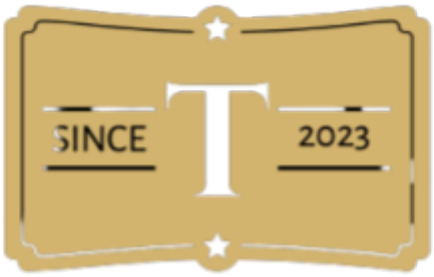


Ticket Purchasing System
C-Document

Prepared by

Kyle Galingan, 501037503

Riddhima Garg, 501100477

Shanaya John, 501108746

Francis Porca, 501102477

Roxie Reginold, 501087897

Barbod Salehinoparvar, 501120154

*Each member contributed equally

Introduction To Software Engineering

CPS 406 (Section 08) - Team 20

April 6, 2023

Table Of Contents

System Domain and Objectives

Ticket Tango enables users to purchase tickets for events like concerts, sports games, and theatre/art productions. The application allows users to create an account, browse events easily, add tickets to their cart, and complete their transactions. After purchasing tickets, the application will update the user's order history where the user can view their tickets. The overall breakdown of the application is as follows:

- ➔ User Management: The application allows for creating and managing user accounts, such as login and sign-up.
- ➔ Event Database: A database that includes details of an event and allows users to view the list of active events and further view information on that event.
- ➔ Customer Database: A database that keeps track of customer and administrator account information and updates itself as more customers create an account.
- ➔ Ticket Selection and Purchasing: a cart functionality that allows users to add tickets to their cart, view them, and purchase them. This section also manages purchasing of tickets where users input credit/debit card information.
- ➔ Interface: Ticket Tango is user-friendly, with easy navigation and a simple design. Colours used in the interface allows users to distinguish the app functionality.

This product aims to provide customers with a safe and convenient platform for purchasing tickets and allow organizers to streamline the tickets for their events.

Development Methodology

The development methodology used for this project is a combination of **Phase model** (Figure 1) and **Spiral model** (Figure 2). Different implementations will be developed in phases, with the spiral model being implemented to direct the prototyping and development of the phase.

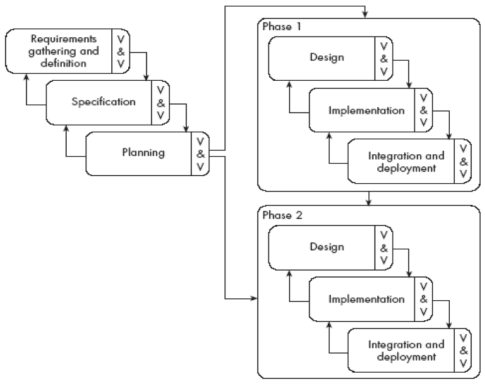


Figure 1: Phase Release Model

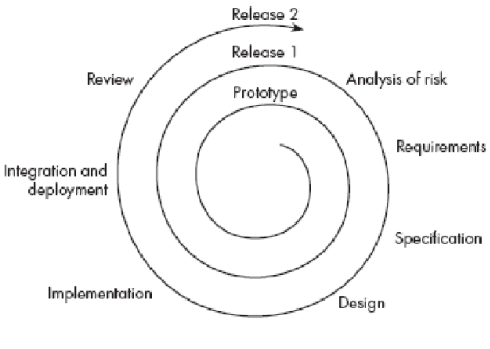


Figure 2: Spiral Model

The implementation of the phase model will ensure specific goals are completed by the end of set due dates. Within each phase, the spiral model is implemented to provide continuous feedback from peers and improve the tasks worked on within the phase. The title of the phases corresponds to the main task to be completed, the workflow for completing the task is done with a spiral methodology with prototypes and improvements being made.

Our project had 3 main releases/spirals; implementation of System classes, our first prototype, and our final product that includes testing/documentation.

The reason why we chose to develop with both the phase release and spiral model is to take advantage of each of their strengths. The phase release model provides a more structured approach that ensures each phase is completed thoroughly before moving on, while the Spiral model allows for flexibility and adaptability as we move through the phases or as problems arise.

Team Responsibilities and Role Descriptions

Our team will be functioning using the combination of the Chief programmer and the egoless approach, where all of our subteams are interconnected through tasks and have a team leader. The following is a role breakdown:

Project Manager: Roxie

- ➔ Ensure all requirements are precise, address problems, and oversee each team member's work.

Lead Front-end Developer: Riddhima

- ➔ Designing, developing, and maintaining the user interface using Java. Implement interactive features that make the software user-friendly. Participate in design-based discussion, address feedback and assist with implementation.

Lead Back end Developer: Shanaya

- ➔ Integrate databases, and ensure an effective and secure flow of data. Collaborate with front-end developers to ensure a seamless user experience and an error-proof program.

Lead Tester/Debugging: Kyle

- ➔ Verify the functionality of classes through test cases in our application. Fixing bugs and testing the quality of the product.

Lead Documentor: Francis

- ➔ Ensure that the code documentation accurately reflects the design, implementation, and functionality of the software. This includes documenting the purpose of each code module, the input and output parameters, the data structures used, and any algorithms or logic that are implemented.

Lead Detailed Design: Barbod

- ➔ Create a clear and compelling presentation that effectively communicates the project's goals, objectives, design, implementation, and results to stakeholders.

Tasks Needed (Table 1)

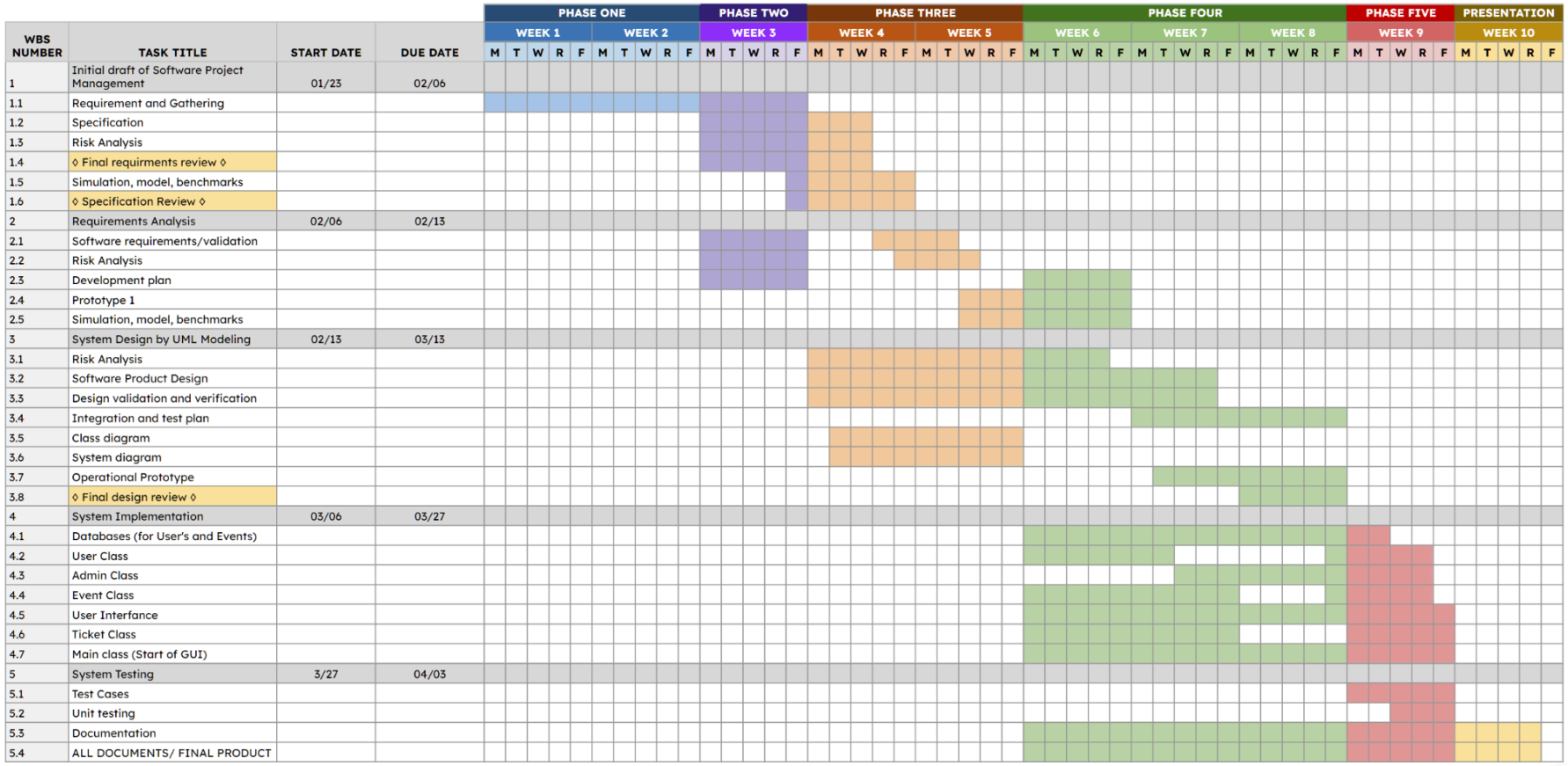
Tasks	Assigned member
Documentation	Francis, Roxie, Kyle
Project management/maintenance	Roxie
Programming	All
Detailed design	Barbod, Riddhima
Feasibility study/Researcher	All
User interface design	All
Validation/ verification	Roxie, Kyle
System Integration	All
Database Design	Shanaya

Table 1: System task breakdown

Gantt Chart (Table 2)

PROJECT TITLE	Ticket Purchasing System	COMPANY NAME	Ticket Tango
PROJECT MANAGER	Roxie R.	DATE	Apr 4, 2023

WBS NUMBER	TASK TITLE	START DATE	DUE DATE
1	Initial draft of Software Project Management	01/23	02/06
1.1	Requirement and Gathering		
1.2	Specification		
1.3	Risk Analysis		
1.4	◊ Final requirments review ◊		
1.5	Simulation, model, benchmarks		
1.6	◊ Specification Review ◊		
2	Requirements Analysis	02/06	02/13
2.1	Software requirements/validation		
2.2	Risk Analysis		
2.3	Development plan		
2.4	Prototype 1		
2.5	Simulation, model, benchmarks		
3	System Design by UML Modeling	02/13	03/13
3.1	Risk Analysis		
3.2	Software Product Design		
3.3	Design validation and verification		
3.4	Integration and test plan		
3.5	Class diagram		
3.6	System diagram		
3.7	Operational Prototype		
3.8	◊ Final design review ◊		
4	System Implementation	03/06	03/27
4.1	Databases (for User's and Events)		
4.2	User Class		
4.3	Admin Class		
4.4	Event Class		
4.5	User Interfance		
4.6	Ticket Class		
4.7	Main class (Start of GUI)		
5	System Testing	3/27	04/03
5.1	Test Cases		
5.2	Unit testing		
5.3	Documentation		
5.4	ALL DOCUMENTS/ FINAL PRODUCT		



Requirement Analysis

Major Goals and Sub-goals

The goal of our software is to have customers be able to purchase a ticket to their desired event and have organizations streamline their events. In association, this will be done through our subgoals of registering new customers and storing information about existing customers, viewing relevant events, adding ticket to cart, and finalizing with payment and confirmation that the ticket has been purchased. Similarly, the organization will be able to register as an organization and propose their event to be included in the system. In addition, the admin will be able to keep track of events, add events and take events from organizations and display it on the software.

Constraints

The constraints on our system are divided into two types; **Design constraints** and **Process constraints**.

Design constraints

- Java
- Windows/MacOS
- Java Swing (GUI)
- .txt file for customer and event database
- Simple Data layout for the system
- Replit

Process constraints

- The application will go through limited amounts of testing after each deployment/prototype according to our development methodology.
- Documentation will be updated for each development phase.
- The final product will be pushed to GitHub.
- Payment connections to bank account will be assumed for our system.
- Our stakeholders could affect the approval of our application, which could affect the development and deployment of our prototypes.

Major Actors and Components

The significant actors include the customer, administrator or management, and organization. Components include view events, customer registration and login, databases to hold account and event information, reserving or cancelling the ticket, adding and removing events, system administrative modifying, and user quality of life navigation pages such as FAQ will be used to achieve the goals of the software.

Table of Functional Requirements (FR) - Table 3

ID	Description
FR-1	User Login/Registration: To purchase a ticket, users must be able to create an account and log in. The registration page will require users to input information such as name, email address, and password.
FR-2	Event Browse/Search: Users can search for tickets on the event table and click on 'view event' to get details about specific events.
FR-3	Event Management: Users must be able to view available tickets for an event, along with the prices, location, date and ticket availability information. This allows users to determine the specific ticket they want to purchase.
FR-4	Payment and Security: Users must be able to purchase ticketing through the payment gateways. This implementation will check the input information's validity and protect user-sensitive data.
FR-5	Profile Management: Users must be able to view their personal information and view order history. This allows users to control their data and will enable them to view their activity on the application.
FR-6	Admin panel: Administrators must be able to manage events, update ticket information, access user information and manage other application functionality. The admins will have a separate interface with the necessary permissions and functionalities.
FR-7	Ticket management: Users must be able to cancel orders and receive a refund for events they no longer wish to attend. Cancellation and refunds will be given based on the terms and conditions of the event organizers.
FR-8	User Interface: The application has a user-friendly interface that provides users with clear instructions on purchasing tickets. The system will utilize colours, fonts, and images to make the application easy to navigate and understand.
FR-9	Cart Management: Users must be able to add, remove and check out items from the shopping cart. This functionality enables users to purchase multiple tickets from multiple events with a single transaction.
FR-10	Databases: The systems store information about events, users, and tickets availability in the database. The database is designed to be secure and handle a large amount of data.

Table of Non-Functional Requirements (NFR) - Table 4

ID	Description
NFR01	Reserving a ticket: There should be a time limit of 5 minutes on the time a user spends on the checkout page.
NFR02	Secure Login (verification): The system should prompt the user for confirmation by entering a code sent through email or phone. The user should receive that code within 2 minutes.
NFR03	Check ticket availability: The system should check if there are any tickets available before the customer proceeds to order/cart.
NFR04	System activity: Our application should be able to withstand many customers depending on the number of users/accounts created.
NFR05	Information Security: Our system should protect user and payment information.
NFR06	Order Summary: A customer's ticket purchase information should be generated within 30 seconds.
NFR07	Filter Search: Selected filters should have resulted in less than a minute.
NFR08	Event information: Any requested event changes to an organization's page should appear within an hour.
NFR09	Usability: The application should be able to be used for 24 hours of the day unless maintenance is being performed.
NFR10	Robustness: Our application should be able to handle incorrect input from payment information and login.
NFR11	Old events do not show up for customers

Table of Assumptions (A) - Table 5

ID	Description
AS-1	Tickets have set prices: assume that all the tickets for an event have set prices determined by the event organizer.
AS-2	Secure Payment: assume that the payment process is safe and complies with guidance. Assume the user enters valid payment information.
AS-3	User authentication: assume that users have a valid email address and that their information is correct upon sign-up.
AS-4	Stable connection: assume users can access the internet and find the application online.
AS-5	Security/Firewall: the application is secure and protected by firewalls that prevent unauthorized access and protect against cyber attacks.
AS-6	Localization: assume the system supports various languages, currencies, date and time formats, and other cultural/political regulations.
AS-7	Event availability: The event cannot be cancelled or rescheduled.
AS-8	Events are legitimated and authorized: assume all events listed on the system are valid.
AS-9	Customer Support: Assume that users are provided access to customer service where they can address any issues or concerns.
AS-10	Compliance with laws and regulations: the application complies with all relevant laws and regulations, including data privacy, consumer protection, and other applicable regulation.
AS-11	Performance scalability: assume that applications handle large volumes of users, events, and ticket sales.
AS-12	Customers received refund after cancelling tickets for event
AS-13	Customers received an email confirmation and receipt of their purchase.

Table of Use Cases (UC) - Table 6

ID	Name	Description
UC01	Creating an account	This scenario describes a new customer registering for an account.
UC02	Login	This scenario describes an existing customer logging into their account.
UC03	Purchasing a ticket	This scenario describes the process of purchasing a ticket on the application.
UC04	Canceling a purchased ticket	This scenario describes the process of a customer cancelling a ticket.
UC05	Viewing events	This scenario describes a customer wanting to view events.
UC06	Searching events	This scenario describes a user wanting to search for events.
UC07	Removing from cart	This scenario describes a customer adding a ticket to the shopping cart.
UC08	Add to cart	This scenario describes the process of a customer adding a ticket to the shopping cart
UC09	Create an event	This scenario describes the process of creating an event that is displayed on the ticket purchasing application.
UC10	Viewing order history	In this scenario, the customer will have an overview of what they have previously ordered.
UC11	Delete event	This scenario describes the admin deleting an event from the main page and database.
UC12	Add organization	In this scenario, the admin will add the organization so they will have a username and password, and they can request to edit their event or add or delete the event.
UC13	Access customer or organization information	This scenario describes the admin accessing a customer's or organization's account information to resolve an issue. Information being accessed includes email, password and history of transactions.
UC14	View purchased ticket information	This scenario describes a customer wanting to view information on their purchased ticket(s)z
qscqa1wcdwx1	Payment Process	This scenario describes customers inputting their credit/debit card information and receiving a ticket confirmation into their account.
UC16	View cart	This scenario describes the user viewing the contents in their cart.

Table of Use Case Scenarios (UCS)

STD01: Creating an account (Table 7)
Use Case Name: UC #01
Description: This scenario describes a new customer registering for an account.
Primary Actors: • Customer
Pre-condition: • The customer must have a valid email address • The customer must adhere to any privacy policies or terms of service.
Trigger Event: User clicks on Login / Register.
Main Scenario Steps: 1. User includes first and last name. 2. User enters a valid email address and a valid password with special requirements. 3. Use case ends when the customer successfully creates an account.
Special Requirements: • Minimum password conditions   ◦ I.e., it must contain at least n letters, one symbol, one cap, one lowercase, etc.
Success Post-Conditions • The customer account is created • The customer is granted access to customer based menus. • The customer is able to place an order. • The customer can see other user settings • User information is stored in the system.
Failure Post-Conditions • The account is not created. • User information is not stored in the system.
Extensions: None

STD02: Login (Table 8)
Use Case Name: UC#02
Description: This scenario describes an existing customer logging into their account.
Primary Actors: • Customer (Existing)
Pre-condition: • Data of customer information and credentials are stored and registered in the system.
Trigger Event: The user clicks the login button.
Main Scenario Steps: 1. User enters username (email address) and password 2. User clicks "*Login*".
Special Requirements: None
Success Post-Conditions • The customer is logged in after successful verification. • The customer is logged into the system. • The customer can place an order. • The customer can see other user settings.
Failure Post-Conditions • Account is denied access to the system.
Extensions: None

STD03: Purchasing Ticket (Table 9)
Use Case Name: UC#03
Description: This scenario describes the process of purchasing ticket on the application
Primary Actors: • Customer
Pre-condition: • The user must have registered an account on the ticket purchasing system.
Trigger Event: • Customer wants to purchase a ticket for an event
Main Scenario Steps: 1. Customer logs into the application. 2. Customer search for an event. 3. Customer adds the ticket for an event to the cart. 4. Customer selects “proceed to checkout” and chooses a payment method. 5. Customer inputs payment information and confirms the transaction. 6. System processes the payment and issues the ticket. 7. Customer returns to the event page.
Special Requirement: • The cart is not empty. • The payment process is approved.
Post-condition Failure: • The customer’s Payment was not processed, and they were not issued the ticket(s)
Post-condition Success: • The customer’s payment was processed and they were issued the ticket(s) • The customer account information was updated to include the purchase information • The event organizers were notified of the ticket sale and the event updated the seat availability.
Extensions: • If the Payment is declined, an error message will display.

STD04: Ticket Cancellation (Table 10)
Use Case Name: UC #04
Description: This scenario describes the process of a customer canceling a ticket.
Primary Actors: • Customer
Pre-condition: • The customer has purchased a ticket for an event and wishes to cancel the purchase.
Trigger Event: • Customer requests cancellation for a ticket purchase
Main Scenario Steps: 1. Customer log into the application 2. Navigator to “My Tickets” section 3. Customer selects the ticket they want to cancel. 4. System displays information about the selected ticket and the user clicks the “Cancel Purchase” button. 5. The system displays a prompt and asks the user for confirmation. 6. Customers confirm the cancellation, if applicable. 7. System displays a confirmation that the purchase is cancelled and refunds the customer.
Special Requirement: None
Post-condition Failure: • The ticket purchase could not be canceled
Post-condition Success: • The ticket purchase is cancelled, and the customer is notified of the cancellation.
Extensions: • If the ticket is not eligible for cancellation, the system will display an error message explaining to the customer why the cancellation cannot be completed. • If the user experiences an issue during the cancellation process, they can either go to the FAQ page or contact customer support directly for assistance.

STD05: View Events (Table 11)
Use Case Name: UC#05
Description: This scenario describes a customer wanting to view events.
Primary Actors: • Customer
Pre-condition: • Information about the event (time, Date, etc.) is available in the database
Trigger Event: • User navigates to the view events section of our application.
Main Scenario Steps: 1. User enters the view events page 2. The current events are displayed on the page
Special Requirements: None
Success Post-Conditions • Events are successfully shown and viewed.
Failure Post-Conditions • A planned event is not shown or can be viewed within the system.
Extensions: None

STD06: Search events (Table 12)
Use Case Name: UC #06
Description: This scenario describes a user wanting to search for events.
Primary Actors: • Customer
Pre-condition: • Name of the event is stored in the database.
Trigger Event: • User goes to the search bar to look for events.
Main Scenario Steps: 1. User enters words into the search bar to look for names of events 2. Use case ends when search results appear.
Special requirements: • Name matches the title of the event(s) stored on the system.
Success Post-Conditions: • Event that was searched for was successfully brought up. • Event information is shown.
Failure Post-Conditions: • Search bar is unable to find the event.
Extensions: • The customer has access to FAQ if they have any questions.

STD07: Remove from cart (Table 13)
Use Case Name: UC #07
Description: This scenario describes the process of a customer removing a ticket from the shopping cart
Primary Actors: • Customer
Pre-condition: • The customer must be register and logged in • The customer must have a ticket in their shopping cart.
Trigger Event: • Customer wishes to remove a ticket from their cart
Main Scenario Steps: 1. The customer navigates to their shopping cart. 2. The customer selects the ticket they wish to remove from their cart. 3. The customer clicks the event in their cart that they wish to remove. 4. The item is removed from the customer's cart. 5. The system updates the cart total.
Special Requirement: None
Post-condition Failure: • The item could not be removed
Post-condition Success: • The item is removed from their cart • The cart total is updated
Extensions: • If any issues or concerns arise, the customer can contact customer support or refer to FAQ.

STD08: Add to Cart (Table 14)
Use Case Name: UC #08
Description: This scenario describes the process of a customer adding a ticket to the shopping cart
Primary Actors: • Customer
Pre-condition: • The customer is registered and logged in • The customer wishes to purchase a ticket
Trigger Event: • Customer decides to add ticket to their cart
Main Scenario Steps: 1. The customer finds an event they want to attend and tickets are available. 2. The customer clicks on the "Add to Cart" button. 3. The system adds the ticket to the customer's cart and updates the cart total.
Special Requirements: • In the cart, items are reserved until the program is exited.
Failure Post-Conditions: • The ticket is no longer available and cannot be add to the cart • The ticket could not be add to the cart due to some error
Success Post-Conditions: • The ticket is added to customer’s cart • The cart total is updated
Extensions: • If an process cannot be completed, the customer is notified

STD09: Create an Event (Table 15)
Use Case Name: UC #09
Description: This scenario describes the process of creating an event that is displayed on the ticket purchasing application.
Primary Actors: • Administration
Pre-condition: • The organizer must have requested (with specific details) an event they wish to host.
Trigger Event: • Organizer wish to host an event
Main Scenario Steps: 1. System administration reviews the organizer's request 2. The admin sets the event name, Date, location, and number of tickets based on the organizer's request. 3. The admin sets the prices for all tickets and makes the event available.
Special Requirements: None
Failure Post-Conditions: • The event could not be created
Success Post-Conditions: • The event was created and is available on the application
Extensions: None

STD10: Viewing order history (Table 16)
Use Case Name: UC #10
Description: In this scenario, the customer will have an overview of what they have previously ordered.
Primary Actors: • Customer
Pre-condition: • The customer has previously purchased a ticket
Trigger Event: • Customer wants to view previous purchases.
Main Scenario Steps: 1. The customer clicks “Profile” 2. The customer sees a list of previous tickets purchased.
Special Requirement: None
Post-condition Failure: • The customer is unable to view previous purchases
Post-condition Success: • The customer can previous all of their previous purchases
Extensions: None

STD11: Delete event (Table 17)
Use Case Name: UC #11
Description: This scenario describes the admin deleting an event from the main page and database.
Primary Actors: • Admin
Pre-condition: • The event should already exist.
Trigger Event: • Admin wants to Remove an event.
Main Scenario Steps: 1. Admin logs into the system using “Admin Login”. 2. Admin selects events that are to be removed. 3. Admin selects “Remove Event”
Special Requirement: None
Post-condition Failure: • The event could not be deleted
Post-condition Success: • The event was successfully deleted
Extensions: • If the event does not exist, it will generate an error message.

STD12: Add organization (Table 18)
Use Case Name: UC #12
Description: In this scenario, the admin will add the organization so they have a username and password, and they can request to edit their event or add or delete the event.
Primary Actors: • Admin
Pre-condition: • The organization is valid and is not a scam.
Trigger Event: • The organization wants to be added to our website.
Main Scenario Steps: 1. Admin will go to the modifying organization. 2. Admin will choose to add the organization. 3. Generate a username and password for the organization and add it to the Database.
Special Requirement: None
Post-condition Failure: • The organization failed to be added to the database.
Post-condition Success: • The organization was successfully added to the Database.
Extensions: None

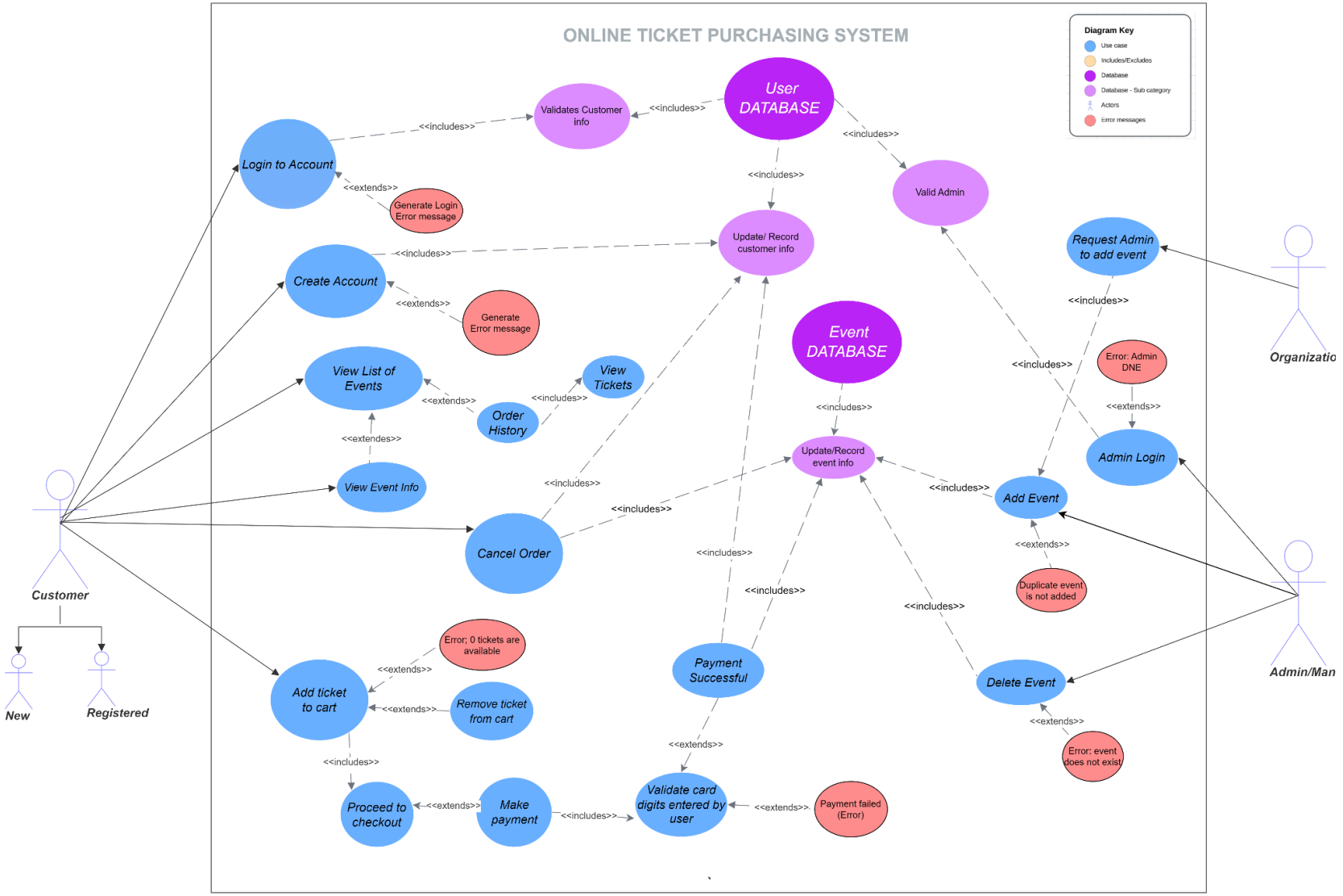
STD13: Access customer or organization information (Table 19)
Use Case Name: UC #13
Description: This scenario describes the admin accessing a customer's or organization's account information to resolve an issue. Information being accessed includes email, password and history of transactions.
Primary Actors: • Administrator/ Management
Pre-condition: • Customer/organization has an account • There's an issue that requires the admin's attention
Trigger Event: • Customer/organization asks the admin to check their account.
Main Scenario Steps: 1. Admin logs into their system 2. Searches for the customer/organization's account details 3. System returns account information
Success Post-Conditions: • Account information is outputted to the admin
Failure Post-Conditions: • Account does not exist
Extensions: None

STD14: View purchased ticket information (Table 20)
Use Case Name: UC #14
Description: This scenario describes a customer wanting to view information on their purchased ticket(s)
Primary Actors: • Customer
Pre-condition: • Customer has purchased tickets
Trigger Event: • Customer clicks on Order History in their account
Main Scenario Steps: 1. The customer selects view my tickets 2. A list of tickets they have purchased is returned 3. User clicks on the ticket info they want to view 4. The system displays the prices, dates, locations, and other ticket information.
Success Post-Conditions: • Ticket info is displayed to the customer
Failure Post-Conditions: • Customer is unable to view ticket ○ No purchase history ○ The tickets they purchased have been canceled by the organizer or admin
Extension: None

STD15: Payment Process (Table 21)
Use Case Name: UC #15
Description: This scenario describes customers inputting their credit/debit card informationPoshAllModules and receiving a ticket confirmation into their account.
Primary Actors: • Customer
Pre-condition: • User has reviewed their cart and proceeds to checkout.
Trigger Event: • User selects a payment method after reviewing Order Summary
Main Scenario Steps: 1. Users input their billing address and card details into the required fields. 2. The system verifies that every input is the data type that is required ○ Ensures that integer values are inputted and no extra digits 3. Customer clicks “Confirm Payment” 4. After payment is confirmed, customer clicks “Okay”
Special Requirement: • Cart is not empty
Success Post-Conditions: • Payment is successful • Customer receives a ticket confirmation in their account • The database is updated as the ticket number being sold
Failure Post-Condition: • Payment is unsuccessful • The user is shown an error message that the Payment is unsuccessful • The user is taken back to the payment page
Extensions: • Contact customer support

STD16: View cart (Table 22)
Use Case Name: UC#16
Description: This scenario describes the user viewing the contents in their cart.
Primary Actors: • Customer
Pre-condition: • Cart must exist • Items added to cart must be inside the cart.
Trigger Event: • View Cart
Main Scenario Steps: 1. The user selects view cart 2. Cart contents are displayed to user 3. The use case ends when the items added to the cart are shown, or if no items are shown the cart appears empty.
Success Post-Conditions: • Items added to the cart are viewed inside the cart.
Failure Post-Condition: • Items added to the cart are not viewed inside the cart.
Extensions: None

Use Case Diagram (UCD)



Sequence Diagrams

Create an Account and Log-in

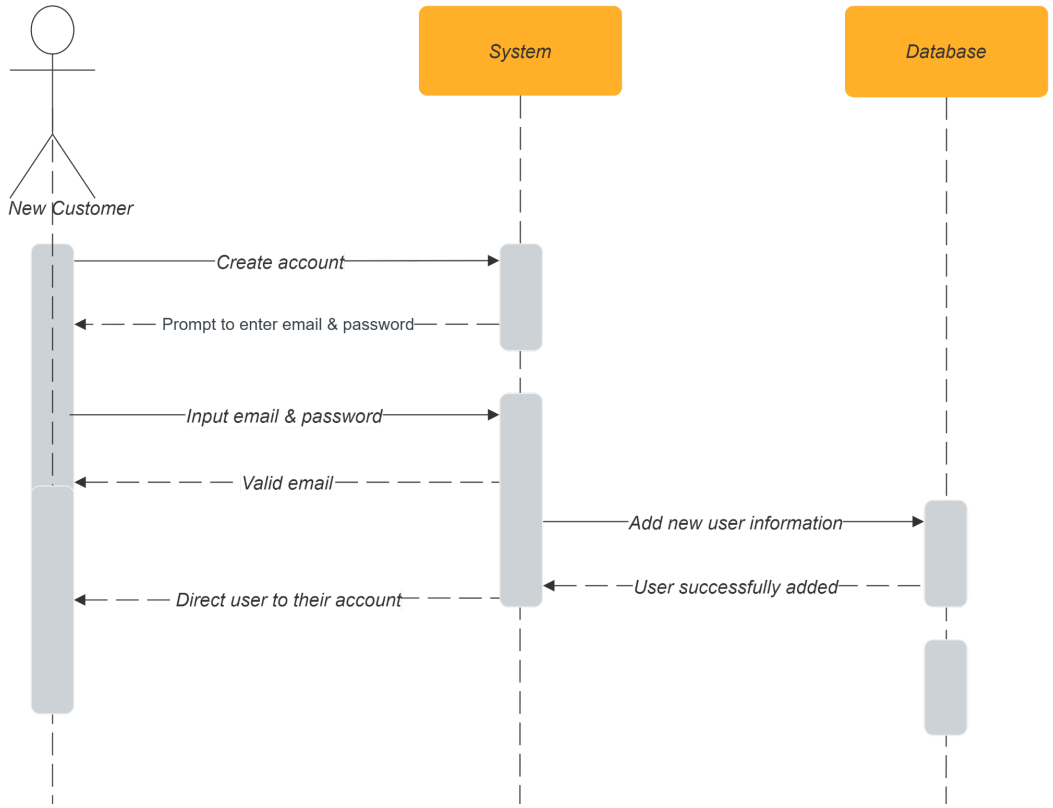


Figure 4

In this sequence diagram, a new customer wants to create an account on our site. The system validates the customer's email and adds the user to the database and redirects them to our site. The system ensures that the customer does not have an admin domain when creating an account.

Admin adds events

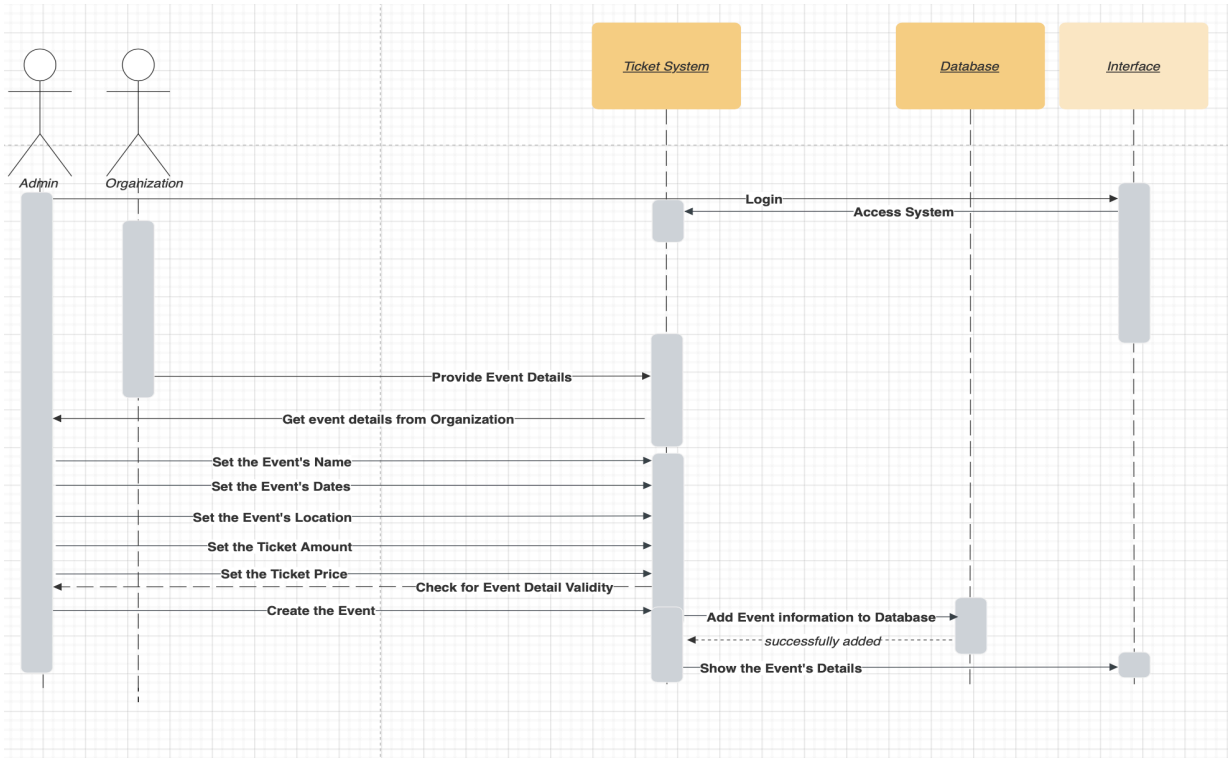


Figure 5

It is assumed that the login was successful. It is also assumed that event details are correct and valid (formatting included).
This sequence diagram shows the process of adding an event to the system and displays the event. It includes two actors, the admin and the organization. Both users log in, gaining access to the ticket system. The organization proposes an event and inputs its proposed event's details to the system. The admin retrieves these details from the system and begins setting event details such as name, dates, location, ticket count, and price. All the required information is checked for validity, and an event is created and stored in the database. The event becomes available and is shown.

Admin removes events

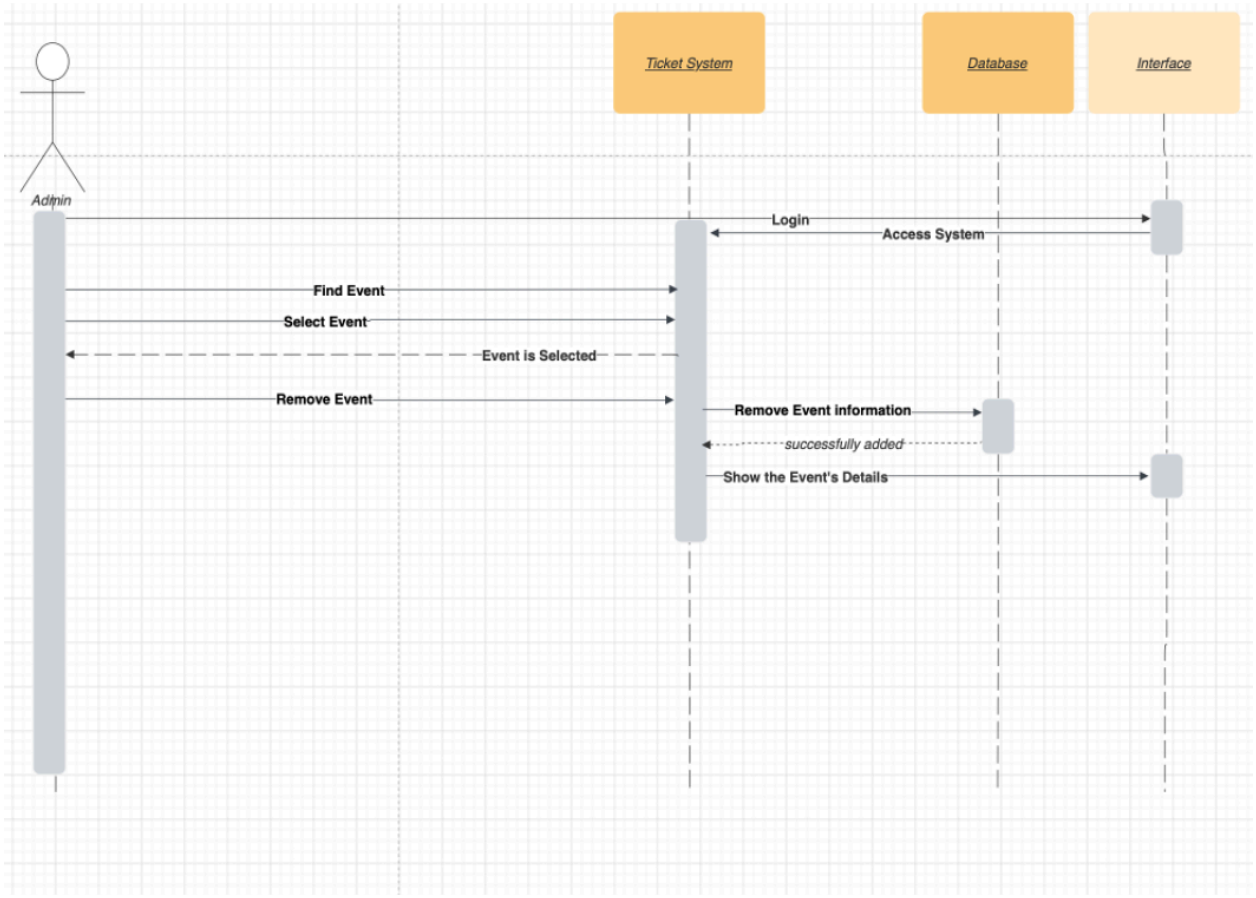


Figure 6

This sequence diagram starts with having the admin login. The admin then searches for an event and resets if it cannot be found. If successful, the admin selects to remove the event and is asked to confirm. The admin can also cancel the process and verify if it has been deleted.

Purchasing ticket

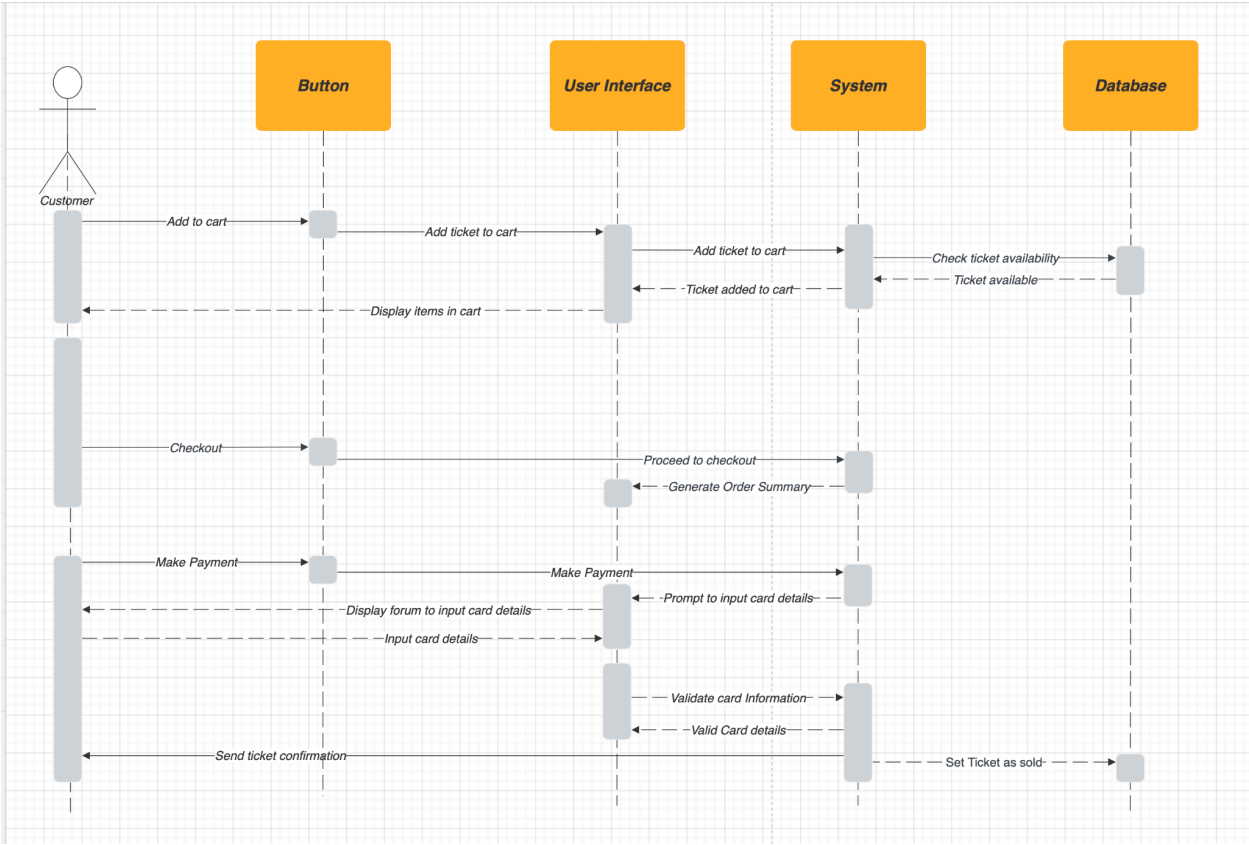


Figure 7

In this sequence diagram, a customer adds a ticket to the cart. The system checks for ticket availability and redirects them to the cart page. The user reviews the order summary and proceeds to checkout. Then, the user enters their card details and places their order. In the backend, the system verifies the card details to ensure the card is valid. Then, a ticket confirmation is sent, and that ticket is marked as sold.

Customer views their order history

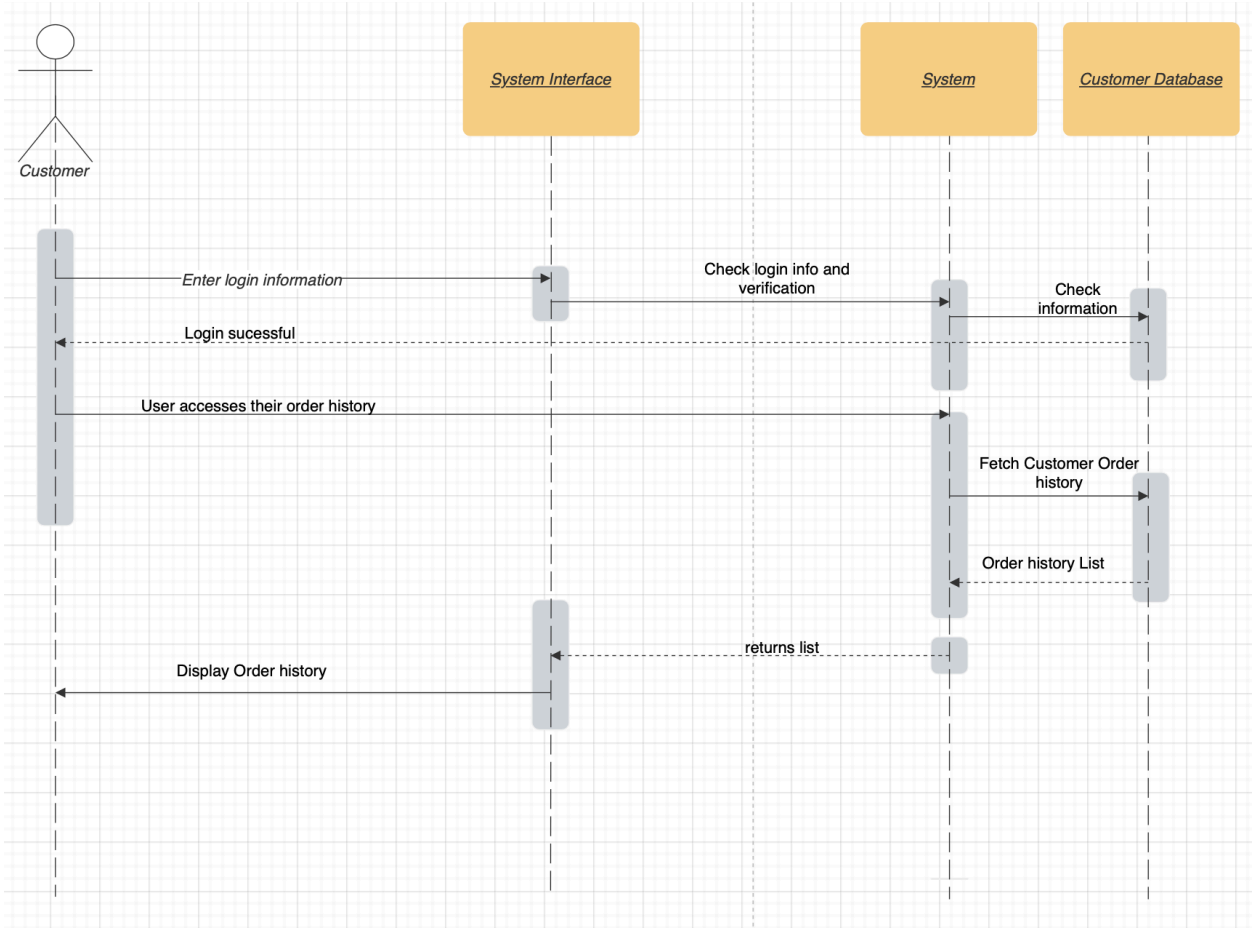


Figure 8

This sequence diagram depicts a customer viewing their order history and the interaction between the interface and the system. After logging in, the customer selects "Order history," which allows the system to retrieve the information from the database. From the interface, the user can view their current and past orders. Overall, this sequence diagram clearly represents the events that take place in our system for the customer to view their order history.
This sequence diagram assumes that the user exists and that the login and verification are successful without error.

Cancelling ticket

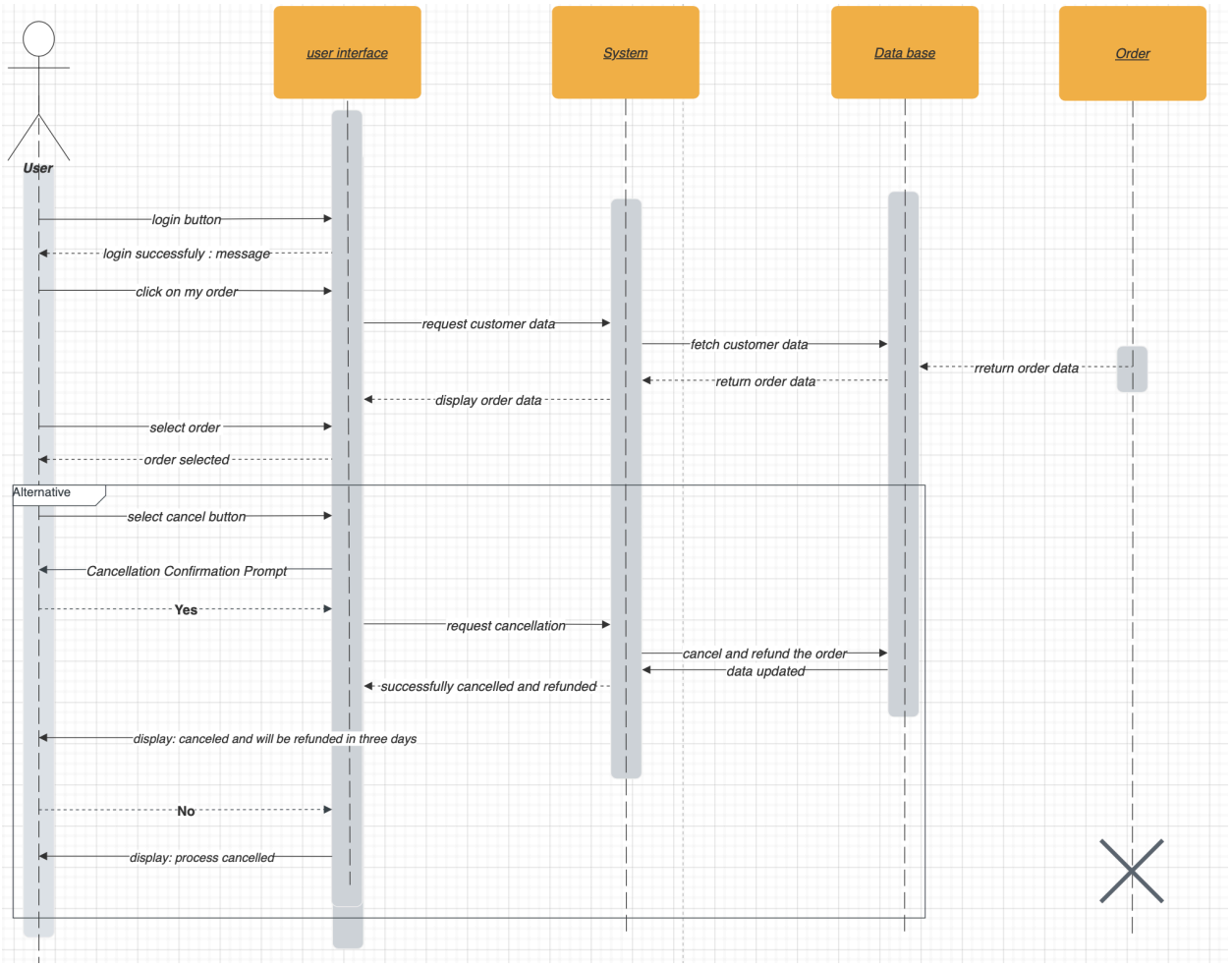


Figure 9

The user first logs in to their account and then clicks on the My ticket button, which takes them to their ticket information page (assuming they have one or more tickets purchased). The user initiates the cancellation process by clicking the "Cancel Ticket" button on the My Ticket page. It will prompt a cancellation confirmation to the user, and in this step, if they choose no, the process will be terminated; however, if the answer is yes, the system retrieves the ticket details to be cancelled, including the ticket number and the amount paid. The system calculates the refund amount based on the cancellation policy and deducts any cancellation charges, if applicable. The system processes the refund and sends a notification to the user indicating that the cancellation has been processed. The user receives the refund, and the ticket is marked as cancelled in the system.

Organization Login

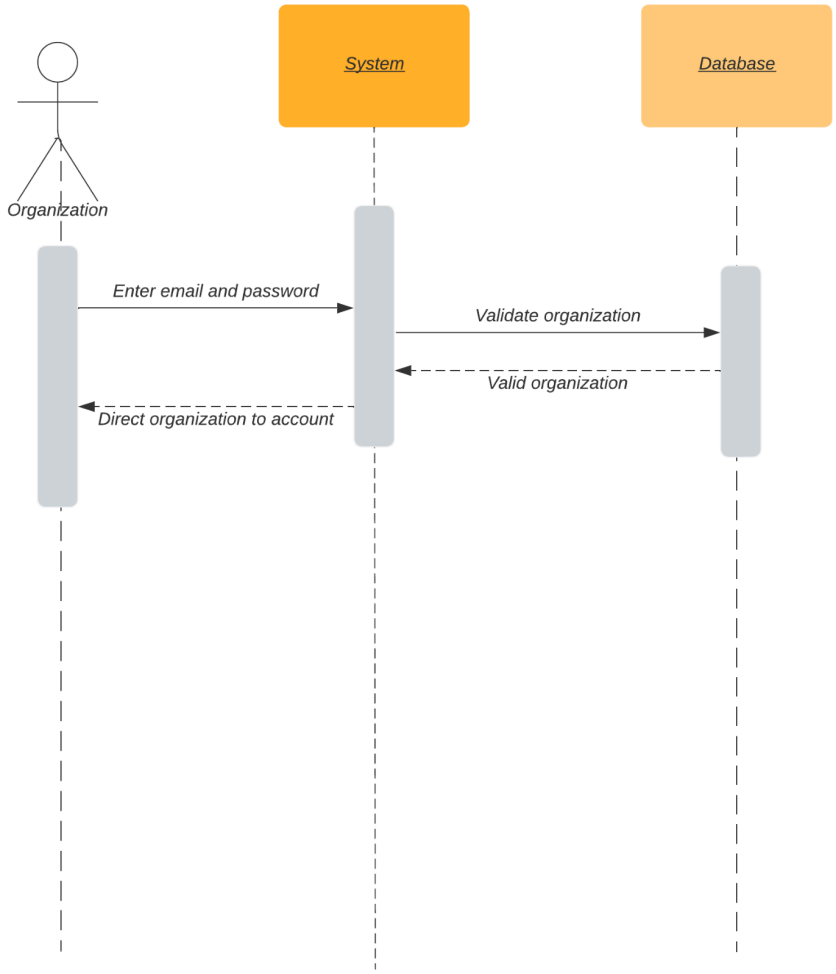


Figure 10

In this sequence diagram, an existing organization inputs their email and password. The account credentials are validated with the database. Once the information is valid, the organization is redirected to their account.

State Chart Diagrams

Create an account and Log in

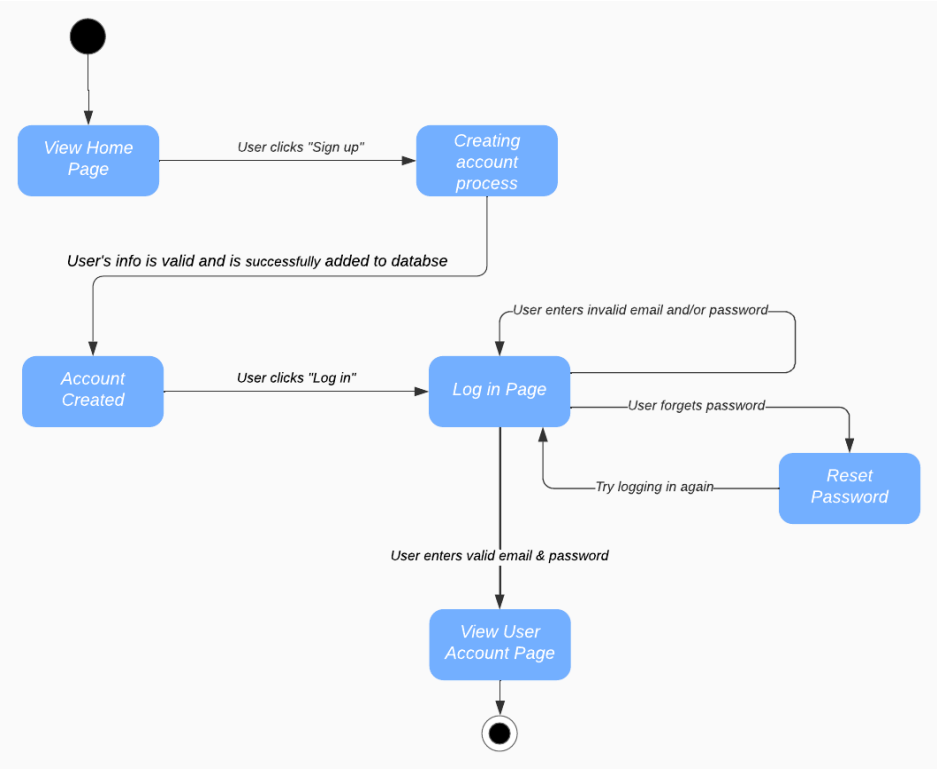


Figure 11

In this state diagram, a new user is joining the system. Initially, the user is not in the system database, so they click the "Sign up" button on the home page, which brings them to a forum page where they enter the required information to make an account, i.e. email and password. After submitting the information, the system validates the email address and adds the user to the database, which takes the user to the "Account created" state. To access the account, the user has to log in with the credentials they used to sign up for the account on the "Log in Page." Until the user enters the correct login information, the user will not be able to access their account and will display an error message. If the user forgets their password, they will enter the "Reset password" state, where they can enter their email, and a resetting password link will be sent to their email. If the email and password are valid, the user transitions to the next and final state, i.e. their account.

Add event to system

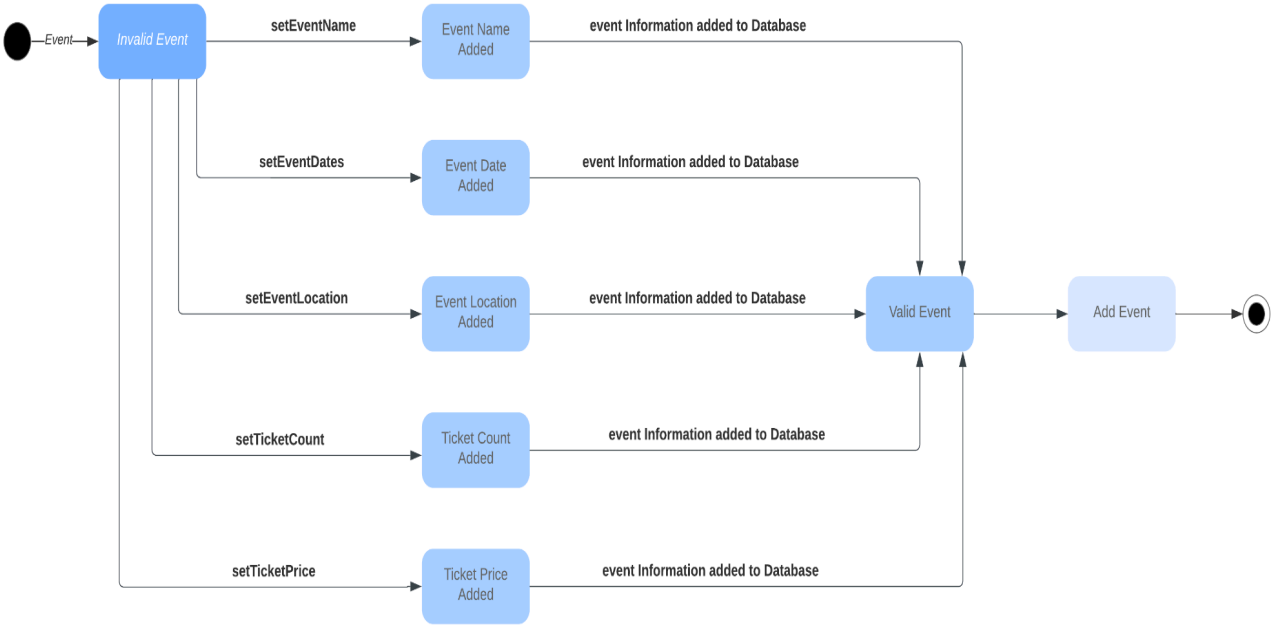
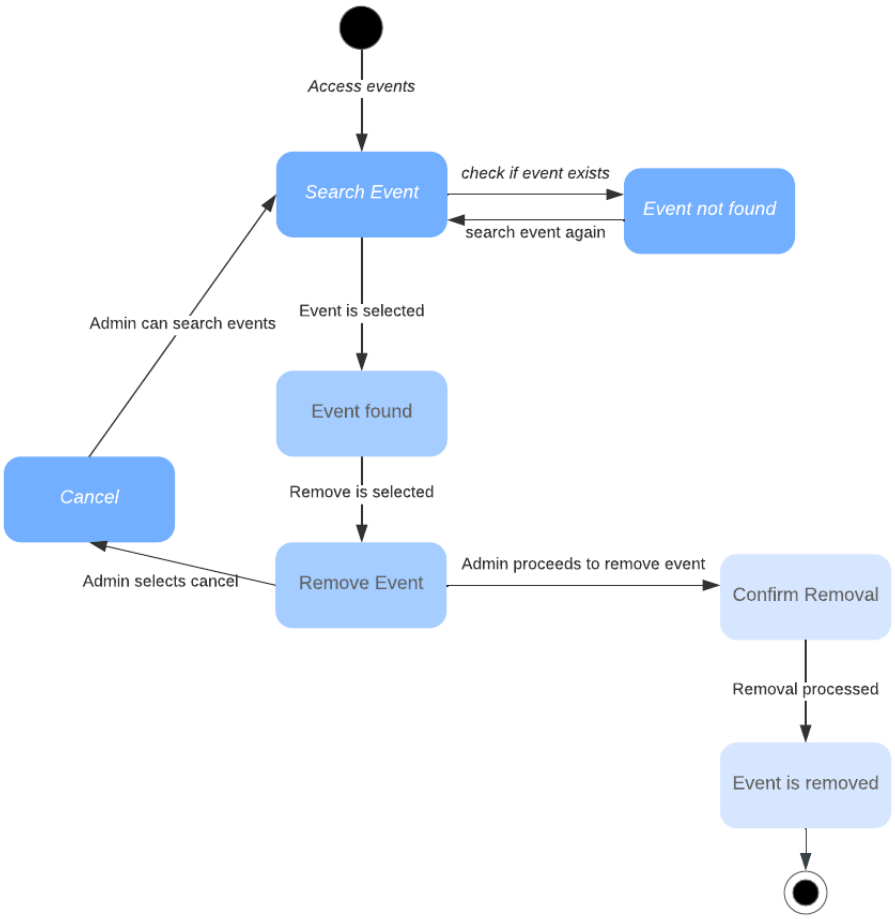


Figure 12

This state chart diagram shows the flow of states while adding an event. The event's initial state is considered an "invalid event" and goes through different states of adding information. The states include, Event Name Added, Event Date Added, Event Location Added, Ticket Count Added, and Ticket Price Added. During the "Event Name Added" state, to arrive at that state, the proposed event's name must be added to the system. Similarly, once the event date, location, total number of tickets, and price are set, the event is validated for all required information. Once all the information is needed, the event is added, and the final state is reached.

Remove Event



F

Figure 13

This state diagram shows the different states when removing an event. The admin starts with the login, then searches, removes the event, and confirms. Along the way, the program loops back if there are errors or requests to return to a state.

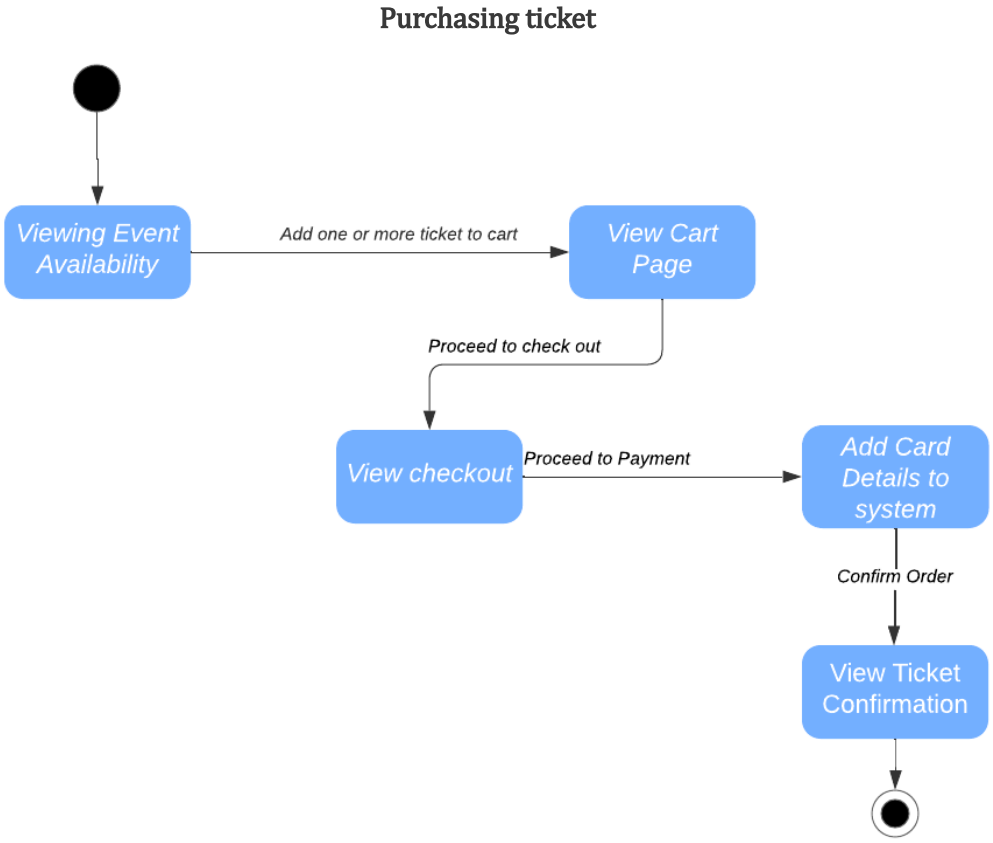


Figure 14

The ticket purchasing process starts with viewing event availability. After selecting a ticket, it will be added to the cart. The user can then click on "My cart" to view the tickets in the cart. Once the user is satisfied, the user will proceed to checkout. This is where they will be able to view the summary of their order and ticket prices. From there, they can proceed to checkout in the "Payment" state, and the user can enter their payment details. Once their order is confirmed, the ticket confirmation is sent to their account and a message of successful order placement is displayed.

Customer views Order history and details

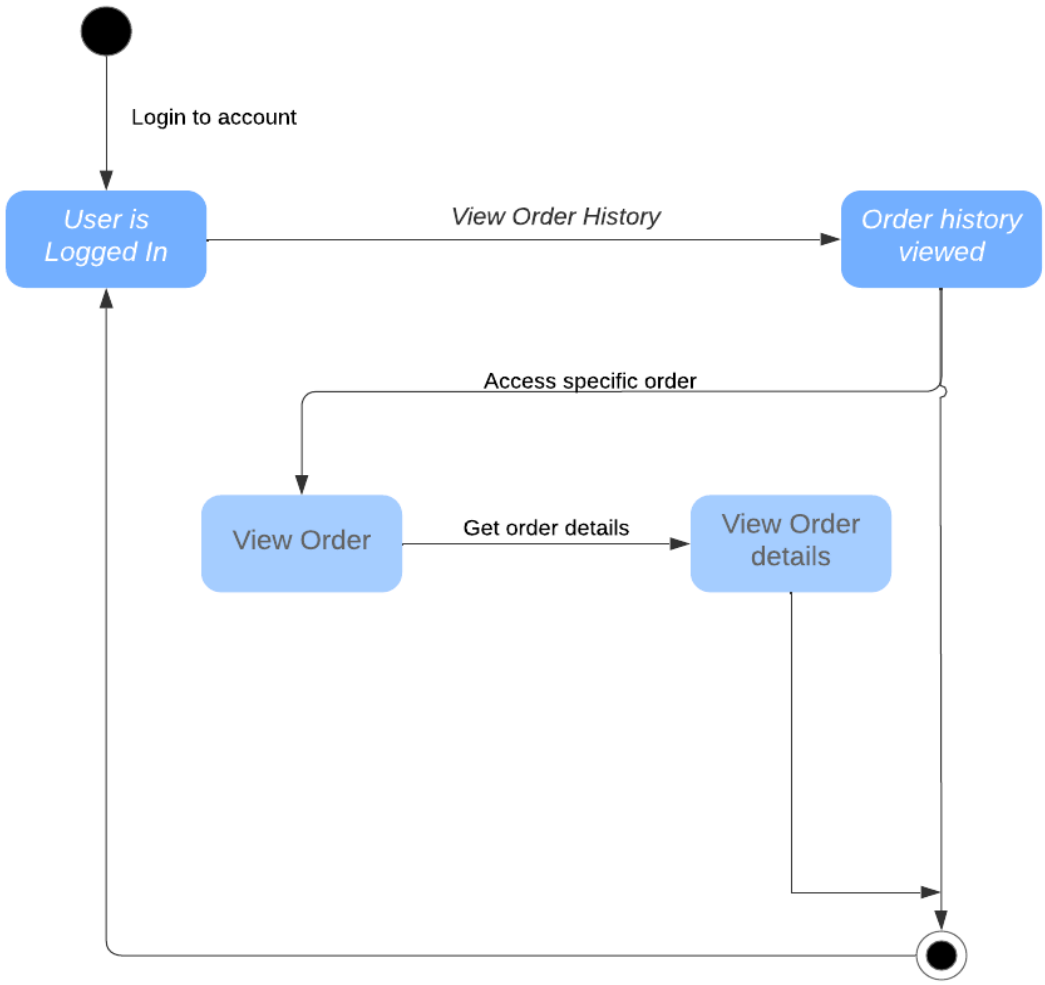


Figure 15

This state diagram models how customers view their order history and details. The order history state is when the customer views their current and past orders, assuming they have purchased tickets. From there, the customer can access their ticket or order details for a specific event.
This state diagram clearly and simply represents the customer's actions to view their order history, allowing them to navigate through their tickets and other information when necessary.

Cancelling tickets

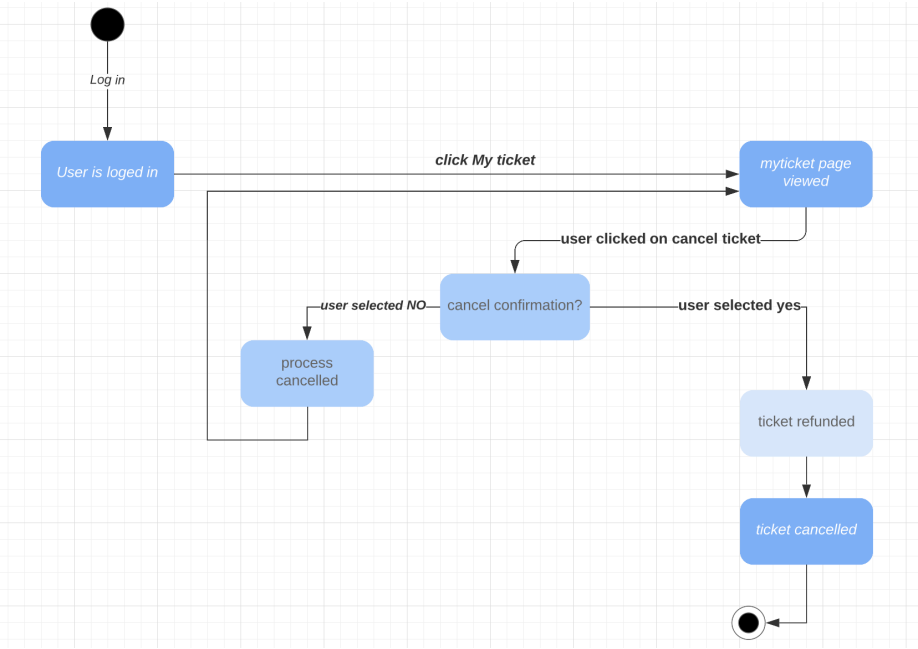


Figure 16

This is the system's starting state when the user is logged in. The ticket order history is viewed when the user has clicked on my tickets button. This state of Cancel Requested is entered when the user clicks on the "Cancel Ticket" button on the website, and It will ask for confirmation of the cancellation; the user has to choose yes or no. This state of Refund Processing is entered when the system processes the refund and sends a notification to the user indicating that the cancellation has been processed. The final state is that the ticket is cancelled, where the user receives the refund and the ticket is marked as cancelled in the system.

Organization Login

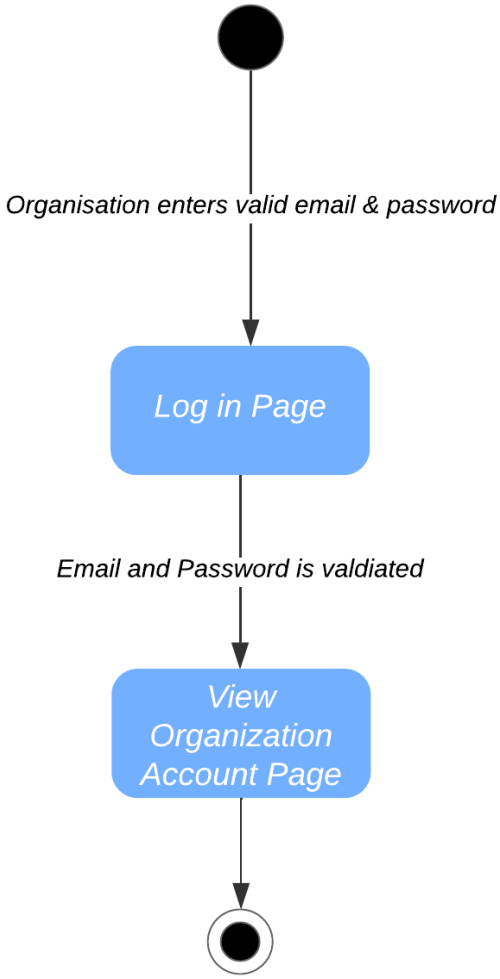


Figure 17

The initial state is that an organization is not logged in. The organization then enters a valid email and password. The information is validated, and the organization is redirected to their account page. The organization is then logged into the system, and the final state is reached.

Class Diagram

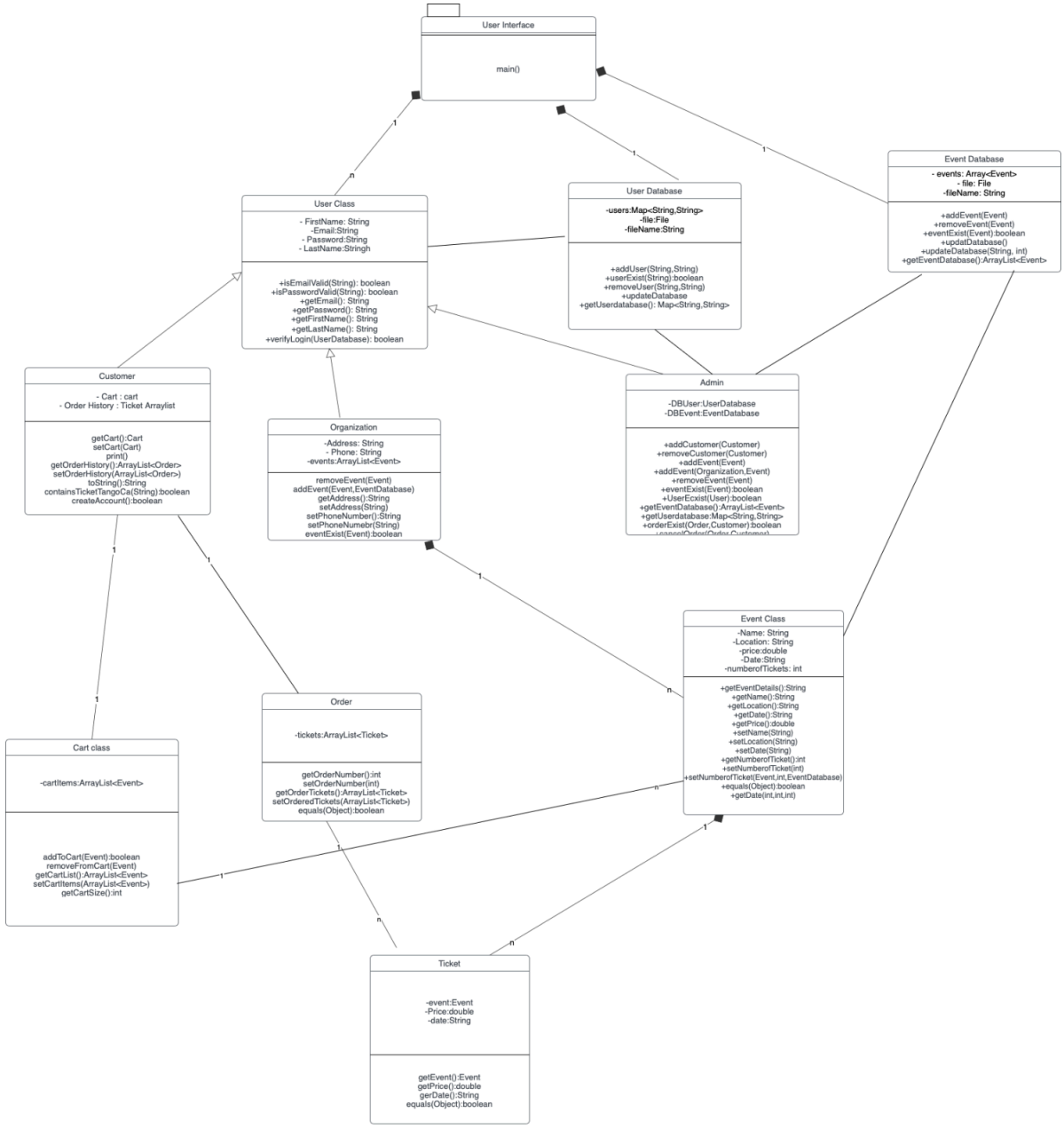


Figure 18

Our class diagram shows how our system will interact with the classes and their components. Our main System classes are our actors; Customer, Organization and Admin. The customer class works with the cart, ticket and order class. Each instance of an organization would contain the events that will happen. Lastly, every customer will have a cart from which they add, remove, or purchase event tickets.

Activity Diagram

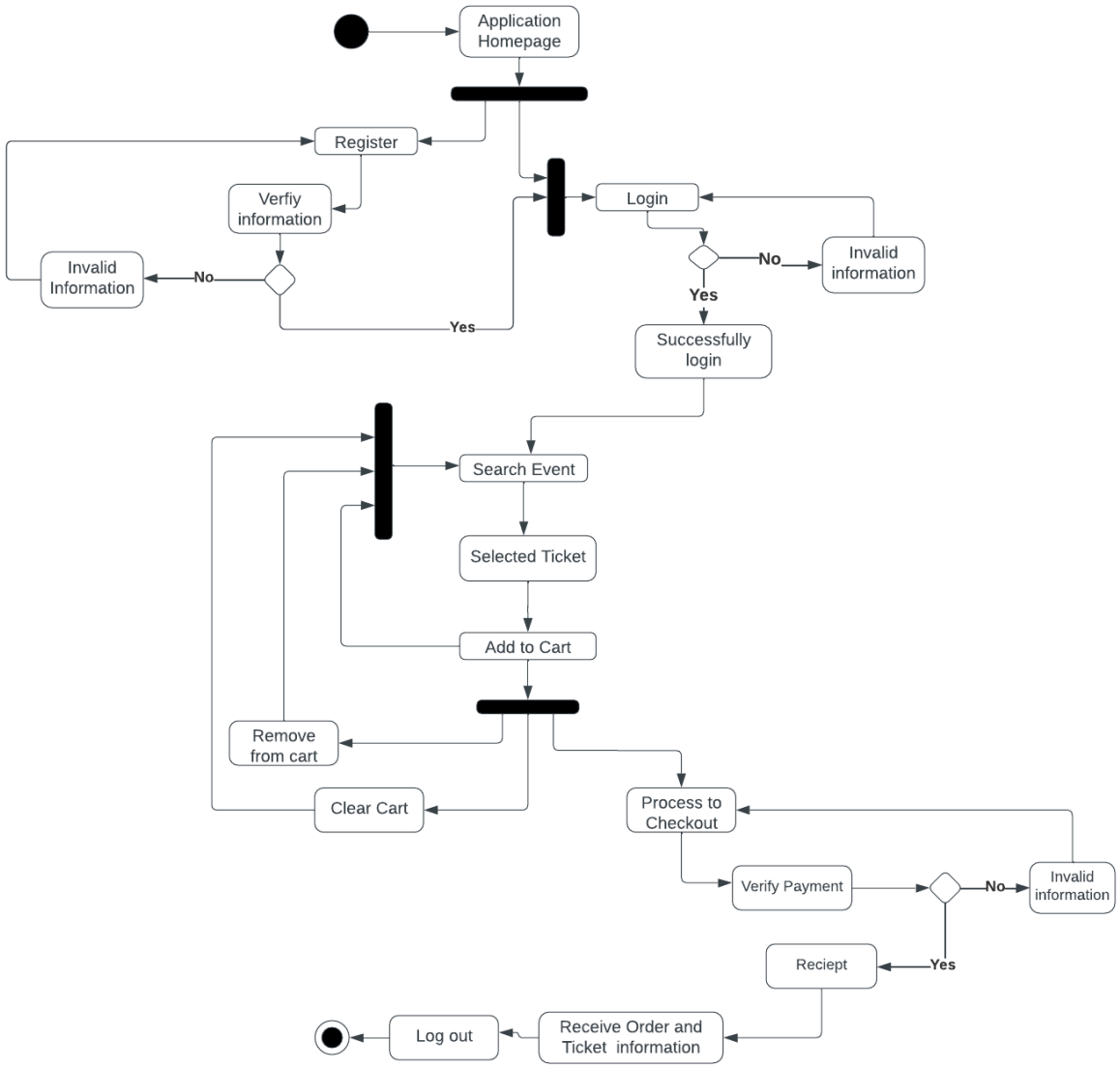


Figure 19

This activity diagram illustrates a typical workflow of a user purchasing a ticket on the application. The design aims to ensure a simple, efficient, and effective tool for users, with various alternative paths depending on the validity of the users' input. In addition, decision points on the login page (for registration and login) and the cart (for browsing events, removing items, clearing the cart, or proceeding to checkout) offer flexibility and options in the user journey. The steps involved in purchasing a ticket for an event start from selecting the event and adding it to the cart to generating the ticket(s) after payment is processed. Overall, the flow of the activity diagram is simplistic and straightforward, accurately displaying the flow of events while prioritizing user experience and usability.

Pattern Design

Pattern description for ticket purchasing system using the Gang of Four styles as shown in Table 23:

Table 23: Pattern Design description

Context:	The online ticket-purchasing application enables users to purchase tickets for events. The system is in charge of maintaining the accurate number and updating an event's ticket availability.
Problem:	The ticket system should guarantee the consistency and reliability of ticket sales by tracking ticket sales and preventing the overselling of tickets.
Force:	• The system needs to maintain and update ticket inventory in real-time accurately • The system needs to account for ticket purchases and cancellation • The system needs to prevent overselling or underselling of tickets.
Solution:	The behavioural patterns observer would be used to solve the problem. An event is an object that is designed to maintain the inventory of the ticket availability for the event. When a customer purchases a ticket, the event object is notified and updates the system interface with the correct number of available tickets. Likewise, if a customer cancels a purchased ticket, the event object is declared and updates the system interface. This allows users to view the event availability in real time, ensuring that event tickets are sold consistently and reliably.
Antipatterns:	Not using any pattern at all, can lead to inconsistent and unreliable ticket sales and inventory management.
Reference:	Lecture Notes: Topic 6 Lecture Note 1- Design Pattern.

System Implementation

Description of Implementation

Java is a high-level, object-oriented programming language that allows us to create simple, reliable applications that can run on different types of hardware and operating systems without requiring any additional modifications. Java is known for developing secure and robust applications, making it a good choice for creating a ticket purchasing system. Refer to Table 1.1 to see all the packages that we implemented. In addition, we created two packages called Database and System.We use the IDE intellij to code our system and used the codeshare function to collaboratively create our application.

Furthermore, Java Swing, the graphical user interface (GUI) toolkit for Java, provides key components such as buttons, text fields, and dialogs. These components allow users to input information and interact with our system. Customers can view events, add to their cart, enter payment information, and view their purchasing history on our application. All of these features contribute to the system's objective of creating a user-friendly, efficient, interactive system for individuals to purchase tickets.

Test cases were developed using JUnit4 to provide a virtual environment to test and run written code. Components such as Test, and Assert were used to compare expected and actual outputs of our program. Developers can run the *Tests* folder or run individual tests for the program. Individual test cases were developed per class and the methods contained in the class. Using JUnit allows for system developers to create individual test cases and witness pass or failed test cases.

Table 24: Java Packages in Ticket Purchasing System

Packages	Description
java.io	Provides classes for input and output operations, including reading and writing to file..
java.util	Provides a collection of useful utility classes and data structures, including ArrayLists, HashMaps, Scanners, and more.
System	Provides access to system class and functions, such as Event, Ticket, User, etc.
java.time	Provides classes for working with dates, times, and durations, such as LocalDate
Database	Provides access to database classes, such as UserDatabase and EventDatabase.

java.util.regex	Provides classes and interfaces for matching patterns in strings, including Matcher and Pattern.
java.awt	Provides classes for creating graphical user interfaces (GUI), including components like buttons, labels, and text fields.
javax.swing	Provides a set of GUI components that are more advanced than AWT, including JTable and JFrame.
JUnit 4	Provides methods used for unit testing. Test cases were developed using components such as Test and Assert to compare expected and actual output.

Description of Challenges and Solution

Table 25: Challenges faced during the project and its solutions

Problem	Solution
Designing and implementing a database that supports continuous updates and provides seamless accessibility to multiple classes/objects	To tackle this problem, we opted for a text file to store data, which we would read from at the start of the program and update whenever there were changes.
The occurrence of null pointer errors in different parts of the GUI, when a customer did not select or provide an input.	To address this issue, we implemented a check within the GUI code to ensure that the input values provided by the user were valid. Specifically, we made sure that the user could not proceed if an input was not provided or was invalid and even coded restrict text field input to alphanumeric characters or numbers, thereby preventing the occurrence of null pointer errors.
Converting string to numeric values	To address this issue, we utilized the trim() method to eliminate any leading or trailing white spaces that could potentially trigger processing errors during string-to-numeric parsing.
Difficulties connecting the GUI to Java classes and ensuring the synchronization of object sharing and modification between them.	To solve this problem, we ensured that the User classes contained all essential data. This allowed for easy object sharing between different GUI classes, enabling convenient modification and accurate information retrieval. In addition, we included an equals method to ensure accurate object comparison

Code Sample

Customer Class

```
package System;
import java.util.ArrayList;
/**
 * This Java class called "Customer" represents a customer object that inherits
properties of a user,
 * such as login details. The class includes methods to set and get a customer's
shipping address,
 * as well as a cart and order history.
 * The Customer class is designed to be utilized to manage orders for events or
tickets.
 * This class is able to store and manage
 * customer information, such as their cart, and order history. Additionally ,using
an ArrayList data structure,
 * The class can easily manage multiple orders within a customer's order history.
 *
 * @author   TicketTango
 * @version 1.0
 * @since   2023-04-02
 */

//The Customer class inherits from user and will be the main actor of our system,
//they will have their own cart and order history and login details.
public class Customer extends User{
  private Cart Cart;
  private ArrayList<Order> OrderHistory; // change to arraylist of orders

    public Customer(){
        this.Cart = new Cart();
    }
  /**
   * Constructor for setting shipping address.
   * @param email The email of the customer.
   * @param password The password of the customer.
   * @param FirstName The first name of the customer.
   * @param LastName The last name of the customer.
   */
  public Customer(String email, String password, String FirstName, String
LastName){
    super(email,password, FirstName, LastName);
    this.Cart = new Cart();
    this.OrderHistory= new ArrayList<>();
  }
```

```
/**
 * Constructor for a general customer.
 * @param email The email of the customer.
 * @param password The password of the customer.
 */
  public Customer(String email, String password){
      super(email,password);
      this.Cart = new Cart();
      this.OrderHistory= new ArrayList<>();
  }
/**
 * Getters for cart.
 * @return Cart The Cart object to be retrieved.
 */
public Cart getCart()
{
  return Cart;
}
/**
 * Setter for cart.
 * @param c The Cart object that is being set.
 */
public void setCart(Cart c)
{
  this.Cart = c;
}
/**
 * Test to print Customer Name to terminal to ensure creation.
 */
public void print()
{
  System.out.printf("\nName: %-20s ", getFirstName());
}
/**
 * Retrieves the order history.
 * @return ArrayList Order An ArrayList of the order history.
 */
public  ArrayList<Order> getOrderHistory(){
  return this.OrderHistory;
}
/**
 * Sets the order history.
 * @param OrderHistory The new order history to be set.
 */
public  void setOrderHistory(ArrayList<Order> OrderHistory){
  this.OrderHistory = OrderHistory;
```

```
  }
  /**
   * Returns user email and password as String
   * @return String Email and Password are returned as a String.
   */
  public String toString() {
      return getEmail() + "," + getPassword();
  }
  /**
   * Checks if the given String contains @ticket.tango.ca
   * @param input The String that will be checked.
   * @return boolean True if String contains @ticket.tango.ca, false otherwise.
   */
  public boolean containsTicketTangoCa(String input) {
    return input.contains("@ticket.tango.ca");
  }
  /**
   * Creates a new account that does not have a Ticket Tango email.
   * @return boolean True if valid email/password and not Ticket Tango email, false
otherwise.
   */
  public boolean createAccount() {
    return isEmailValid(getEmail()) && isPasswordValid(getPassword()) &&
!containsTicketTangoCa(getEmail());
  }
}
```

EventDatabse Class

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;
import System.*;
/**
 * The EventDatabase class represents a database of events, stored in a
text file.
 * It allows users to add, remove, and update events in the database.
 *
 * @autor   TicketTango
 * @version 1.0
 * @since   2023-04-02
 */
```

```
public class EventDatabase {
  // instance variables
  private ArrayList<Event> events; // to store events
  private File file; // to reference file
  private String fileName= "EventDatabase.txt"; // name of file

  /**
   * Creates a new EventDatabase object.
   * Reads events from a file and stores them in an ArrayList 'Event'
   * @throws IOException if an error occurs while reading from the file.
   */
  public EventDatabase(){
    // initialize instance variables
    this.events = new ArrayList<Event>();
    this.file = new File(fileName);

    // read events from file
    try {
      Scanner scanner = new Scanner(file);
      while (scanner.hasNextLine()) {
        String line = scanner.nextLine();
        String[] parts = line.split(",");
        if (parts.length < 5) {
          System.out.println("Invalid input line: " + line);
          continue;
        }
        String name = parts[0].trim();
        String location = parts[1].trim();
        String date = parts[2].trim();
        String numberOfTicketsStr = parts[3].trim();
        int numberOfTickets = Integer.parseInt(numberOfTicketsStr);
        String priceStr = parts[4].trim();
        double price = Double.parseDouble(priceStr);
        Event event = new Event(name, location, date, numberOfTickets,
price);
        events.add(event);
      }
      scanner.close();
    } catch (IOException e) {
      System.out.println("An error occurred while reading from the file: "
+ e.getMessage());
    }
```

```
  }
  /**
   * Checks if event already exists.
   * Adds an event to the database if event does not already exist.
   *
   * @param event The event to be added.
   */
  public void addEvent(Event event) {
    // check if the event already exists in the database
    if (!eventExists(event)) {
      events.add(event);
    }
      // updates the database
      updateDatabase();
  }
  /**
   * Removes an event from the database.
   *
   * @param ev The Event object to be remove
   */
  public void removeEvent(Event ev) {
    for (Event event :events ) {
      if (event.equals(ev)) {

        events.remove(event);
        updateDatabase();
        return;
      }
    }
  }

  /**
   * Checks if an event exists in the database.
   *
   * @param event The Event object to be checked.
   * @return boolean True if  the event exists in database, false
otherwise.
   */
  public boolean eventExists(Event event) {
    for (Event e : events) {
      if (e.equals(event)) {
        return true;
```

```
      }
    }
    return false;
  }
  /**
   * Updates the database file.
   * Updates the number of tickets for each event.
   */
  public void updateDatabase(){
    try {
      BufferedWriter writer = new BufferedWriter(new FileWriter(fileName,
false));
      // loop through events and update the number of tickets for the
specified event
      for (Event event : events) {
          writer.write(event.getName() + ", " +
                  event.getLocation() + ", " +
                  event.getDate() + ", " +
                  event.getNumberofTickets() + ", " +
                  event.getPrice());

        writer.newLine();
      }
      writer.close();
      System.out.println("Events written to " + fileName + "
successfully.");
    } catch (IOException e) {
      System.out.println("Error writing events to file: " +
e.getMessage());
    }
  }
  /**
   * Update the number of tickets for a specific event in the database
   * @param ev The event to be updated.
   * @param num The new number of tickets available
   */
  public void updateDatabase(Event ev, int num) {
    try {
      BufferedWriter writer = new BufferedWriter(new FileWriter(fileName,
false));
      // loop through events and update the number of tickets for the
specified event
```

```
      for (Event event : events) {
        if(ev.getName().equals(event.getName())){
            event.setNumberofTickets(num);
          writer.write(event.getName() + ", " +
                  event.getLocation() + ", " +
                  event.getDate() + ", " +
                  num + ", " + //Change ticket available
                  event.getPrice());
        } else {
          writer.write(event.getName() + ", " +
                  event.getLocation() + ", " +
                  event.getDate() + ", " +
                  event.getNumberofTickets() + ", " +
                  event.getPrice());
        }
        writer.newLine();
      }
      writer.close();
      System.out.println("Events written to " + fileName + "
successfully.");
    } catch (IOException e) {
      System.out.println("Error writing events to file: " +
e.getMessage());
    }
  }
  /**
   * Gets the list of events in the database
   * @return events, An ArrayList of Events.
   */
  public ArrayList<Event> getEventDatabase() {
    return events;
  }
}
```

Customer Class

```
package System;
import java.util.ArrayList;
/**
 * This Java class called "Customer" represents a customer object that
inherits properties of a user,
 * such as login details. The class includes methods to set and get a
```

```
customer's shipping address,
 * as well as a cart and order history.
 *
 * The Customer class is designed to be utilized to manage orders for
events or tickets.
 * With its efficient implementation, the class is able to store and manage
 * customer information, such as their shipping address, cart, and order
history. Additionally ,using an ArrayList data structure,
 * the class can easily manage multiple orders within a customer's order
history.
 *
 * @author   TicketTango
 * @version 1.0
 * @since   2023-04-02
 */

//The Customer class inherits from user and will be the main actor of our
system,
//they will have their own cart and order history and login details.
public class Customer extends User{

  private Cart Cart;

  private ArrayList<Order> OrderHistory; // change to arraylist of orders

    public Customer(){
        this.Cart = new Cart();
    }
  /**
   * Constructor for setting shipping address.
   *
   * @param email The email of the customer.
   * @param password The password of the customer.
   * @param FirstName The first name of the customer.
   * @param LastName The last name of the customer.
   */
  public Customer(String email, String password, String FirstName, String
LastName){

    super(email,password, FirstName, LastName);
    this.Cart = new Cart();
    this.OrderHistory= new ArrayList<>();
```

```
}

/**
 * Constructor for a general customer.
 *
 * @param email The email of the customer.
 * @param password The password of the customer.
 */
  public Customer(String email, String password){

      super(email,password);
      this.Cart = new Cart();
      this.OrderHistory= new ArrayList<>();
  }
/**
 * Getters for cart.
 *
 * @return Cart The Cart object to be retrieved.
 */
public Cart getCart()
{
  return Cart;
}
/**
 * Setter for cart.
 *
 * @param c The Cart object that is being set.
 */
public void setCart(Cart c)
{
  this.Cart = c;
}
/**
 * Test to print Customer Name to terminal to ensure creation.
 *
 */
public void print()
{
  System.out.printf("\nName: %-20s ", getFirstName());
}

/**
```

```
    * Retrieves the order history.
    *
    * @return ArrayList Order An ArrayList of the order history.
    */
   public  ArrayList<Order> getOrderHistory(){
     return this.OrderHistory;
   }
   /**
    * Sets the order history.
    *
    * @param OrderHistory The new order history to be set.
    */
   public  void setOrderHistory(ArrayList<Order> OrderHistory){
     this.OrderHistory = OrderHistory;
   }
   /**
    * Returns user email and password as String
    * @return String Email and Password are returned as a String.
    */
   public String toString() {
       return getEmail() + "," + getPassword();
   }
   /**
    * Checks if the given String contains @ticket.tango.ca
    * @param input The String that will be checked.
    * @return boolean True if String contains @ticket.tango.ca, false
otherwise.
    */
   public boolean containsTicketTangoCa(String input) {
     return input.contains("@ticket.tango.ca");
   }
   /**
    * Creates a new account that does not have a Ticket Tango email.
    * @return boolean True if valid email/password and not Ticket Tango
email, false otherwise.
    */
   public boolean createAccount() {
     return isEmailValid(getEmail()) && isPasswordValid(getPassword()) &&
!containsTicketTangoCa(getEmail());
   }

}
```

System Testing

Documentation and Unit Test Case Samples

Note: view zip folder submission for more Unit test Cases

AdminTest

```
public class AdminTest {
    Admin a1 = new Admin("roxie@ticket.tango.ca", "admin41");
    Customer c1 = new Customer("rr7972@gmail.com", "roller002");
    Event e1 = new Event("Rihanna", "New York", "2024-12-12", 1000, 200);

    /**
     * Checks if customer is added successfully from the database by the admin.
     * @result The Customer's email and password is added to the User Database
     */
    @Test
    public void addCustomer() {
        a1.addCustomer(c1);
        User user= c1;
        assertTrue(a1.userExist(user));
    }
}
```

CartTest

```
public class CartTest {
    /**
     * Checks if the event is successfully added or removed from the customer's
cart.
     * @result The customer's cart size should be updated after adding or removing
events.
     */
    @Test
    public void getCartSize() {
        Customer c1 = new Customer();
        assertTrue(c1.getCart().getCartList().isEmpty());
        Event e1 = new Event("BTS", "Toronto", "2023-07-07", 200, 250);
        Cart cartC1 = c1.getCart();
        cartC1.addToCart(e1);
        assertEquals(1, c1.getCart().getCartSize());
        assertNotEquals(0, c1.getCart().getCartSize());
        cartC1.removeFromCart(e1);
```

```
        c1.setCart(cartC1);
        assertEquals(0, c1.getCart().getCartSize());
    }
}
```

CustomerTest

```
public class CustomerTest {
    Customer cust1 = new Customer("cats@cats.ca", "rizz101");
    Cart cust1Cart = cust1.getCart();
    Event addEvent1 = new Event("BTS", "Toronto", "2023-07-07", 500, 250);
    Event addEvent2 = new Event("BTS", "Toronto", "2023-07-07", 500, 250);
    /**
     * Checks if customer's cart is successfully updated after adding events to
their cart.
     * @result The customer's cart size should be updated to have both events in
their cart.
     */
    @Test
    public void getCart() {
        cust1Cart.addToCart(addEvent1);
        cust1Cart.addToCart(addEvent2);
        cust1.setCart(cust1Cart);
        assertEquals(2,cust1.getCart().getCartSize());
    }
    /**
     * Checks if the order is successfully added to the customer's order history.
     * @result The customer's order history should have one order after adding the
order to their order history
     */
    @Test
    public void getOrderHistory() {
        ArrayList<Order> oh = new ArrayList<>();
        ArrayList<Ticket> c1Tickets = new ArrayList<>();
        Ticket ticket1 = new Ticket(addEvent1, addEvent1.getPrice(), "2023-04-02");
        Ticket ticket2 = new Ticket(addEvent2, addEvent2.getPrice(), "2023-04-02");
        c1Tickets.add(ticket1);
        c1Tickets.add(ticket2);
        Order orderC1 = new Order(101,c1Tickets);
        oh.add(orderC1);
        cust1.setOrderHistory(oh);
        assertEquals(1, cust1.getOrderHistory().size());
    }
}
```

AdminTest

```
public class EventTest {
    Event testEvent = new Event("Event Name", "Event Location","2024-01-15", 900,
160.56);
    Event testEvent2 = new Event("Test Event", "Location", "2016-03-16",
500,120.29);
    /**
     * Checks Event details and displays its parameters.
     * @result The event's Name, Event location, date, number of tickets, and price
     *
     */
    @Test
    public void getEventDetails() {
        assertEquals("Name; Event Name Location: Event Location Date: 2024-01-15",
"Name; " + testEvent.getName() +
                " Location: " + testEvent.getLocation() + " Date: " +
testEvent.getDate());
        assertEquals("Name; Test Event Location: Location Date: 2016-03-16", "Name;
" + testEvent2.getName() +
                " Location: " + testEvent2.getLocation() + " Date: " +
testEvent2.getDate());
    }
    /**
     * Checks the events name
     * @result The event's name is equal to the name initialized
     */
    @Test
    public void getName() {
        assertEquals(testEvent.getName(), "Event Name");
        assertNotEquals("Taylor Swift", testEvent.getName());
        assertEquals(testEvent2.getName(), "Test Event");
        assertNotEquals("Event Name", testEvent2.getName());
    }
}
```

OrderTest

```
public class OrderTest {
    ArrayList<Ticket> tickets = new ArrayList<>();
    Order testOrder = new Order(123, tickets);
    Order testOrder2 = new Order(134, tickets);
    /**
     * Gets the order's order number.
     * @result Same order number as testOrder initialization.
     * **/
    @Test
```

```
    public void getOrderNumber() {
        assertEquals(testOrder.getOrderNumber(),123);
        assertNotEquals(testOrder.getOrderNumber(),321);
    }
    /**
     * Sets the order's order number.
     * @result initialized from method.Order number is set to new order number by
method.
     * **/
    @Test
    public void setOrderNumber() {
        assertEquals(testOrder.getOrderNumber(),123);
        testOrder.setOrderNumber(321);
        assertEquals(testOrder.getOrderNumber(),321);
    }

}
```

OrganizationTest

```
public class OrganizationTest{
    ArrayList<Event> eventTester = new ArrayList<>();
    Organization testOrg = new Organization("test@email.com", "Password", "123
Address Rd.",
            "416-555-5555", eventTester);
    Event testEvent = new Event("Justin Bieber", "AIR CANADA CENTRE", "2025-02-02",
60, 60.0);
    /**
     * View the arraylist containing an organization's events.
     * @result When an event is added, the list size is increased. Event is added
and gotten from the arraylist. 0->1
     * **/
    @Test
    public void viewEvents() {
        assertEquals(testOrg.viewEvents().size(), 0);
        testOrg.viewEvents().add(testEvent);
        assertEquals(testOrg.viewEvents().size(), 1);
        assertEquals(testOrg.viewEvents().get(0),testEvent);
    }
    /**
     * Events added are removed from the arraylist of events.
     * @result Event size is decreased after an event is removed. 1->0
     * **/
    @Test
    public void removeEvent() {
        testOrg.viewEvents().add(testEvent);
```

```
        assertEquals(testOrg.viewEvents().size(),1);
        testOrg.removeEvent(testEvent);
        assertEquals(testOrg.viewEvents().size(),0);
    }
}
```

TicketTest

```
public class TicketTest {
    Event addEvent1 = new Event("BTS", "Toronto", "2023-07-07", 1000, 250);
    Ticket t1 = new Ticket(addEvent1, addEvent1.getPrice(), "2023-04-02");
    /**
     * Checks if the event object from ticket equals the event.
     * @result The ticket's event equals the event object
     */
    @Test
    public void getEvent() {
        assertEquals(addEvent1, t1.getEvent());
    }
    /**
     * Checks if the ticket's event price is the same as the original ticket
object's price
     * @result The ticket's event price equals the original ticket object's price
     */
    @Test
    public void getPrice() {
        assertEquals(addEvent1.getPrice(), t1.getPrice(), 0.00001);
    }
}
```

UserTest

```
public class UserTest {
    /**
     * Checks if the admin email equals the the expected result from the getEmail()
method.
     * @result The expected email and the admin's email are equal.
     */
    @Test
    public void getEmail() {
        Admin a1 = new Admin("roxie@ticket.tango.ca","admin101");
        assertEquals("roxie@ticket.tango.ca", a1.getEmail());
    }
    /**
     * Checks if the admin password equals the expected result from the
getPassword() method.
```

```
     * @result The expected password and the admin's password are equal
     */
    @Test
    public void getPassword() {
        Admin a1 = new Admin("roxie@ticket.tango.ca","admin101");
        assertEquals("admin101", a1.getPassword());
    }
}
```

Test case Documentation

Login Test

```
Amy@gmail.com,Sunflower
```

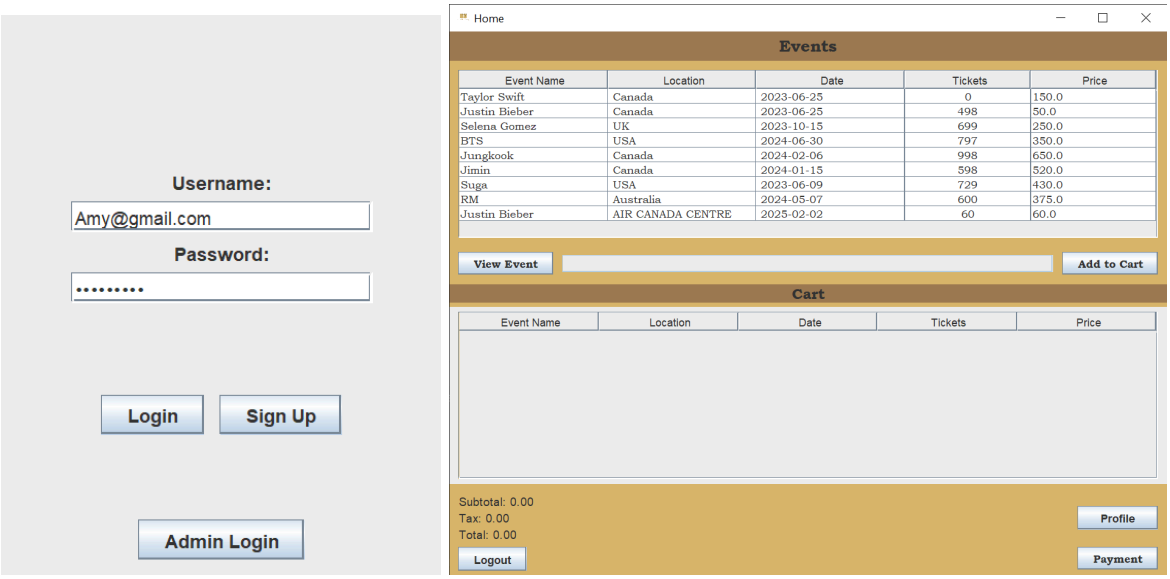
User email and password are stored in the database.

1. Enter your username and password in the corresponding fields. If you do not have an account, click the "Create Account" button to register.
2. Click the "Login" button to submit your credentials.
3. If your credentials are valid, you will be redirected to the events page where you can view your upcoming events.

***Note: If your credentials are invalid or incomplete, an error message will be displayed.

TestCase 1

Purpose: Validates Login and redirects users to the main page.
Expected Outcome: Valid Login -> Redirect to the Events page

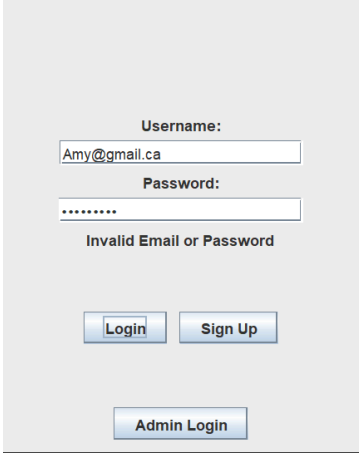


When the correct username and password are inputted, clicking the login button (left) will redirect the customer to the events page (right).

TestCase 2

Purpose: Check if the username is valid.

Expected Outcome: Invalid Username-> Error Message

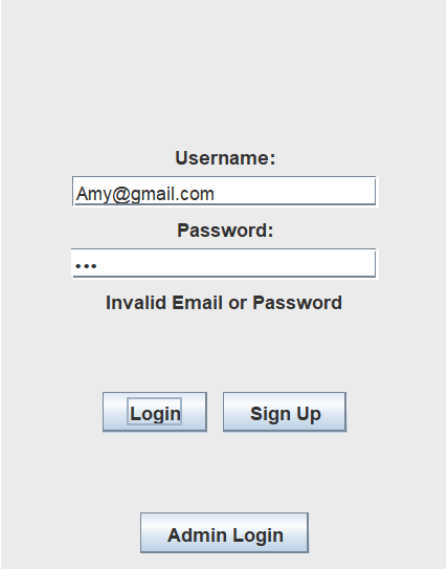


An incorrect email address is inputted for the username, the correct password is inputted. Results in an error message since the email address is incorrect with information stored on the database.

TestCase 3

Purpose: Check if the password is valid.

Expected Outcome: Invalid Password -> Error Message



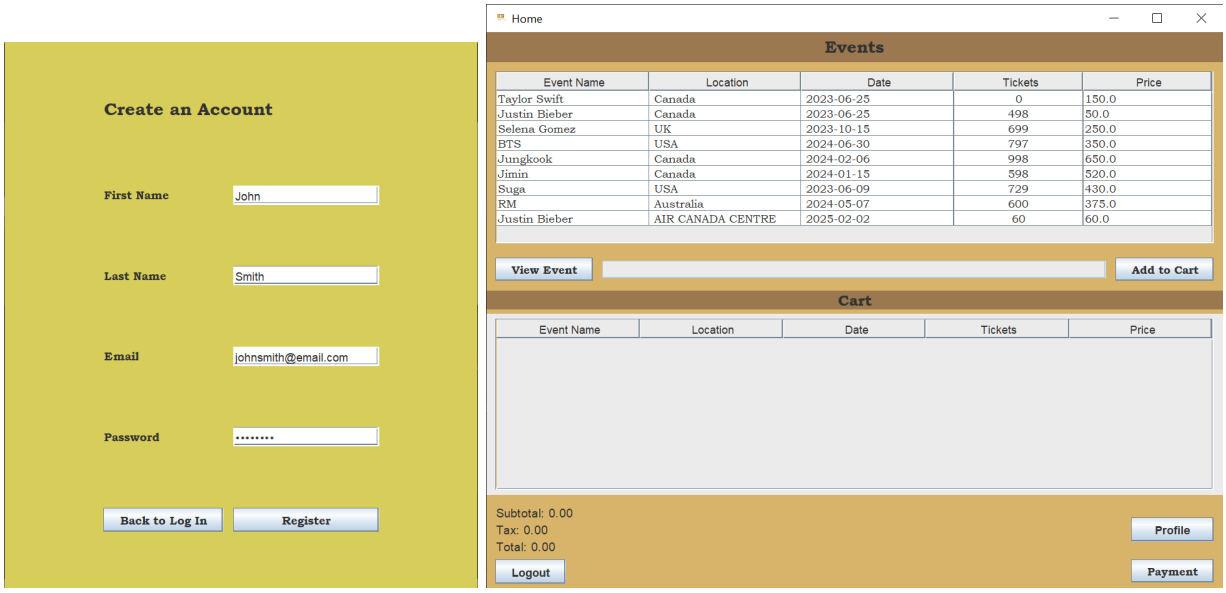
The correct email address is inputted for the username, an incorrect password is inputted. Results in an error since information does not match with information stored on the database.

Create an Account

TestCase 4

Purpose: Create a valid User account.

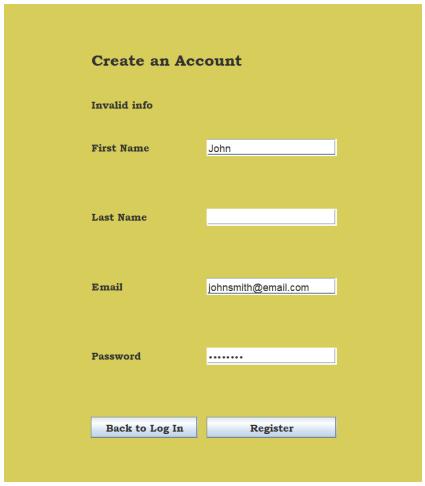
Expected Outcome: Valid Sign up -> Register -> Events Page (Check the database for updated information)



Valid information is inputted for First Name, Last Name, Email, and Password. Validity includes no numbers for first and last names, a minimum length of 2, the email must have a valid domain (include “@example.domain”), and the password must have a minimum length of 8. Once the Register button is clicked and the information is valid, the user is redirected to the events page.

TestCase 5

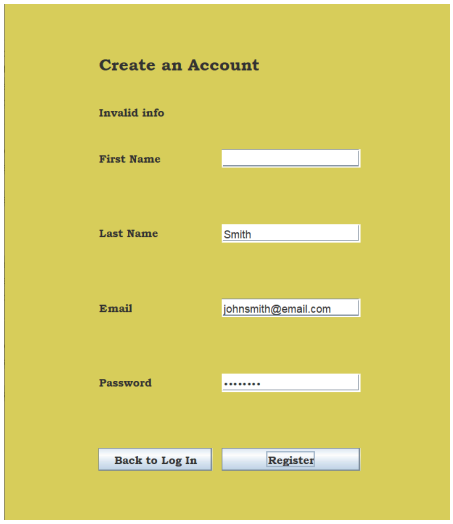
Purpose: Check if the last name is valid.
Expected Outcome: Invalid last name -> Error message



Valid information is inputted for First Name, Email, and Password. The Last Name field is empty and an error message, “Invalid info” is thrown.

TestCase 6

Purpose: Check if the first name is valid.
Expected Outcome: Invalid first name -> Error message

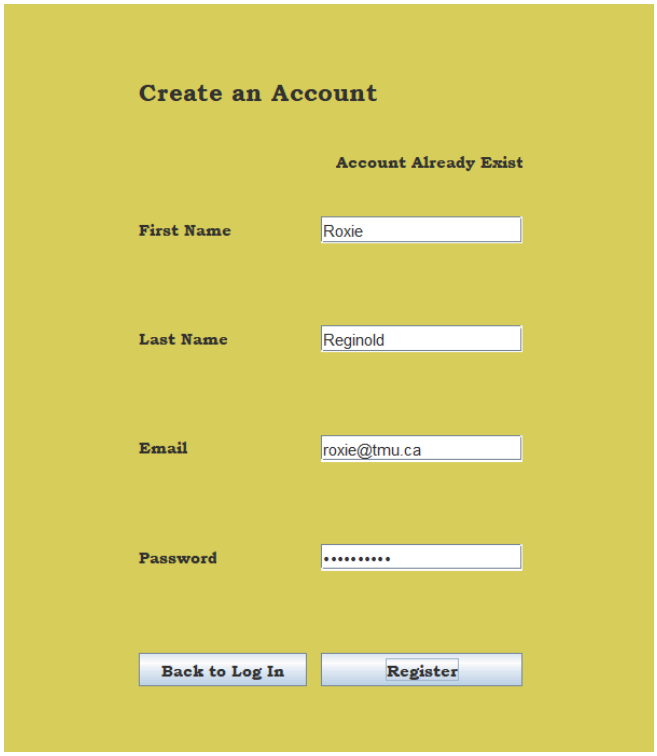


Valid information is inputted for Last Name, Email, and Password. The first name field is empty and an error message, “Invalid info” is thrown.

TestCase 7

Purpose: Check if email already exists.

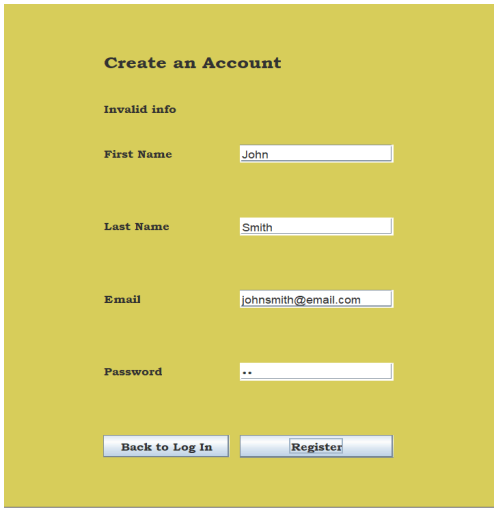
Expected Outcome: Registered email -> Error message: Account Already Exists



Valid information is inputted for First Name, Last Name, and Password. Email already exists in the database (registered user) and an error message, “Account Already Exists” is thrown.

TestCase 8

Purpose: Formatting: Check if the inputted password is valid.
Expected Outcome: Invalid password → Error message



Valid information is inputted for First Name, Last Name, and Email. Password does not satisfy password requirements of minimum length 8 and an error message, "Invalid info" is thrown.

Event Test

```
public class EventTest {
    Event testEvent = new Event( name: "Event Name",  location: "Event Location", date: "2024-01-15",  numberofTickets: 900,  price: 160.56);
    Event testEvent2 = new Event( name: "Test Event",  location: "Location",  date: "2016-03-16",  numberofTickets: 500, price: 120.29);
```

testEvent = A new Event object.
testEvent2 = A new Event Object.

```
@Test
public void getName() {
    assertEquals(testEvent.getName(),  actual: "Event Name");
    assertNotEquals( unexpected: "Taylor Swift", testEvent.getName());
    assertEquals(testEvent2.getName(),  actual: "Test Event");
    assertNotEquals( unexpected: "Event Name", testEvent2.getName());
}
```

TestCase 9

Purpose of this Unit: Checks if Function retrieves the right name for the event.
Expected Outcome: Same event name set during testEvent and testEvent2 creation.

```
@Test
public void setNumberofTickets() {
    Event ticketEvent = new Event(this.testEvent.getName(),this.testEvent.getLocation(),this.testEvent.getDate(),
            this.testEvent.getNumberofTickets(), this.testEvent.getPrice());
    assertEquals(ticketEvent.getNumberofTickets(), this.testEvent.getNumberofTickets());
    ticketEvent.setNumberofTickets(ticketEvent.getName(), numberofTickets: 200,new EventDatabase());
    assertNotEquals(ticketEvent.getNumberofTickets(),this.testEvent.getNumberofTickets());
}
```

TestCase 10

Purpose of this Unit: Checks if the function sets the right number of tickets for an event.
Expected Outcome: Same output for newly set number for testEvent.

OrderTest

```
public class OrderTest {
    ArrayList<Ticket> tickets = new ArrayList<>();
    Order testOrder = new Order( orderNumber: 123, tickets);
    Order testOrder2 = new Order( orderNumber: 134, tickets);
```

tickets=a new arraylist containing tickets used to hold an order.
testOrder=a new order created with order number 123 and arraylist tickets.
testOrder2=a new order created with order number 134 and arraylist tickets.

```
public void getOrderNumber() {
    assertEquals(testOrder.getOrderNumber(), actual: 123);
    assertNotEquals(testOrder.getOrderNumber(), actual: 321);
}
```

TestCase 19

Purpose of this Unit: Function retrieves order number set from initialization.
Expected Outcome: Same order number as initialized in testOrder creation.

```
@Test
public void getOrderedTickets() {
    Event testEvent = new Event( name: "Justin Bieber", location: "AIR CANADA CENTRE", date: "2025-02-02", numberofTickets: 60, price: 60.0);
    Ticket testTicket = new Ticket(testEvent, price: 190.00, date: "2025-04-29");
    assertEquals(testOrder.getOrderedTickets().size(), actual: 0);
    testOrder.getOrderedTickets().add(testTicket);
    assertEquals(testOrder.getOrderedTickets().size(), actual: 1);
}
```

testEvent = A new event.
testTicket = A new ticket for the event.

TestCase 22

Purpose of this Unit: Function retrieves ordered tickets.

Expected Outcome: Size of ordered tickets is increased by 1. Ticket is added to tickets arraylist.

```
@Test
public void setOrderedTickets() {
    ArrayList<Ticket> testTickets = new ArrayList<>();
    Event testEvent = new Event( name: "Justin Bieber",  location: "AIR CANADA CENTRE",  date: "2025-02-02",  numberofTickets: 60,  price: 60.0);
    Ticket testTicket = new Ticket(testEvent,  price: 190.00,  date: "2025-04-29");
    testTickets.add(testTicket);
    testOrder.setOrderedTickets(testTickets);
    assertEquals(testOrder.getOrderedTickets(),testTickets);
}
```

testTickets = A new arraylist initialized for an order.

TestCase 23

Purpose of this Unit: Function sets an arraylist of ordered tickets.

Expected Outcome: The ordered tickets arraylist's size of testOrder is the same as the newly set testTickets arraylist's size.

```
@Test
public void testEquals() {
    assertNotEquals(testOrder, testOrder2);
}
```

TestCase 24

Purpose of this Unit: Check if Two events with different details are different.

Expected Outcome: testOrder is not equal to testOrder2. Details are different, therefore different orders.

Organization Test

```
public class OrganizationTest{
    ArrayList<Event> eventTester = new ArrayList<Event>();
    Organization testOrg = new Organization( email: "test@email.com",  password: "Password",  address: "123 Address Rd.",
            phoneNumber: "416-555-5555", eventTester);
    Event testEvent = new Event( name: "Justin Bieber",  location: "AIR CANADA CENTRE",  date: "2025-02-02",  numberofTickets: 60,  price: 60.0);
```

eventTester = an ArrayList of events.

testOrg = A new organization object.

testEvent = A new event object.

```
@Test
public void viewEvents() {
    assertEquals(testOrg.viewEvents().size(), actual: 0);
    testOrg.viewEvents().add(testEvent);
    assertEquals(testOrg.viewEvents().size(), actual: 1);
    assertEquals(testOrg.viewEvents().get(0),testEvent);
}
```

TestCase 25

Purpose of this Unit: Arraylist contains newly added testEvent.
Expected Outcome: The size of eventTester ArrayList increases when an event is added. From 0 to 1. View added event from ArrayList.

```
@Test
public void setAddress() {
    assertEquals(testOrg.getAddress(), actual: "123 Address Rd.");
    testOrg.setAddress("1234 Address Rd.");
    assertEquals(testOrg.getAddress(), actual: "1234 Address Rd.");
    assertNotEquals(testOrg.getAddress(), actual: "123 Address Rd.");

}
```

TestCase 28

Purpose of this Unit: Function sets new address for the organization.
Expected Outcome: Address of testOrg is different from first initialization, after function call the address is newly set. 123 Address Rd. -> 1234 Address Rd.

```
@Test
public void eventExists() {
    assertEquals(testOrg.viewEvents().size(), actual: 0);
    assertFalse(testOrg.eventExists(testEvent));
    testOrg.viewEvents().add(testEvent);
    assertEquals(testOrg.viewEvents().size(), actual: 1);
    assertTrue(testOrg.eventExists(testEvent));

}
```

TestCase 31

Purpose of this Unit: The event added is contained in the organization's ArrayList of events.

Expected Outcome: An organization exists in the object's ArrayList, size of ArrayList is increased.

Admin Test

```
public class AdminTest {
    Admin a1 = new Admin( email: "roxie@ticket.tango.ca",  password: "admin41");
    Customer c1 = new Customer( email: "rr7972@gmail.com",  password: "roller002");
    Event e1 = new Event( name: "Rihanna",  location: "New York",  date: "2024-12-12",  numberofTickets: 1000,  price: 200);
```

a1 = a new admin object with an existing email and password.
c1= a new customer object.
e1 = a new event object.

```
@Test
public void addEvent() {
    a1.addEvent(e1);
    assertTrue(a1.eventExist(e1));
}
```

TestCase 34
Purpose of this Unit: The admin checks to ensure that the event that was added to the event database exists in our system.
Expected Outcome: The event that was added to the event database now exists in our system.

```
@Test
public void removeEvent() {
    a1.removeEvent(e1);
    assertFalse(a1.eventExist(e1));
}
```

TestCase 35
Purpose of this Unit: The admin checks to ensure that the event that was removed from the database no longer exists.
Expected Outcome: The event that was removed does not exist in the database.

```java
@Test
public void cancelOrder() {
    ArrayList<Order> oh = new ArrayList<>();
    ArrayList<Ticket> c1Tickets = new ArrayList<>();
    Ticket ticket1 = new Ticket(e1, e1.getPrice(), date: "2023-04-02");
    c1Tickets.add(ticket1);
    Order orderC1 = new Order( orderNumber: 101,c1Tickets);
    oh.add(orderC1);
    c1.setOrderHistory(oh);
    a1.cancelOrder(orderC1, c1);
    assertEquals( expected: 0, c1.getOrderHistory().size());

}
```

TestCase 36

Purpose of this Unit: The admin checks the customer's order history to see if the order still exists after cancelling the order.

Expected Outcome: The order was removed from the order history of the customer.

Cart Test

```
@Test
public void getCartSize() {
    Customer c1 = new Customer();
    assertTrue(c1.getCart().getCartList().isEmpty());
    Event e1 = new Event( name: "BTS",  location: "Toronto",  date: "2023-07-07",  numberofTickets: 200,  price: 250);
    Cart cartC1 = c1.getCart();
    cartC1.addToCart(e1);
    assertEquals( expected: 1, c1.getCart().getCartSize());
    assertNotEquals( unexpected: 0, c1.getCart().getCartSize());
    cartC1.removeFromCart(e1);
    c1.setCart(cartC1);
    assertEquals( expected: 0, c1.getCart().getCartSize());
}
```

TestCase 37

Purpose of this Unit: Checks if the event is successfully added or removed from the customer's cart.
Expected Outcome: The customer's cart size should be updated after adding or removing events.

Customer Test

```
public class CustomerTest {
    Customer cust1 = new Customer( email: "cats@cats.ca",  password: "rizz101");
    Cart cust1Cart = cust1.getCart();
    Event addEvent1 = new Event( name: "BTS",  location: "Toronto",  date: "2023-07-07",  numberofTickets: 500,  price: 250);
    Event addEvent2 = new Event( name: "BTS",  location: "Toronto",  date: "2023-07-07",  numberofTickets: 500,  price: 250);
```

cust1 = a customer object that already exists in our system.
cust1Cart = cust1's cart object.
addEvent1 and addEvent are Event objects that share the same event details.

TestCase 39

```
@Test
public void getCart() {

    cust1Cart.addToCart(addEvent1);
    cust1Cart.addToCart(addEvent2);
    cust1.setCart(cust1Cart);
    assertEquals( expected: 2,cust1.getCart().getCartSize());
}
```

Purpose of this Unit: Checks if a customer's cart is successfully updated after adding events to their cart.
Expected Outcome: The customer's cart size should be updated to have both events in their cart.

Ticket Test

```
public class TicketTest {
    Event addEvent1 = new Event( name: "BTS", location: "Toronto", date: "2023-07-07", numberofTickets: 1000, price: 250);
    Ticket t1 = new Ticket(addEvent1, addEvent1.getPrice(), date: "2023-04-02");
```

addEvent1 = an Event object.
T1 = a Ticket object for the addEvent1 object.

```
@Test
public void getEvent() {
    assertTrue(addEvent1.equals(t1.getEvent()));
}
```

TestCase 42
Purpose of this Unit: Checks if the event object from the ticket equals the event.
Expected Outcome: The ticket's event equals the event object.

```
@Test
public void getPrice() {
    assertEquals(addEvent1.getPrice(), t1.getPrice(), delta: 0.00001);
}
```

TestCase 43
Purpose of this Unit: Checks if the ticket's event price is the same as the original ticket object's price.
Expected Outcome: The ticket's event price equals the original ticket object's price.

User Manual

Manual Version 1.0

Introduction

For the latest software updates about this product, visit *TicketTango*.

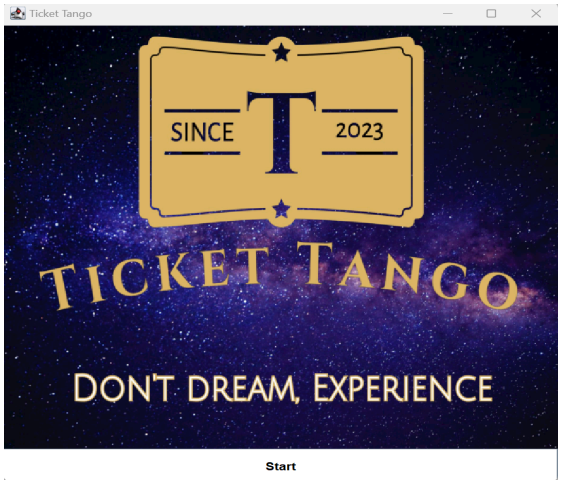
For additional software support, visit *TicketTango*.

Installation

To start *Ticket Tango* extract the zip folder containing the *TicketTango.jar,* (Oracle Java Runtime Environment required) and run the program.

Welcome Start

Start: Begin software.



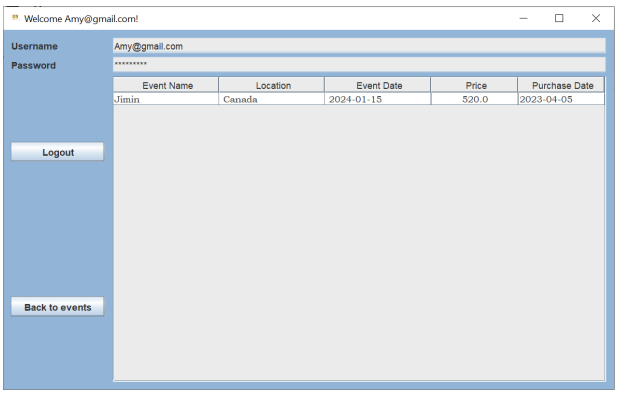
Account Info

Displays the Username and Password of the account.

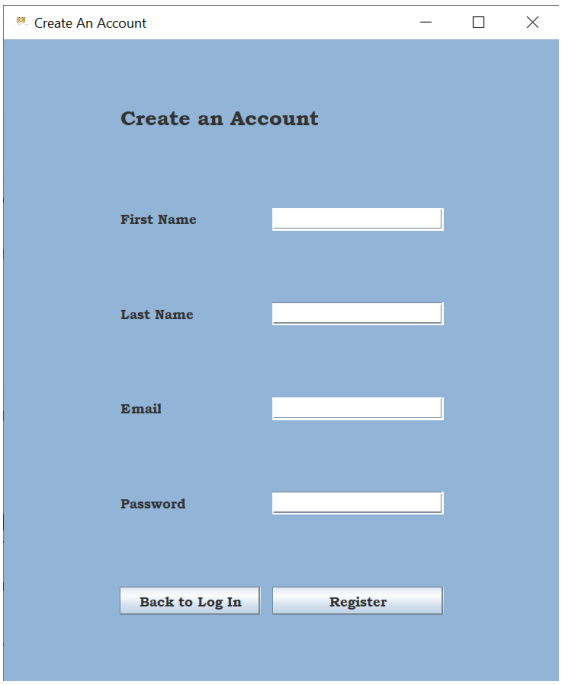
Displays ticket information for the event ticket purchased. *Event name*: name of the event, *Location*: location of the event, *Event Date*: date of the event, *Price*: price of the event, *Purchase date*: date the ticket was purchased.

Logout: Logs out of user session. Returns to **Login** upon activation.

Back to events: Returns to **Event Page**.



Create Account



First Name: First (given) name. Must be more than 1 letter long.

Last Name: Last (family) name. Must be more than 1 letter long.

Email: Email address used to log into the system.

Password: Password used to log into the system. Passwords must be at least 8 characters long.

Back to Log In: Returns to **Login**.

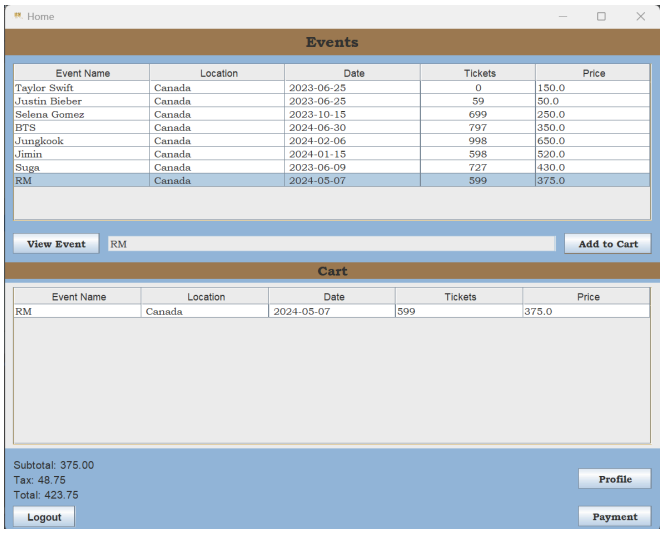
Register: Creates an account, and proceeds to **Event Page** if all credentials are met.

Event Info

Contains detailed information on the event. *Name*: name of the event, *Date*: date of the event, *Location*: location of the event, *Price*: price of the event, *Event Info*: information containing the amount of tickets left as well as additional event information.

Back to Events: Returns to **Event Page**.

Event Page



Events

Table containing ongoing events. *Event name*: name of the event, *Location*: location of the event, *Event Date*: date of the event, *Tickets*: number of tickets left for the event, *Price*: price of the event. To add an event ticket to cart, select the row containing the desired event and the event name will be selected. Select **Add to Cart** to add the event ticket to your cart. To view more information on the event, select **View Event**.

Cart

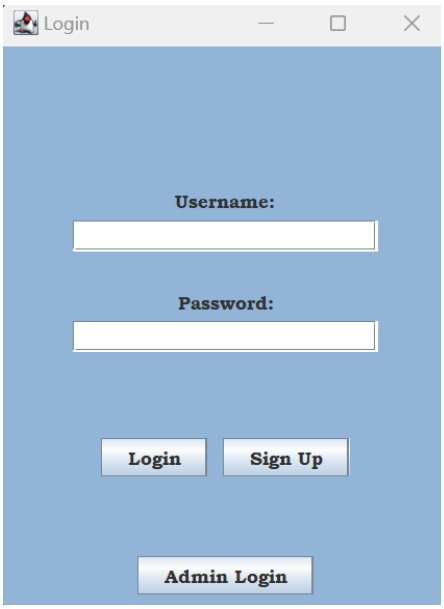
Your cart contains the event information for the ticket added. Items can be *removed* from cart by clicking on the desired ticket on the table. Estimated total is displayed on the bottom left page.

Profile: See **Account Info** for more information.

Logout: Logs out of user session. Returns to **Login** upon activation.

Payment: Proceeds to payment for items in cart. See **Payment** for more information.

Login



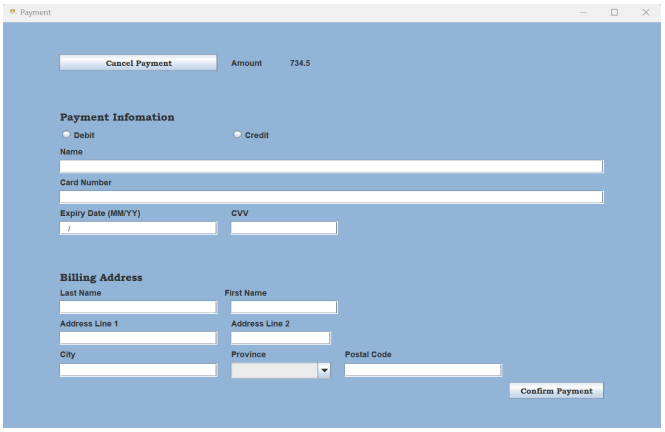
Logs into the system with customer account credentials.
Username: The email used when the account was created.
Password: The password used when the account was created.
Login: Log into the system using the Username and Password provided. Ensure that all details are accurate.
Sign Up: Create an account, for more information visit **Create Account**
Admin Login: Log in using administrator credentials.

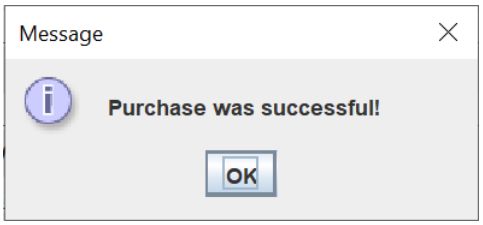
Payment

The total amount displayed is shown "*Amount <Price>*". Payment information includes *Debit* or *Credit*, *Name*, *Card Number*, *Expiry Date(MM/YY)*, *CVV*. Billing address information includes *Last Name*, *First Name*, *Address Line 1*, *Address Line 2*, *City*, *Province*, *Postal Code*.

Cancel Payment: Returns to **Event Page**.

Confirm Payment: Confirms payment field. Ensure all information is correct before proceeding.

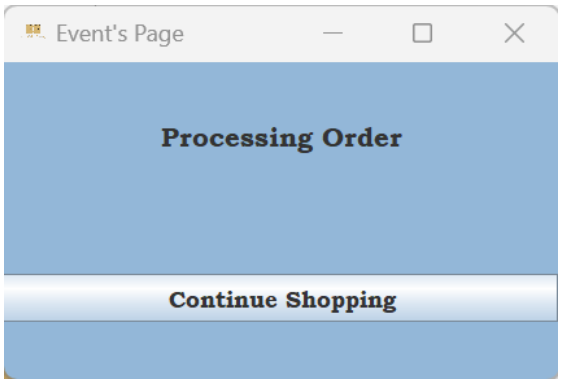




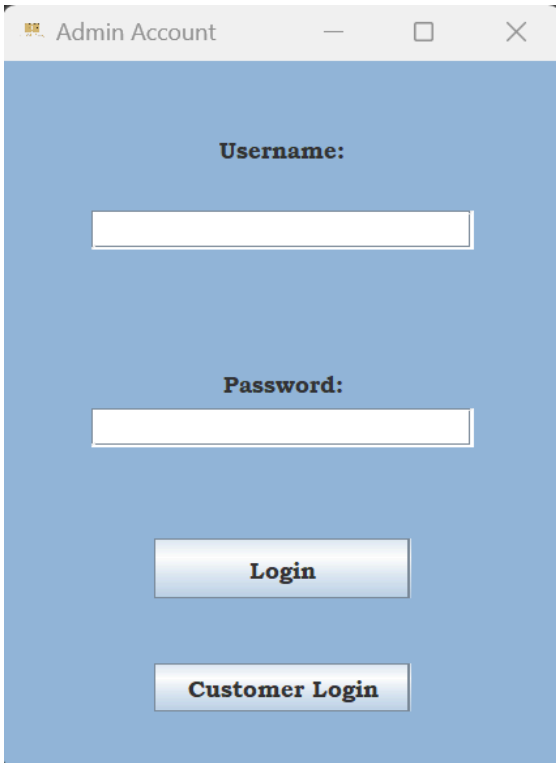
Processing

Processes the current order.

Continue Shopping: Returns to **Event Page.**



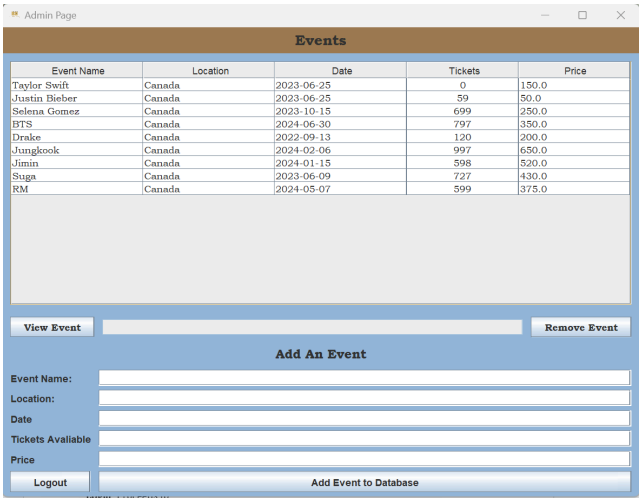
Admin Login



Logs into the system with admin account credentials.
Login: Accesses **Admin Page** using the admin Username and Password provided. Ensure that all details are accurate.

Customer Login: Return to **Login** page to continue using the program as a customer.

Admin Page



Displays information on the event with administrative permissions to add and remove an event.

View Event: See **Event Info** for more information.

Remove Event: Removes the selected event from the system.

Add An Event

Event Name: The name of the event.

Location: The location the event is held.

Date: The date of the event's occurrence.

Tickets Available: Number of tickets available for the event.

Price: The price of the event per ticket.

Logout: Proceeds back to **Login** page.

Add Event to Database: Event gets added to the database and can be displayed for users.

Technical Information

Recommended Specifications:	Java Development Toolkit (JDK) running at least Oracle Java version 17. At least 500MB of free hard drive space.

Trademarks and Licenses

Manual Version 1.0