Allows customers to purchase tickets, ensuring the pool does not run out of available tickets.
- **Thread Safety:** Uses synchronized methods to avoid race conditions and ensure that multiple threads can safely interact with the ticket pool.

java

```
public class TicketPool {
    private List<Integer> tickets;
    private Configuration config;

    public TicketPool(List<Integer> tickets, Configuration config) {
        this.tickets = tickets;
        this.config = config;
    }

    public synchronized void addTickets(int ticket, int VendorID) {
        this.tickets.add(ticket);
        config.setTotalTickets(tickets.size());
    }

    public synchronized void removeTickets(int BuyerID) {
        this.tickets.remove(0);
        config.setTotalTickets(tickets.size());
    }
}
```

This class plays a critical role in managing tickets efficiently while maintaining the system's integrity and ensuring that operations are executed correctly.

3. Vendor Class

The Vendor class simulates the process of ticket addition by vendors. Each vendor operates as a separate thread, adding tickets to the TicketPool at a specified rate. This simulates the real-world scenario where multiple vendors might be adding tickets to the system concurrently.

Key Responsibilities:

- **Multi-threading:** Implements Runnable to enable concurrent ticket addition by multiple vendors.

java

```
public class Vendor implements Runnable {
    private TicketPool ticketPool;
    private int vendorID;
    private boolean running;
    private int noOfTickets;
    public Vendor(TicketPool ticketPool, int vendorID, int noOfTickets) {
        this.ticketPool = ticketPool;
        this.vendorID = vendorID;
        this.noOfTickets = noOfTickets;
    }

    @Override
    public void run() {
        while (running) {
            if (noOfTickets > 0) {
                ticketPool.addTickets(vendorID);
                noOfTickets--;
            }
        }
    }

    public void start() {
        running = true;
        new Thread(this).start();
    }

    public void stop() {
        running = false;
    }
}
```

This class demonstrates the use of multi-threading and synchronized interactions with shared resources, ensuring that tickets are added to the system efficiently and safely.

4. Customer Class

The Customer class represents the customers who purchase tickets. Like the Vendor class, it operates as a separate thread, simulating the process of concurrent ticket purchases.

Key Responsibilities:

- **Multi-threading:** Simulates customer operations in parallel to demonstrate how the system handles multiple customer interactions simultaneously.

java

```
public class Customer implements Runnable {
    private TicketPool ticketPool;
    private int buyerID;
    private boolean running;
    private int noOfTickets;

    public Customer(TicketPool ticketPool, int buyerID, int noOfTickets) {
        this.ticketPool = ticketPool;
        this.buyerID = buyerID;
        this.noOfTickets = noOfTickets;
    }

    @Override
    public void run() {
        while (running) {
            if (noOfTickets > 0) {
                ticketPool.removeTickets(buyerID);
                noOfTickets--;
            }
        }
    }

    public void start() {
        running = true;
        new Thread(this).start();
    }

    public void stop() {
        running = false;
    }
}
```

This class highlights the system's ability to efficiently handle concurrent operations, ensuring that customers can purchase tickets without conflicts or delays.

Graphical User Interface

The graphical user interface (GUI) for the Ticket Management System is built using React, a popular JavaScript library for building user interfaces. The GUI provides a dynamic and interactive platform for users to monitor and control the system.

Features

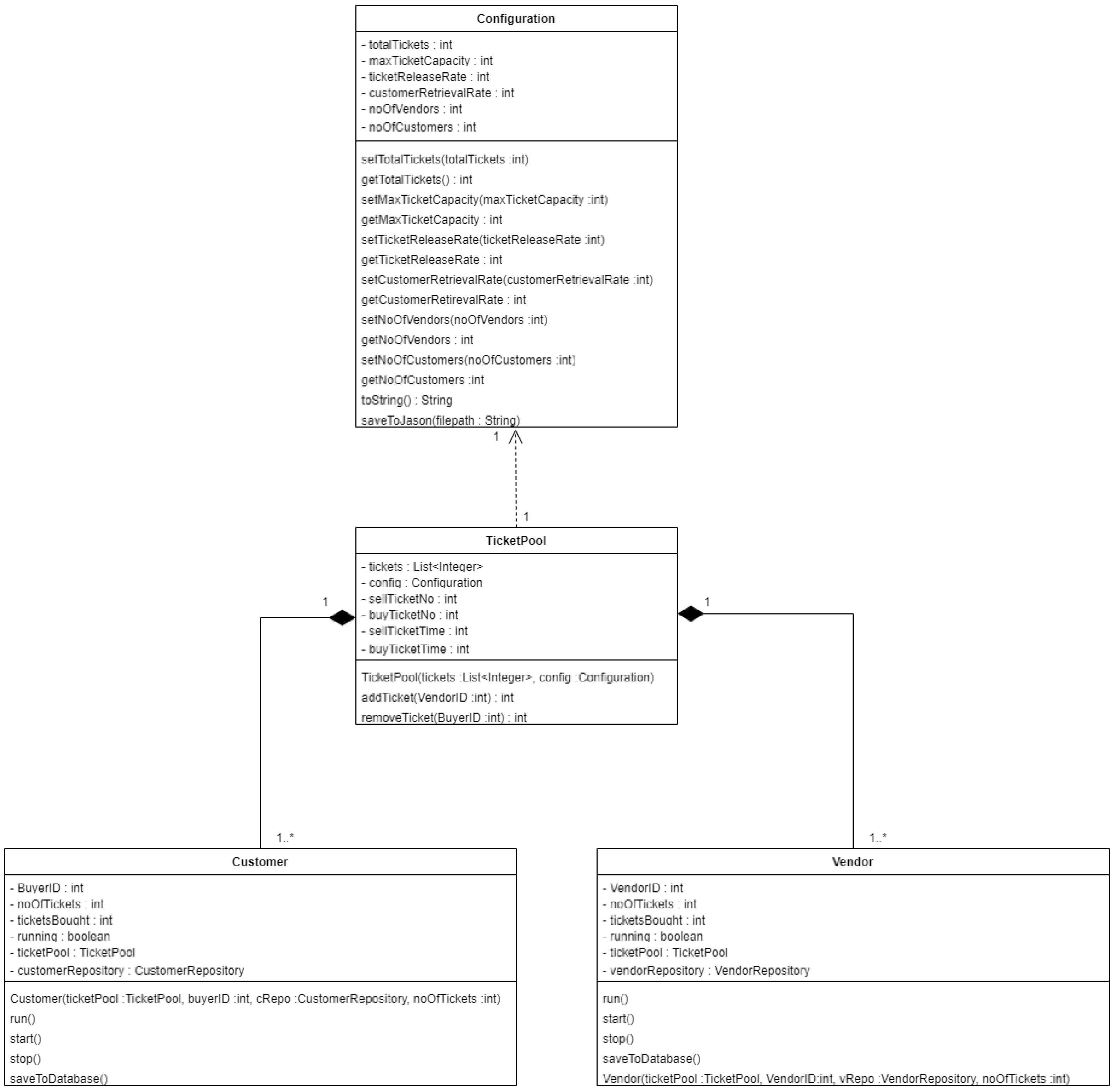
- **Real-Time Updates:** Displays the number of available tickets in the system, providing users with up-to-date information on ticket availability.
- **Configuration Inputs:** Allows users to input configuration parameters such as the number of vendors, customers, ticket release rate, and customer retrieval rate.
- **Start/Stop Controls:** Provides buttons to start and stop the ticketing system, giving users control over system operations.
- **User-Friendly Layout:** Ensures an intuitive and seamless user experience, making it easy for users to interact with the system.

Key Components

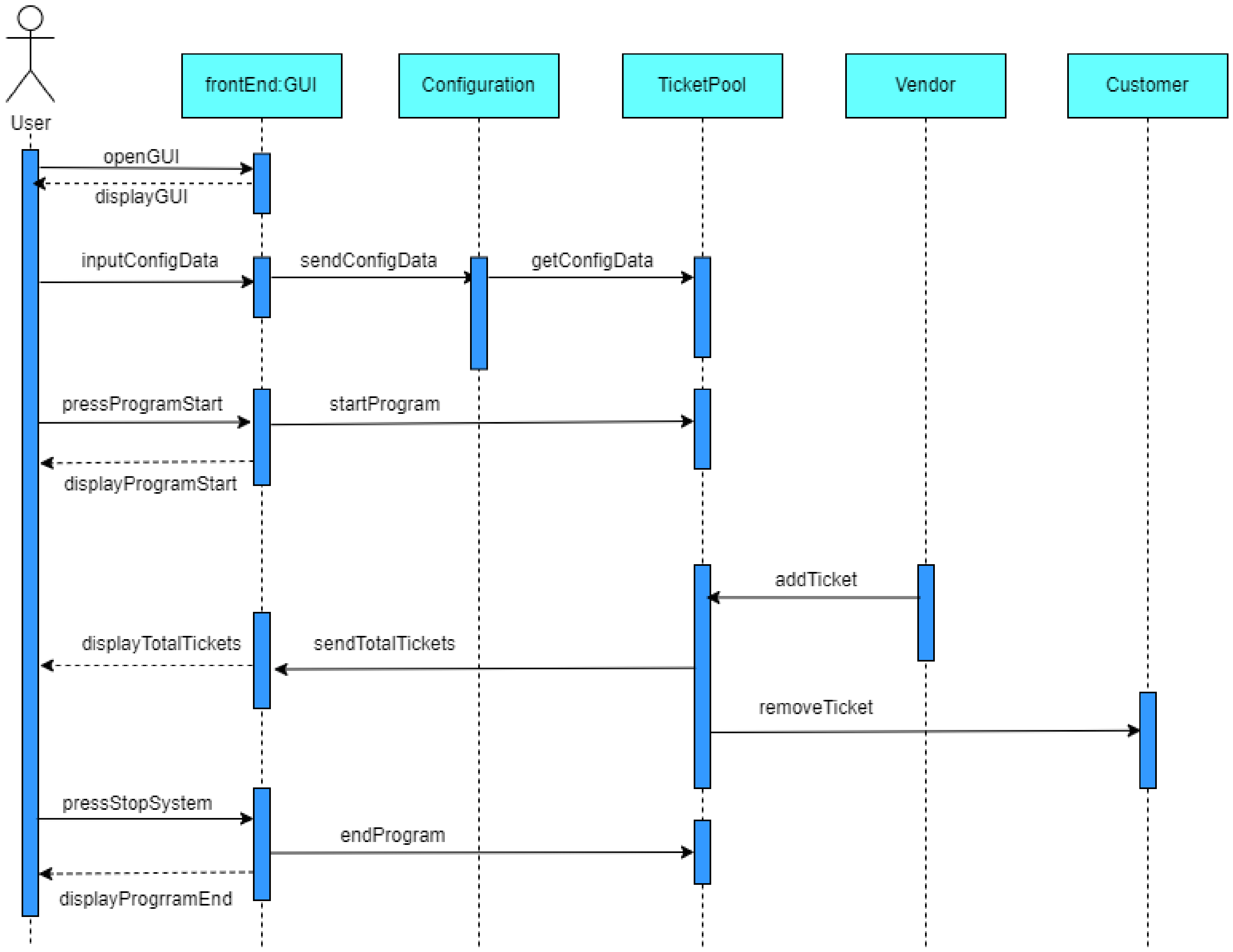
1. **Configuration Form:** A React form component that allows users to enter configuration parameters, making it easy to customize the system's settings.
2. **Live Status Display:** A component that shows real-time ticket availability and system activity, helping users stay informed about the current state of the system.

3. **Control Buttons:** Buttons that allow users to start and stop the system, triggering backend operations via API calls and providing a convenient way to manage system operations.

Class Diagram


Configuration
- totalTickets : int
- maxTicketCapacity : int
- ticketReleaseRate : int
- customerRetrievalRate : int
- noOfVendors : int
- noOfCustomers : int
setTotalTickets(totalTickets :int)
getTotalTickets() : int
setMaxTicketCapacity(maxTicketCapacity :int)
getMaxTicketCapacity : int
setTicketReleaseRate(ticketReleaseRate :int)
getTicketReleaseRate : int
setCustomerRetrievalRate(customerRetrievalRate :int)
getCustomerRetirevalRate : int
setNoOfVendors(noOfVendors :int)
getNoOfVendors : int
setNoOfCustomers(noOfCustomers :int)
getNoOfCustomers :int
toString() : String
saveToJason(filepath : String)
1
1
TicketPool
- tickets : List<Integer>
- config : Configuration
- sellTicketNo : int
- buyTicketNo : int
- sellTicketTime : int
- buyTicketTime : int
TicketPool(tickets :List<Integer>, config :Configuration)
addTicket(VendorID :int) : int
removeTicket(BuyerID :int) : int
1
1
1..*
1..*
Customer
- BuyerID : int
- noOfTickets : int
- ticketsBought : int
- running : boolean
- ticketPool : TicketPool
- customerRepository : CustomerRepository
Customer(ticketPool :TicketPool, buyerID :int, cRepo :CustomerRepository, noOfTickets :int)
run()
start()
stop()
saveToDatabase()
Vendor
- VendorID : int
- noOfTickets : int
- ticketsBought : int
- running : boolean
- ticketPool : TicketPool
- vendorRepository : VendorRepository
run()
start()
stop()
saveToDatabase()
Vendor(ticketPool :TicketPool, VendorID:int, vRepo :VendorRepository, noOfTickets :int)

Sequence Diagram



Test Cases

Scenario	Inputs	Expected Output	Output	Pass/Fail
Multiple Vendors adding tickets.	totalTickets:12 ticketReleaseRate:5 customerRetrievalRate:0 maxTicketCapacity:50 noOfVendors:10 noOfCustomers:0	Multiple vendors added tickets.	Multiple vendors added tickets. Single Customer bought Tickets.	Pass
Multiple Customers buying tickets.	totalTickets:40 ticketReleaseRate:0 customerRetrievalRate:3 maxTicketCapacity:50 noOfVendors:0 noOfCustomers:10	Multiple Customer bought tickets.	Multiple Customer bought tickets.	Pass
TicketPool Reaching Maximum Capacity.	totalTickets:10 ticketReleaseRate:1 customerRetrievalRate:0 maxTicketCapacity:10 noOfVendors:1 noOfCustomers:0	Vendor1 Ticket pool is full...... Available tickets : 10	Vendor1 Ticket pool is full...... Available tickets : 10	Pass
Handling of TicketPool reaching zero.	totalTickets:0 ticketReleaseRate:0 customerRetrievalRate:3 maxTicketCapacity:10 noOfVendors:0 noOfCustomers:1	Buyer1 No ticket available.... Available tickets : 0	Buyer1 No ticket available.... Available tickets : 0	Pass
Handling of Customer Retrieval Rate.	totalTickets:10 ticketReleaseRate:0 customerRetrievalRate:2 maxTicketCapacity:20 noOfVendors:0 noOfCustomers:1	Buyer1 bought ticket.... Available tickets : Buyer1 bought ticket.... Available tickets : 8 Buyer1 Retrieval rate exceeded.... Waiting - 12S	Buyer1 bought ticket.... Available tickets : Buyer1 bought ticket.... Available tickets : 8 Buyer1 Retrieval rate exceeded.... Waiting - 12S	Pass
Handling of Ticket Rlease Rate.	totalTickets:0 ticketReleaseRate:2 customerRetrievalRate:0 maxTicketCapacity:50 noOfVendors:1 noOfCustomers:10	Vendor1 added ticket.... Available tickets : Vendor1 added ticket.... Available tickets : Vendor1 Release rate exceeded.... Waiting - 15S	Vendor1 added ticket.... Available tickets : Vendor1 added ticket.... Available tickets : Vendor1 Release rate exceeded.... Waiting - 15S	Pass
Simultaneous Start/Stop Commands (GUI)	totalTickets:12 ticketReleaseRate:2 customerRetrievalRate:3 maxTicketCapacity:50	Start the system	Start the System	Pass

	noOfVendors:3 noOfCustomers:4			
Handling Wrong input values for configuration (CLI)	totalTickets:-1	Please enter a positive number ! Enter the total Number of tickets :	Please enter a positive number ! Enter the total Number of tickets :	Pass
Handling wrong input type for configuration (CLI)	totalTickets:r	Enter a valid number ! Enter the total Number of tickets :	Enter a valid number ! Enter the total Number of tickets :	Pass

Conclusion

The Ticket Management System is a large project that includes most of the elements of modern software engineering. The system integrates key principles including object oriented programming, multi threading and efficient database management to provide a practical and scalable solution to ticket booking. As an example of how Spring Boot for backend development supported by React for a dynamic user interface and H2 for in memory database operations can be used to produce a slick and responsive application.

Not only does this project show how the theoretical concepts learned in an academic setting can be applied in a real world application, but it also gives the student first hand experience of building and managing a real world application. Implementation of this work provides a window into concurrent programming, system design, and the intersection of backend and frontend technologies. The Ticket Management System is a great example of how to design and develop enterprise level applications, and is designed to prepare students for the realities of professional software development.

This system in general illustrates the need for modularity, scalability and efficiency in writing of software. This project exhibits the capability to handle concurrent operations, preserve data integrity, and present a user friendly interface, thereby closing the gap between academic learning and real world application.