OBJECT ORIENTED PROGRAMMING REPORT

INSTRUCTORS:

Dr. Mahmoud Khalil

Eng. Mazen Tarek

TEAM 17:

Eyad Ahmed Habib	24P0240
Karma Adnan Hussein Alsafadi	24P0287
Omar Mohamed Mahmoud	24P0270
Jasmin Ibrahim Fathy Ibrahim Aly	24P0225
Ahmed Mohamed Said Mahmoud Emara	24P0359
Yahia Ahmed Younis	24P0347

Table of contents

1 Back End

1.1 User

The user class is an abstract parent class that defines common attributes for all users of the system, while containing the getters and setters for those attributes. Since this class implements the HasID interface it makes sure that all inheritors of this class also implement the method of get ID which ensures all users have a valid ID. It also override toString and equals to abide with best practices of object oriented programming (to force all inheritor classes to implement those methods).

```
abstract public class User implements HasID {  14 usages  3 inheritors  kim28 *
    protected String email;  7 usages
    protected String username;
    protected String password;  5 usages
    protected DateTime dateOfBirth;  7 usages
    protected Gender gen;  6 usages
    protected String ID;

    protected User (){}

    public String getUsername() { return username; }
    public String getEmail() { return email; }
    public String getPassword() { return password; }
    public DateTime getDateOfBirth() { return dateOfBirth; }
    public Gender getGen() { return gen; }

    public void setEmail(String email) { this.email = email; }
    public void setUsername(String username) { this.username = username; }
    public void setDateOfBirth(DateTime dateOfBirth) { this.dateOfBirth = dateOfBirth; }
    public void setPassword(String password) { this.password = password; }
    public void setGen(Gender gen) { this.gen = gen; }

    public String getID() { return ID; }

    @Override  3 implementations  kim28
    abstract public String toString();

    @Override  3 overrides  kim28
    public boolean equals(Object obj) { return super.equals(obj); }
```

1.2 Admin

The Admin class is a subclass of abstract class User that represents administrative users in the system. Admins have the attributes of user in addition to role and working hours Admins are responsible for managing rooms, categories, and viewing lists of organizers and attendees.

```
private String role;  4 usages
private String workingHours;  4 usages
```

```
public Admin(){}  kim28
public Admin(String email, String username, String password,  kim28 +1
             DateTime dateOfBirth, Gender gen, String role, String workingHours) {
    this.email = email;
    this.username = username;
    this.password = password;
    this.dateOfBirth = dateOfBirth;
    this.gen = gen;
    this.ID = "a" + System.nanoTime();
    this.workingHours = workingHours;
    this.role = role;
}
```

The class contains both parametrized and no arg constructors(to abide by best practice for oop).

```
public String getRole() {return role;}  3 usages
public String getWorkingHours() { return workingHours; }
public String getID() { return ID; }
```

The getID method overrides the method from the implemented interface HasID.

```
public void addRooms(String roomName, int roomCapacity, double rentPrice){  no usages  Jasmin
    Room newroom= new Room(roomName, roomCapacity,rentPrice);
    Database.create(newroom);
}
public void addCat(String catName){  no usages  Jasmin
    Category newCat= new Category(catName);
    Database.create(newCat);
}
```

addRooms() and addCat() allow the admin to create new rooms and cateogries (which gives the admin create abilities for both of those classes)

```
public static ArrayList<Event> searchEvents(String EventName) { return Database.findEvent(EventName); }

public static ArrayList<Organizer> viewOrganizers(){...}

public static ArrayList<Attendee> viewAttendee(){...}
```

The 3 methods shown above, search the database to return array lists to aid the front end part of the program in displaying the values for the user

```
@Override
public String toString(){
    return "Admin{" +
            "email='" + this.email + '\'' +
            ", username='" + this.username + '\'' +
            ", role='" + getRole() + '\'' +
            ", workingHours='" + getWorkingHours() + '\'' +
            '}';
}

@Override
public boolean equals(Object o){
    if (o instanceof Admin){
        return (this.role.equals(((Admin) o).getRole()) && (this.workingHours.equals(((Admin) o).getWorkingHours())));
    }
    else return false;
}
```

The toString and equals method from object class to abide by best practices.

1.3 Organizer

The organizer class is for users who are able to create and manage events. Organizers are also able to view event statistics, view the attendees for their events. An organizer has an attributes of user class (since it inherits from it) and additionally has balance attribute of type wallet.

The figure below shows the method getOrganizedEvents which takes an object of type category as parameter and return array list. This method searches the database to find the organized events which fall under a certain category.

```
public ArrayList<Event> getOrganizedEvents(Category category){  2 usages  Jasmin
    System.out.println(category);
    Object[] T = Database.readAll((new Event()));
    if(category != null) T = category.getEvents().toArray();
    Event[] eventArray = new Event[T.length];
    for(int i=0;i<T.length;i++){
        eventArray[i] = (Event)T[i];
    }
    ArrayList<Event> eventArrayFiltered = new ArrayList<>();
    for(Event event:eventArray){
        if(event.getEventOrg() == this) eventArrayFiltered.add(event);
    }
    return eventArrayFiltered;
}
```

In addition to the previous method, this class overrides toString and equals to abide with best practices.

1.4 Attendee

Attendee is the last class in the program that inherits from the user and implements the HasID interface. The attendee also has the attributes from user as well as the following attributes:

```
private String ID;  4 usages
private int age;  9 usages
private String city;  5 usages
private Wallet balance;  9 usages
private Category[] interest = new Category[3];  8 usages
private ArrayList<Event> bookedEvents = new ArrayList<>();  6 usages
```

The array of categories is made to store 3 categories which are chosen by the attendee when registering. Those categories are used to display the recommended events to the attendee.

Next, the booked events array list is where the all the events the attendee is a part of (by buying tickets) are stored.

```
public Attendee() { this.ID = "A" + System.nanoTime(); }

public Attendee(String email, String username,String password, DateTime dateOfBirth, Gender gen,int age,   kim28 +1
                String address, double walletBalance, String interest1, String interest2, String interest3) {...}
```

As shown in the photo above, this class has a no arg constructor and a parametrized constructor to be able to create full attendee.

```
public int getAge() { return age; }
public String getCity() { return city; }
public double getBalance() { return balance.getBalance(); }
public void attendeeDeposit(double money) { balance.deposit(money); }

public Category[] getInterest() { return interest; }
public ArrayList<Event> getBookedEvents() { return bookedEvents; }
public void setID(String ID) { this.ID = ID; }
public void setAge(int age) { this.age = age; }
public void setCity(String city) { this.city = city; }
public void setBalance(Wallet balance) { this.balance = balance; }
public void setInterest(Category[] interest) { this.interest = interest; }
public void setBookedEvents(ArrayList<Event> bookedEvents) { this.bookedEvents = bookedEvents; }
```

The above photo shows the getters and setters for the attendee class.

The following method is the base for the event statistics viewed by organizer.

```java
public String getAgeGroup(){  1 usage  Ahmed Emara
    if(age < 18){...}
    else if(age < 30){
        return "18-30";
    }
    else if(age < 50){
        return "30-50";
    }
    else return ">50";
}
```

The following snippet of code shows the method that allows the attendee to purchase tickets and it also decrements the balance of the user. This also adds the event in the booked events array for the attendee and adds the attendee to the event array.

```java
public void bookTickets(Event event, Attendee attendee, int ticketCount) {  1 usage  Ahmed Emara
    this.balance.withdraw( amount: ticketCount*event.getTicketPrice());
    bookedEvents.add(event);
    event.addAttendee(attendee,ticketCount);
}
```

1.5 Category

This class implements *HasID* and the following code snippet in shows the class attributes with corresponding comments describing the purpose of each attribute.

```java
private final String catID; //store the ID of each category  4 usages
private String catName; // name of the category  7 usages
public static int totCats = 0; // stores total number of categories;  2 usages
private ArrayList<Event> events = new ArrayList<>();// stores events under each category;  7 usages
private static ArrayList<Category> catList = new ArrayList<>();  3 usages
```

As shown in the following code snippet, a no Arg constructor was made to match the best practices in developing a code which also initializes attributes. As for the parametrized constructor, it initializes the necessary attributes and increments the total number of categories as a whole each time a new object of type category is created.

```
//constructors
// no-arg constructor
public Category() {  3 usages  kim28 *
    this.catID = "C" + System.nanoTime();
    totCats++;
}

// Arg constructor
public Category(String catName) {  2 usages  kim28 +2
    this.catName = catName;
    this.catID = "C" + System.nanoTime();
    totCats++;
}
```

This class also contains getters and setters for all attributesThe following code shows the setters and getters of the class used to access and edit the attributes**.** The getID method is also overridden to implement the HasID interface.

```java
public final void ValidateCatAccess(User obj){  3 usages  kim28 *
    if (!(obj instanceof Admin)) {
        throw new AccessDenied("You do not have permission to use this method." +
                " \n Only Admins are allowed to create categories");
    }
}
```

Since only admins are allowed to Create, delete and update, this method has been made to throw an exception if a user that is not an admin tries to access it as displayed in the figure above.

The method below saves all categories into an array list.

```java
static public void listCat(){  1usage  24P0270 +1
    Object[] T = Database.readAll(new Category());

    Category[] options = new Category[T.length];
    for (int i = 0; i < T.length; i++) {
        options[i] = (Category) T[i];
    }

    catList.clear();
    for(Category e : options){
        catList.add(e);
    }
}
```

In addition to all of the above, this class implements runnable, which allows the usage of multithreading. Multithreading is utilized by the usage of the method mentioned above. The listCat method is put in a try catch syntax which allows the the cateogories to be updated in real time from the database every 20 seconds.

```
@Override   24P0270 +1
public void run() {
    while (true){
        try{
            listCat();
            Thread.sleep( millis: 200);
        }catch (InterruptedException e){
            System.out.println("thread was intruppted");
            break;
        }
    }
}
```

1.6 DateTime

This class is used to represent a calendar where each attribute allows scheduling and enables booking and many other functionalities later to be shown in the code. TimeSlot is an enum that contains morning, afternoon and evening that will be explained later in the code.

```
    // setters & getters
    public int getDay() { return day; }

    public int getMonth() { return month; }

    public int getYear() { return year; }

    public TimeSlot getTime() { return time; }

    public void setDay(int day) {  no usages   kim28

        this.day = day;
    }

    public void setMonth(int month) { this.month = month; }

    public void setYear(int year) { this.year = year; }

    public void setTime(TimeSlot time) { this.time = time; }
```

```java
private int day;  4 usages
private int month;  4 usages
private int year;  4 usages
public TimeSlot time;  5 usages
public DateTime(){}  kim28
public DateTime(int day,int month , int year) { this(day,month,year, time: null); }
public DateTime(int day, int month, int year, TimeSlot time){  kim28
    this.day = day;
    this.month = month;
    this.year = year;
    this.time = time;
}
public DateTime(String S){  Ahmed Emara +1 *
    S = checkFormat(S);
    if(S == null){
        throw new RuntimeException("invalid format");
    }
    this(Integer.parseInt(S.substring(0,2)),
            Integer.parseInt(S.substring(3,5)),Integer.parseInt(S.substring( beginIndex: 6)));
}
```

This class contains a method made to specifically check if the string given is written in the right date format.

```java
public static String checkFormat(String s){  1 usage  Ahmed Emara
    if(s.length() < 8) return null;
    if(s.charAt(1)=='/') s = "0" + s;
    if(s.charAt(4)=='/') s = s.substring(0,3) + "0" + s.substring( beginIndex: 3);
    if(s.length() != 10) return null;
    if(s.charAt(2) != '/' || s.charAt(5) != '/') return null;
    return s;
}
```

1.7 AccessDenied (exception class)

This class and the following classes are custom made exception classes

```java
public class AccessDenied extends RuntimeException {  1 usage  kim28
    public AccessDenied(String message) { super(message); }
}
```

1.8 ExceedLimit (exception class)

```
public class ExceedLimit extends RuntimeException {  1 usage  kim28
    public ExceedLimit(String message) { super(message); }
}
```

1.9 AlreadyExists (exception class)

```
package BackEnd;

public class AlreadyExists extends RuntimeException {  1 usage  kim28
    public AlreadyExists(String message) { super(message); }
}
```

1.10 EventnotAvaible(exception class)

```
package BackEnd;

public class EventnotAvaible extends RuntimeException {  1 usage  kim28
    public EventnotAvaible(String message) { super(message); }
}
```

1.11 FundsNOTenough(exception class)

```
package BackEnd;

public class FundsNOTenough extends RuntimeException {  1 usage  kim28
    public FundsNOTenough(String message) { super(message); }
}
```

1.12 InvalidCategoryindex(exception class)

```
public class InvalidCategoryindex extends RuntimeException {  no usages  kim28
    public InvalidCategoryindex(String message) { super(message); }
}
```

1.13 NotPositiveAmount(exception class)

```
public class NotPostiveAmount extends RuntimeException {  1 usage  kim28
    public NotPostiveAmount(String message) { super(message); }
}
```

1.14 Database

This class is the main data store in the whole program. It utilizes hashmaps in order to be able to store the data. It has many methods (CRUD methods for all data stored inside the hashmap) including methods used for create,update,delete, read (individual), and read all for items stored in the hashmap.

```java
public static void create(Object o) {  kim28
    if(!(o instanceof HasID)){
        throw new RuntimeException("Invalid Object");
    }
    data.put(((HasID)o).getID(),o);
}
public static Object read(String ID){  2 usages  kim28
    Object o = data.get(ID);
    if(o == null){
        throw new RuntimeException("Object Not Found");
    }
    return o;
}
public static Object[] readAll(Object o){  22 usages  kim28
    ArrayList<Object> ret = new ArrayList<>();
    for(String ID:data.keySet()){
        if(!(o.getClass()).equals((data.get(ID)).getClass())) continue;
        ret.add(data.get(ID));
    }
    return ret.toArray();
}
public static void update(Object o) {  2 usages  kim28
    if(!(o instanceof HasID)){
        throw new RuntimeException("Invalid Object");
    }
    data.replace(((HasID)o).getID(),o);
}
```

```java
public static void delete(Object o){  3 usages  kim28
    if(!(o instanceof HasID)){
        throw new RuntimeException("Invalid Object");
    }
    data.remove(((HasID)o).getID());
}
```

The following code searches for users, events and categories inside the database

```java
public static User findUser(String username, String password){  6 usages  kim28
    Object[] A = readAll(new Attendee());
    for(Object o:A) {
        if(((Attendee)o).getUsername().equals(username)
                && ((Attendee)o).getPassword().equals(password)) return (Attendee)o;
    }
    A = readAll(new Organizer());
    for(Object o:A) {
        if(((Organizer)o).getUsername().equals(username)
                && ((Organizer)o).getPassword().equals(password)) return (Organizer)o;
    }
    A = readAll(new Admin());
    for(Object o:A) {
        if(((Admin)o).getUsername().equals(username)
                && ((Admin)o).getPassword().equals(password)) return (Admin)o;
    }
    return null;
}
public static ArrayList<Event> findEvent(String eventName){  3 usages  24P0270 +1 *
    Object[] T = readAll(new Event());
    Event[] options = new Event[T.length];
    for (int i = 0; i < T.length; i++) {
        options[i] = (Event) T[i];
    }
    ArrayList<Event> AllEvents = new ArrayList<>();
    for(Event e : options){
        if(eventName == null || eventName.isEmpty() ||
                e.getEventName().toLowerCase().contains(eventName.toLowerCase()) ){
            AllEvents.add(e);
        }
    }
    return AllEvents;
}
```

```java
public static Category findCat(String catName){  15 usages  Jasmin
    Object[] E = readAll(new Category());
    for(Object o:E) {
        if(((Category)o).getCatName().equals(catName)) return (Category) o;
    }
    return null;
}
```

In addition, there is a large method that creates all objects in an external file to accommodate file handling.

```java
public static void scanInput(File source) {  1 usage  kim28 +2 *
    Scanner in;
    try{
        in = new Scanner(source);
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
    String s = in.next();
    while(!s.equals("*")){
        System.out.println(s);
        switch (s){
            case "Wallet":
                create(new Wallet(in.nextDouble()));
                break;
            case "Room":
                create(new Room(in.next(),in.nextInt(),in.nextDouble()));
                break;
            case "Attendee":
                create(new Attendee(in.next(),in.next(),in.next(),new DateTime(in.next()),
                        (in.nextBoolean()? Gender.MALE : Gender.FEMALE),in.nextInt(),in.next()
                        ,in.nextDouble(),in.next(),in.next(),in.next()));
                break;
            case "Organizer":
                create(new Organizer(in.next(),in.next(),in.next(),new DateTime(in.next()),
                        (in.nextBoolean()? Gender.MALE : Gender.FEMALE),in.nextDouble()));
                break;
            case "Admin":
                create(new Admin(in.next(),in.next(),in.next(),new DateTime(in.next()),
                        (in.nextBoolean()? Gender.MALE : Gender.FEMALE),in.next(),in.next()));
                break;
            case "Category":
                create(new Category(in.next()));
                break;
            case "Event":
                create(new Event(in.next(),(Category)random(new Category()),(Room)random(new Room()),
                        (Organizer)random(new Organizer()),in.nextDouble(),new DateTime()));
        }
        s = in.next();
    }
    System.out.println(s);
    in.close();
}
```

1.15 Event

The event class has the following attributes. Where the last attribute is a hashmap that is used specifically to store data related to gender, demographics and age group of attendees of a certain event. This is to allow the organizer to view event statistics.

```
private String eventID;  7 usages
private String eventName;  5 usages
private Category eventCat;  5 usages
private Room eventRoom;  5 usages
private Organizer eventOrg;  4 usages
private double ticketPrice;  6 usages
private DateTime eventDate;  8 usages
private int eventRoomCap;  5 usages
private int eventAttendees=0;  5 usages
private HashMap<String[],Integer> statistics = new HashMap<>();
```

It also contains constructors: a no arg constructor and a parametrized constructor

```
//Constructors
//no arg constructor
public Event(){this.eventID= "E"+System.nanoTime();}  kim28

//argument constructor
public Event(String eventName, Category eventCat, Room eventRoom, Organizer eventOrg ,
             double ticketPrice, DateTime eventDate){

    //a unique username is set using nano time
    this.eventID= "E"+System.nanoTime();
    this.eventName= eventName;
    this.eventCat= eventCat;
    this.eventRoom= eventRoom;
    this.eventOrg= eventOrg;
    this.ticketPrice= ticketPrice;
    this.eventDate = eventDate;

    //adding this event object to array of this certain category
    eventCat.addEvent(this);
    this.eventRoomCap=eventRoom.getRoomCapacity();
}
```

The event class also includes setters and getters for all of its attributes.

Next, there is a method called isThereEnough which returns a boolean that whether or not the number of tickets required by the attendee is available. The other method in the code snippet below is addAttendee which adds an attendee to the events' array. It also increments the balance of the organizer by 70% of the purchases ticket prices. Also, it updated the statistics hash map.

```
public boolean isThereEnough(int nOfTickets){   1 usage   kim28
    return eventAttendees + nOfTickets <= eventRoomCap;
}
public void addAttendee(Attendee attendee, int nOfTickets){   2 usages   Ahmed Emara +1
    eventAttendees += nOfTickets;
    eventOrg.getBalance().deposit( amount: 0.7*nOfTickets*ticketPrice);
    String[] key = new String[]{attendee.getAgeGroup(),attendee.getGen()==Gender.MALE?"m":"f"};
    statistics.putIfAbsent(key, 0);
    statistics.replace(key,statistics.get(key)+nOfTickets);
}
```

The following code snippet shows the overridden methods which are toString and equals. However in this case, there is an extra to string that is specifically designed for use with attendee, which allows the attendee to see only the necessary information (and not all info available like an admin would).

```
public String AttendeeToString(){   3 usages   Jasmin +1
    String s;
    s="\nEvent name: "+ eventName + "\nEvent Category:"+ eventCat.getCatName() +
            "\nEvent Room:"+ eventRoom.getRoomName() + "\nTicket Price:"+
            ticketPrice + "\nNumber of tickets remaining:"+ (int)(eventRoomCap-eventAttendees) +
            " \nEvent Date(dd/mm/yyyy):" + eventDate.getDay()+"/"+eventDate.getMonth()+"/"+ eventDate.getYear() +
            "\nEvent time slot"+ eventDate.getTime() +"\n\n";

    return s;
}
@Override   kim28
public String toString(){
    String s;
    s="Event ID:"+eventID+"\nEvent name:"+ eventName + "\nEvent Category:"+ eventCat.toString() +
            "\nEvent Room:"+ eventRoom.toString() + "\nEvent organizer:" + eventOrg.toString()+ "\nTicket Price:"+
            ticketPrice + "\nNumber of Event Attendees:"+ eventAttendees+ " out of" + eventRoomCap +
            " hours\nEvent Date:" + eventDate.toString() + "\n\n";

    return s;
}

public boolean equals(Event event){return(this.eventID.equals(event.eventID));}   kim28
```

1.16 Entrance

This class is an aid for GUI login and register to create and initialize objects by values taken from the text fields in the login/register interface

```
public static void registerOrganizer(String email, String username, String password,  2 usages  Jasmin
                                     String DOB,boolean gender,String walletBalance){
    Gender gen= gender?Gender.FEMALE:Gender.MALE;

    Database.create(new Organizer(email,username,password,new DateTime(DOB),gen,Double.parseDouble(walletBalance)));
}

public static void registerAttendee(String email, String username, String password,  2 usages  Jasmin
                                    String DOB,boolean gender,String age, String address, String walletBalance,
                                    String interest1, String interest2, String interest3){
    Gender gen= gender?Gender.FEMALE:Gender.MALE;
    Database.create(new Attendee(email, username, password, new DateTime(DOB),gen, Integer.parseInt(age),
            address,Double.parseDouble(walletBalance),interest1, interest2, interest3));
}

public static User login(String username, String password){  2 usages  Jasmin
    User curUser = Database.findUser(username, password);
    return curUser;
}
```

1.17 Room

This class represents information about physical rooms. It has the following attributes.

```
private final String roomID;  4 usages
private String roomName;  4 usages
private int roomCapacity;  4 usages
private double rentPrice;  4 usages
private Schedule bookedSlots = new Schedule();  3 usages
private static ArrayList<Room> roomList = new ArrayList<>();  3 usages
```

This class implements both HasID and Runnable. HasID requires the room to have a valid id that it can override getID. Also, it overrides run to update list room every 2 seconds in real time.

```java
@Override  24P0270
public void run() {
    while (true){
      try{
          listRooms();
          Thread.sleep( millis: 2000);
      }catch (InterruptedException e){
        System.out.println("thread was intruppted");
        break;
      }
    }
}
```

The class contains both setters and getters for all of its attributes

```java
// accessors \\
public String getID() { return roomID; }
public String getRoomName() { return roomName; }
public int getRoomCapacity() { return roomCapacity; }
public void setRoomCapacity(int roomCapacity) { this.roomCapacity = roomCapacity; }
public double getRentPrice() { return rentPrice; }
// mutators \\
public void setRoomName(String roomName) { this.roomName = roomName; }
public void setRentPrice(double rentPrice) { this.rentPrice = rentPrice; }
public boolean isAvailable(DateTime slot) { return bookedSlots.isAvailable(slot); }

public static ArrayList<Room> getRoomList() { return roomList; }

public void reserveSlot(DateTime slot, Event event) { bookedSlots.add(slot,event); }
```

It also contains both no arg and parametrized constructors

```
public Room() { this( roomName: "", roomCapacity: 100, rentPrice: 100.00); }
public Room(String roomName, int roomCapacity, double rentPrice){   kim28 +1
    roomID = "R" + System.nanoTime();
    this.roomName = roomName;
    this.roomCapacity = roomCapacity;
    this.rentPrice = rentPrice;
}
```

In the following snippet, there is a method called listrooms which add all rooms to an arraylist . Next, run utilizes multithreading to update the array list of rooms in real time every 2 seconds.

```
static public void listRooms(){   1 usage   24P0270 +1
    Object[] T = Database.readAll(new Room());
    Room[] options = new Room[T.length];
    for (int i = 0; i < T.length; i++) {
        options[i] = (Room) T[i];
    }
    roomList.clear();
    for(Room e : options){
            roomList.add(e);
    }
}
@Override   24P0270
public void run() {
    while (true){
      try{
          listRooms();
          Thread.sleep( millis: 2000);
      }catch (InterruptedException e){
        System.out.println("thread was intruppted");
        break;
      }
    }
}
```

1.18 RunCatChecker and RunRoomChecker

Both are classes related to the usage of multithreading in updating the arraylist of rooms and categories in time

```
public class RunCatChecker {  2 usages  24P0270
    public static ExecutorService executor;  4 usages
    RunCatChecker() {  no usages  24P0270

    }
    public static void refreshCat(){  1 usage  24P0270
        if(executor == null||executor.isShutdown()) {
            executor = Executors.newFixedThreadPool( nThreads: 1);
        }
        Category r = new Category();
        executor.execute(r);
    }
}
```

```
public class RunRoomChecker {  4 usages  24P0270 +1
    public static ExecutorService executor;  6 usages
    RunRoomChecker() {  no usages  24P0270

    }
    public static void refreshroom(){  1 usage  24P0270 +1
        if(executor == null||executor.isShutdown()) {
            executor = Executors.newFixedThreadPool( nThreads: 1);
        }
        Room r = new Room();
        executor.execute(r);
    }
}
```

1.19 Gender

Gender is an enum containing values of MALE and FEMALE

```
public enum Gender {   Ahmed Emara +1
    MALE,   8 usages
    FEMALE;   9 usages
    public String getTitle(){   1 usage   Ahmed Emara
        if(this == MALE) return "Mr.";
        else return "Mrs.";
    }
}
```

1.20 HasID

This is an interface that requires the classes to have a valid ID and requires the classes that implement it to override getID

```
public interface HasID {   12 usages   8 implementations   kim28
    public String getID();   7 implementations   kim28
}
```

1.21 TimeSlot

This is an enum that has values for event slots MORNING, AFTERNOON , EVENING

It also has method to translate the values from the radiobutton input in GUI

```java
public enum TimeSlot{  16 usages  Yahia_g +1
    MORNING,  3 usages
    AFTERNOON,  2 usages
    EVENING;  3 usages
    public static TimeSlot translate(int x){  no usages  kim28
        return switch (x) {
            case 0 -> TimeSlot.MORNING;
            case 1 -> TimeSlot.AFTERNOON;
            case 2 -> TimeSlot.EVENING;
            default -> null;
        };
    }


    public String toString() {  Yahia_g
        if(this==MORNING){
            return "Morning";
        }else if(this==EVENING){
            return "Evening";
        }else{
            return "Afternoon";
        }

    }
    static public TimeSlot stringTo(String s){  2 usages  Yahia_g
   if(s.equals("MORNING")){
       return MORNING;
   }else if(s.equals("AFTERNOON")){
       return AFTERNOON;
   }else{
       return EVENING;
   }
    }
```

1.22 Schedule

This class is mainly a hashmap that tackles the point of knowing whether or not a room is available at a certain point in time.

```java
public class Schedule {  2 usages  kim28
    private HashMap<DateTime,Event> hashMap;  6 usages
    public Schedule() { hashMap = new HashMap<>(); }
    public boolean isAvailable(DateTime dateTime) { return hashMap.get(dateTime) == null; }
    public void add(DateTime dateTime, Event event) { hashMap.put(dateTime, event); }
    public void remove(DateTime dateTime) { hashMap.remove(dateTime); }

    @Override  kim28
    public String toString() {
        StringBuilder ret = new StringBuilder("{");
        for(DateTime slot:hashMap.keySet()){
            if(ret.length() != 1) ret.append("; ");
            ret.append((String)("{Slot: " + slot.toString() + "; Event: " + hashMap.get(slot).getID() + "}"));
        }
        ret.append("}");
        return ret.toString();
    }
}
```

1.23 Wallet

This class contains the following attributes

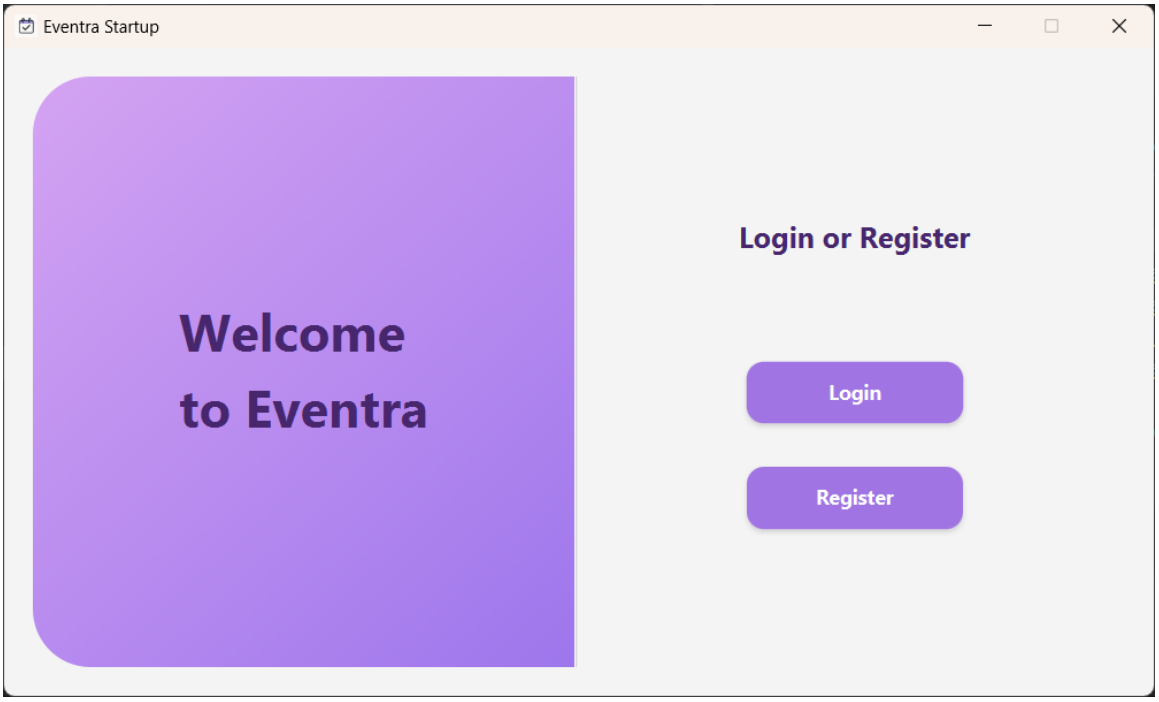
```java
private final String walletID;  4 usages
private double balance;  6 usages
```

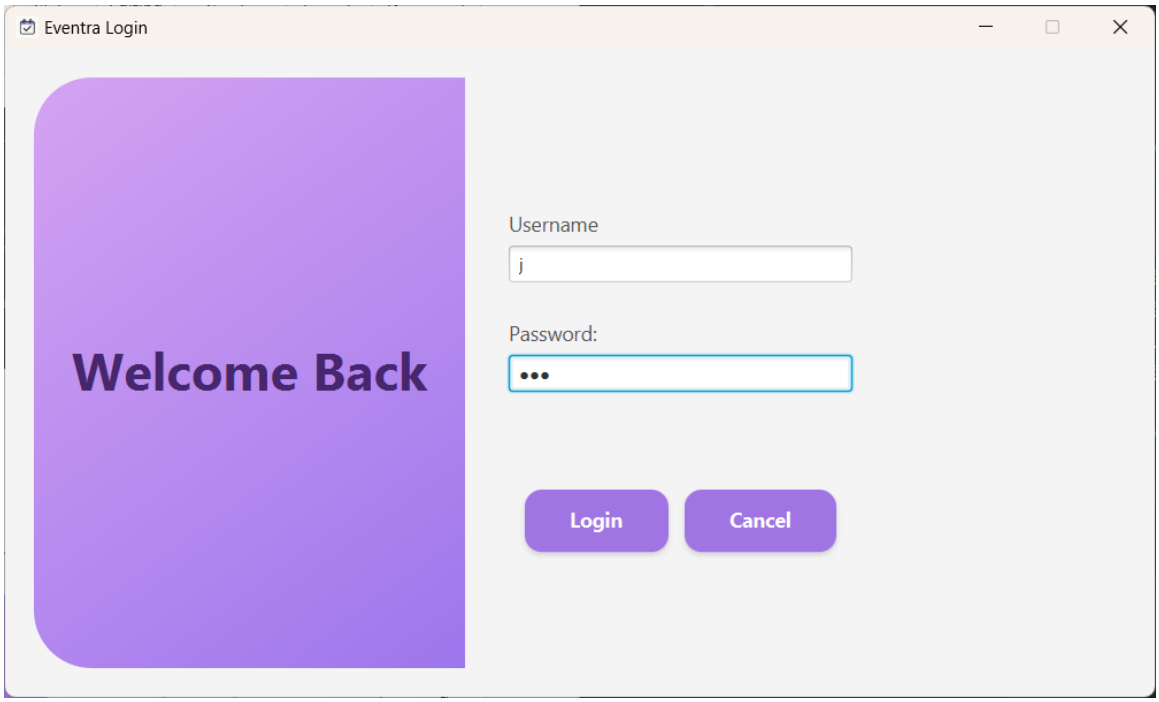
It contains necessary methods like withdraw, isSufficient, and getter methods

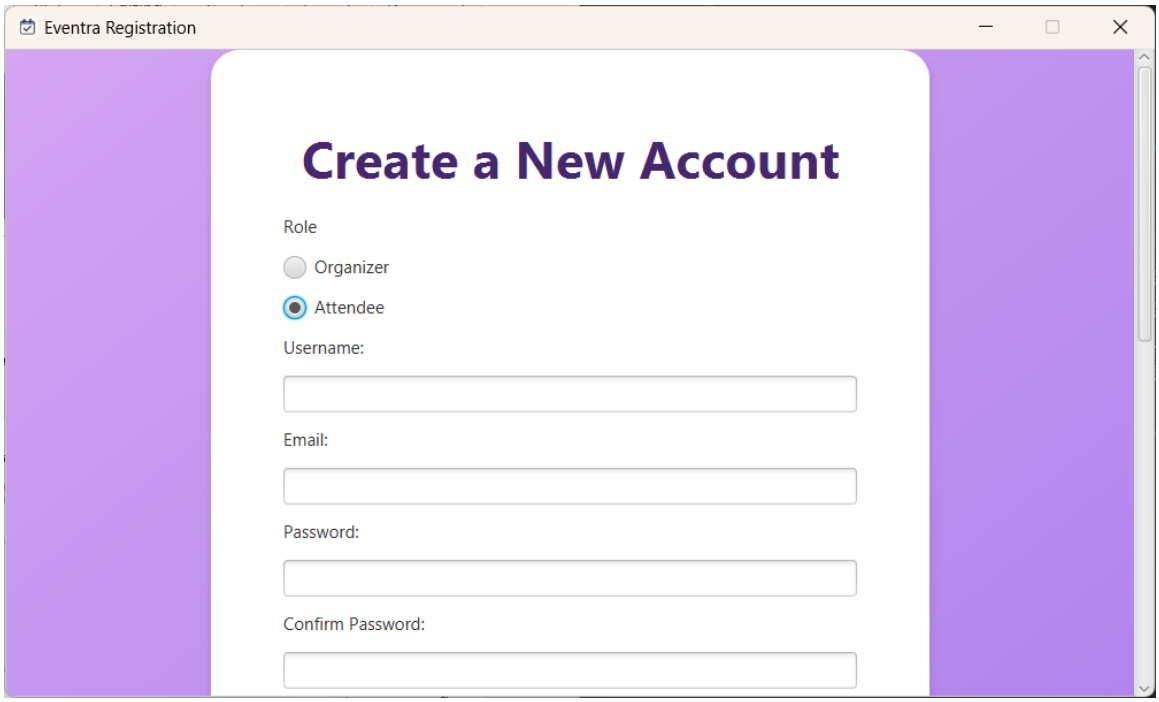
```java
public double getBalance() { return balance; }
public boolean isSufficient(double amount) { return (balance >= amount); }
public void deposit(double amount){  4 usages  kim28
    balance += amount;
    Database.update( o: this);
}
public void withdraw(double amount){  4 usages  kim28
    if(!isSufficient(amount)) throw new IllegalArgumentException("Insufficient Balance; balance: " + balance
    deposit(-amount);
}
public void transferTo(double amount, Wallet recipient){  no usages  kim28
    withdraw(amount);
    recipient.deposit(amount);
}
```

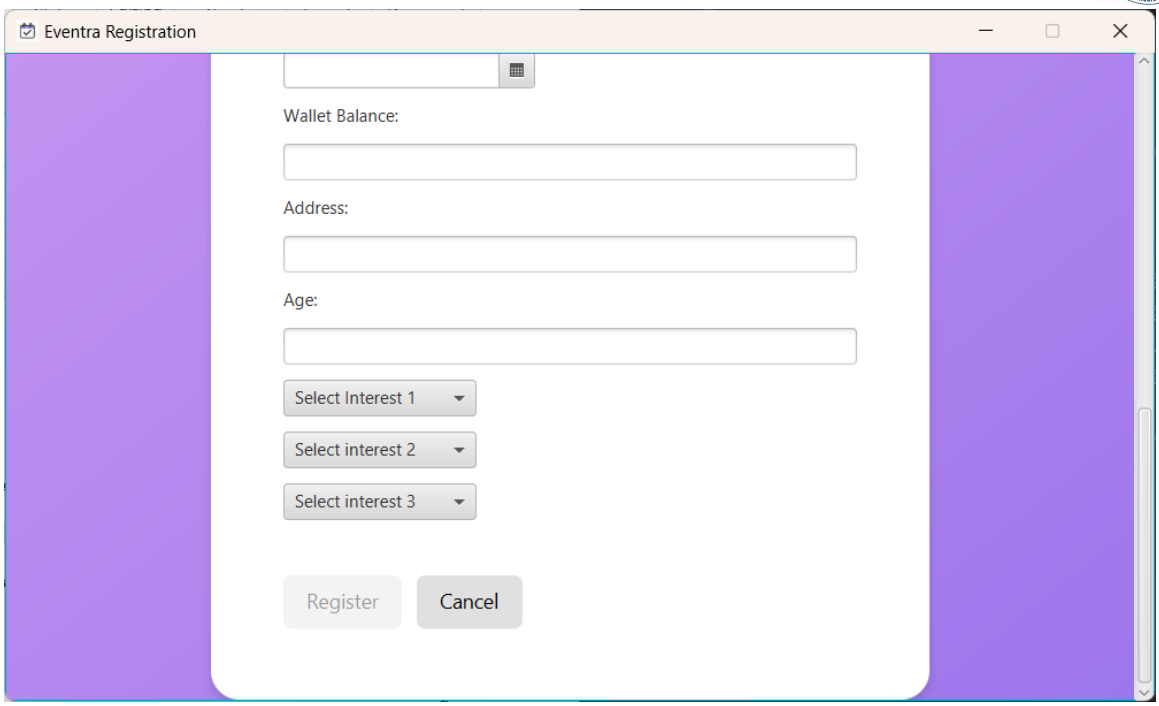
2 Front End

2.1 Login and Register

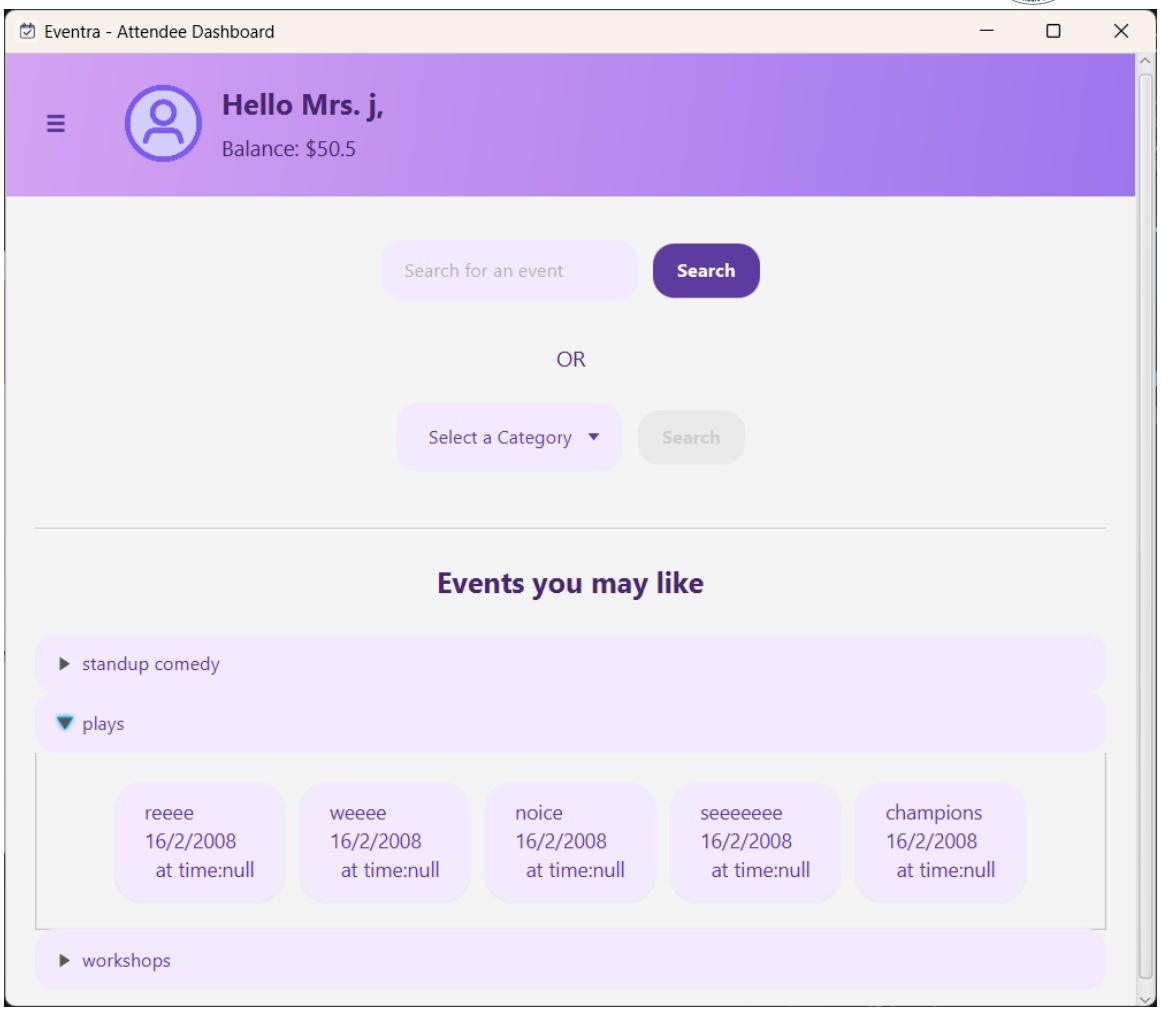


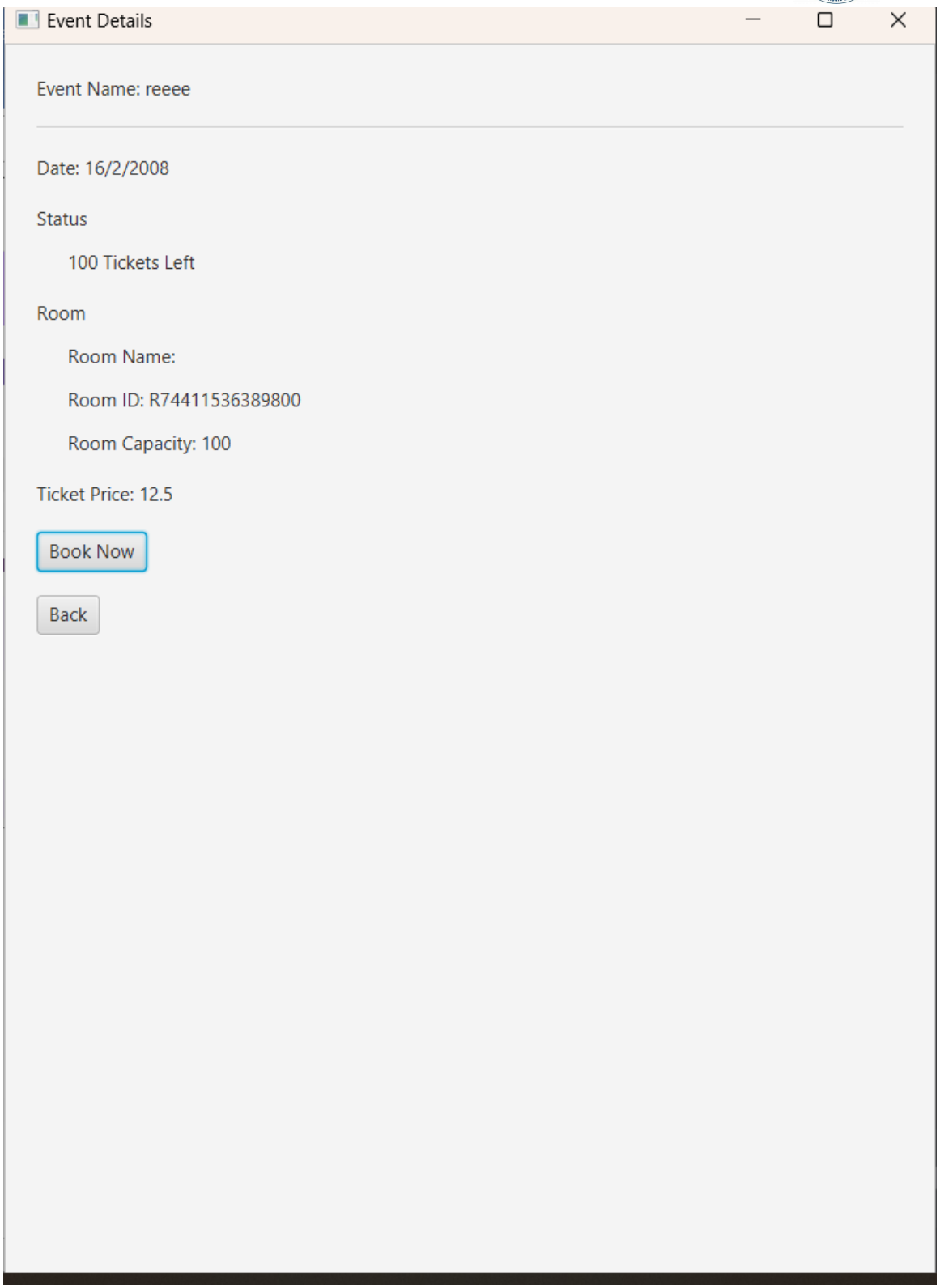

Eventra Login
Welcome Back
Username
j
Password:
Login
Cancel

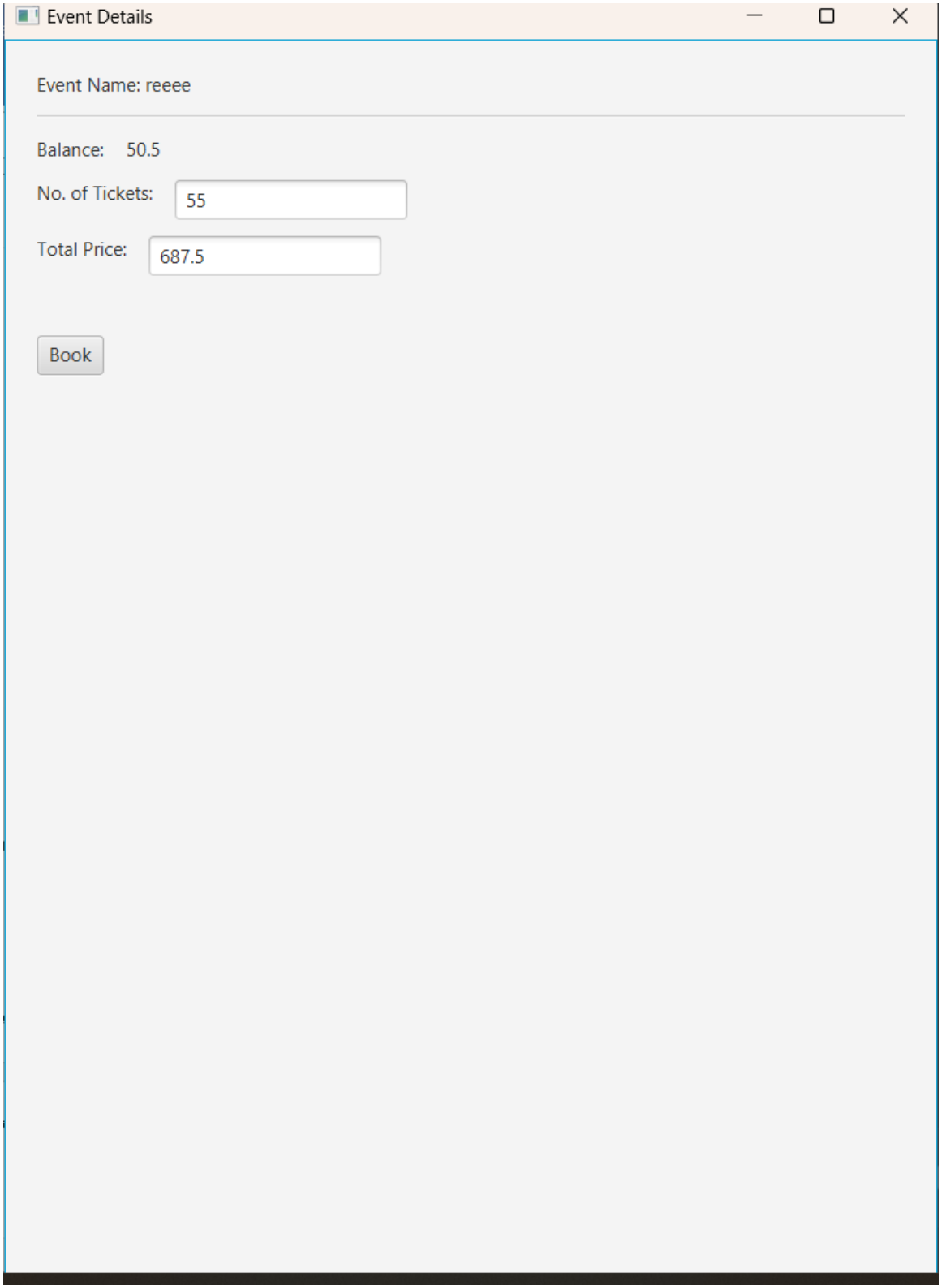

Eventra Registration
Create a New Account
Role
Organizer
Attendee
Username:
Email:
Password:
Confirm Password:

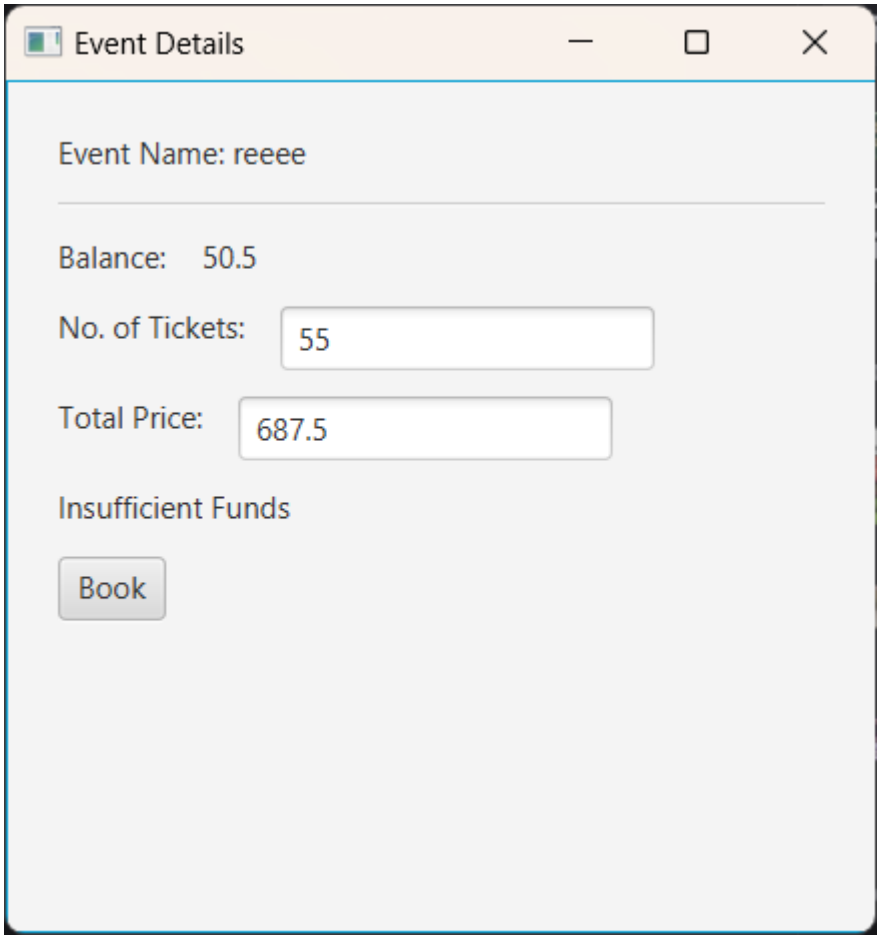


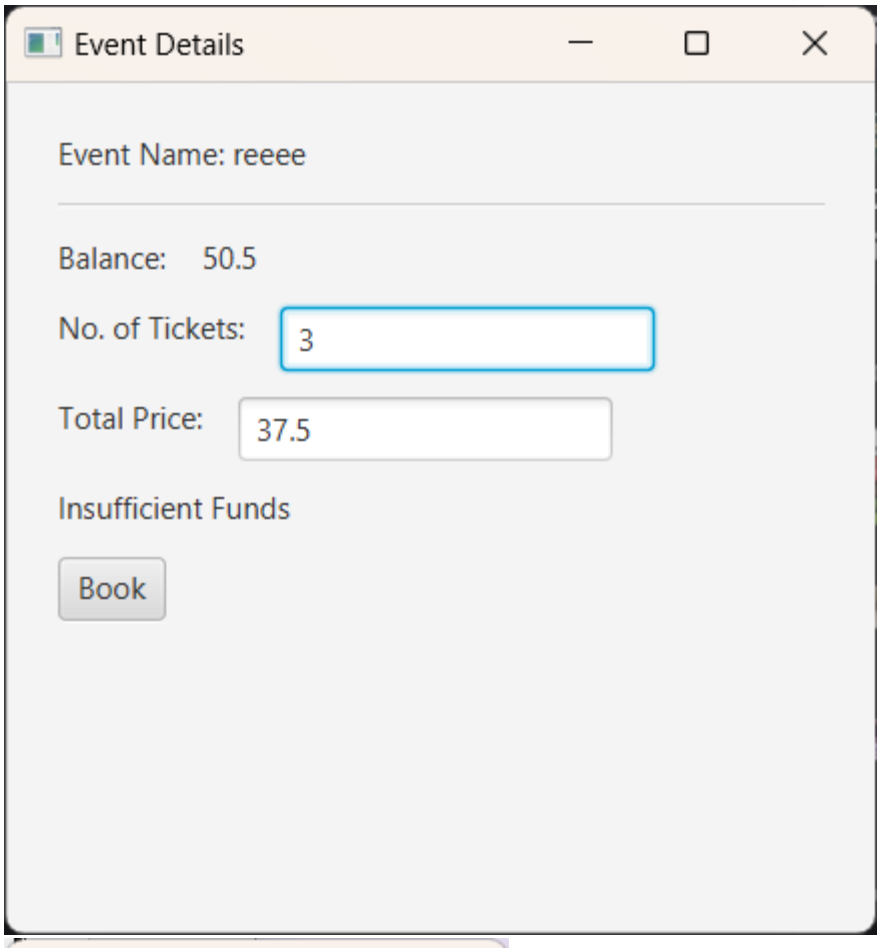
2.2 Attendee Interface

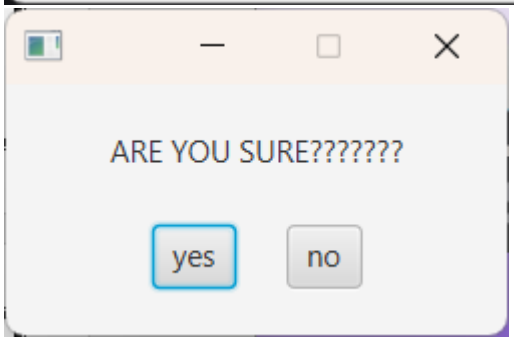

Eventra - Attendee Dashboard
Hello Mrs. j,
Balance: $50.5
Search for an event
Search
OR
Select a Category
Search
Events you may like
standup comedy
plays
reeee
16/2/2008
at time:null
weeee
16/2/2008
at time:null
noice
16/2/2008
at time:null
seeeeeee
16/2/2008
at time:null
champions
16/2/2008
at time:null
workshops

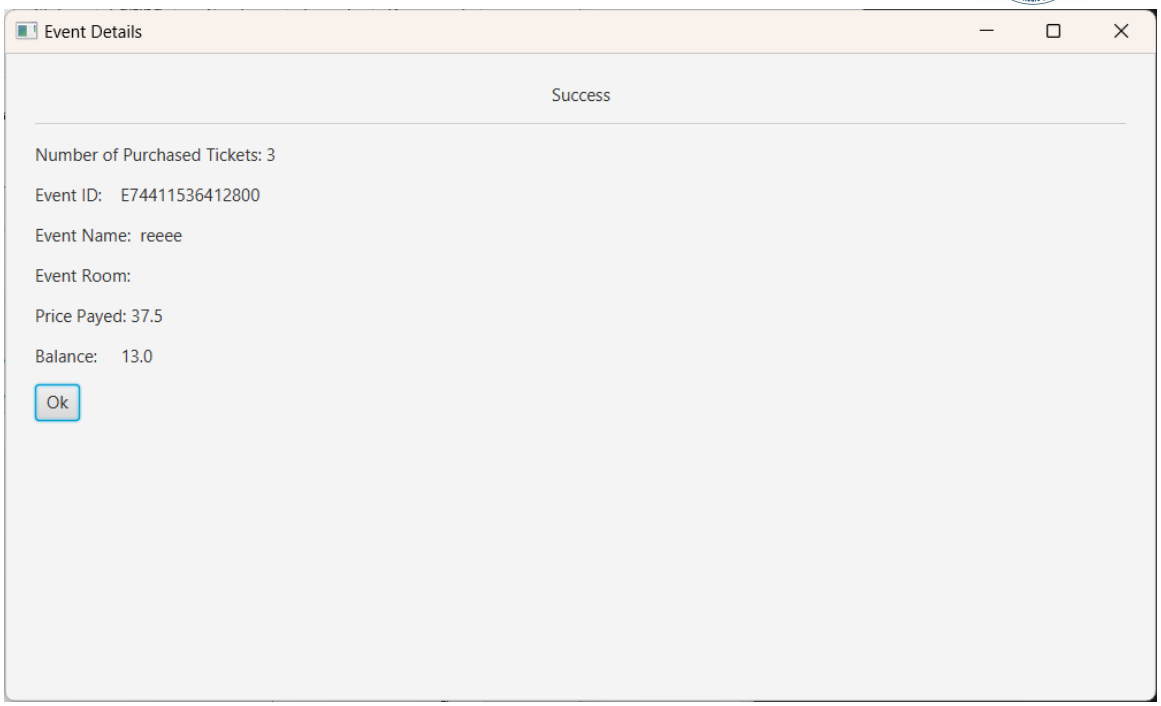

Event Details
Event Name: reeee
Date: 16/2/2008
Status
100 Tickets Left
Room
Room Name:
Room ID: R74411536389800
Room Capacity: 100
Ticket Price: 12.5
Book Now
Back

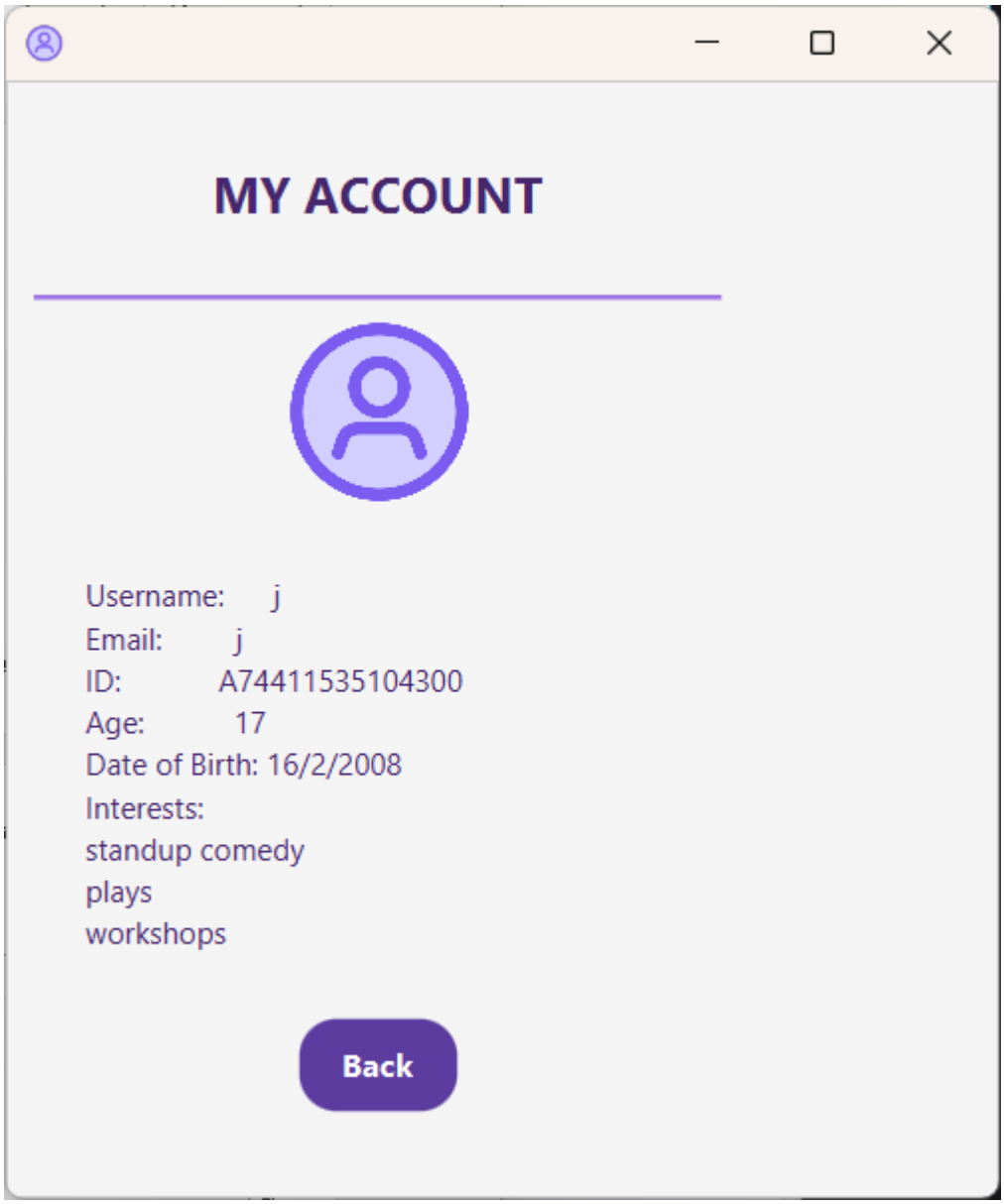

Event Details
Event Name: reeee
Balance: 50.5
No. of Tickets:
55
Total Price:
687.5
Book

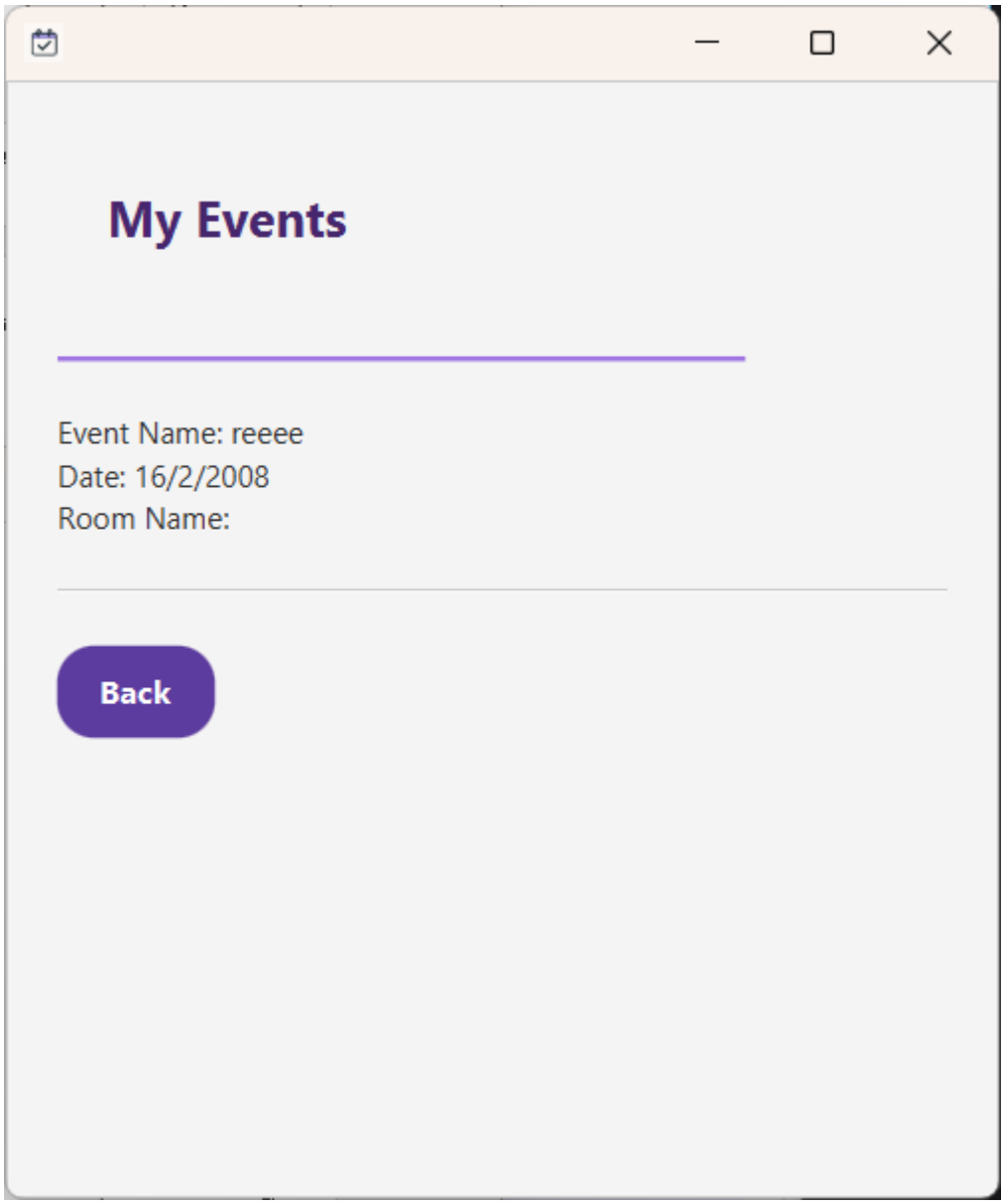

Event Details
Event Name: reeee
Balance: 50.5
No. of Tickets:
55
Total Price:
687.5
Insufficient Funds
Book


Event Details
Event Name: reeee
Balance: 50.5
No. of Tickets:
3
Total Price:
37.5
Insufficient Funds
Book

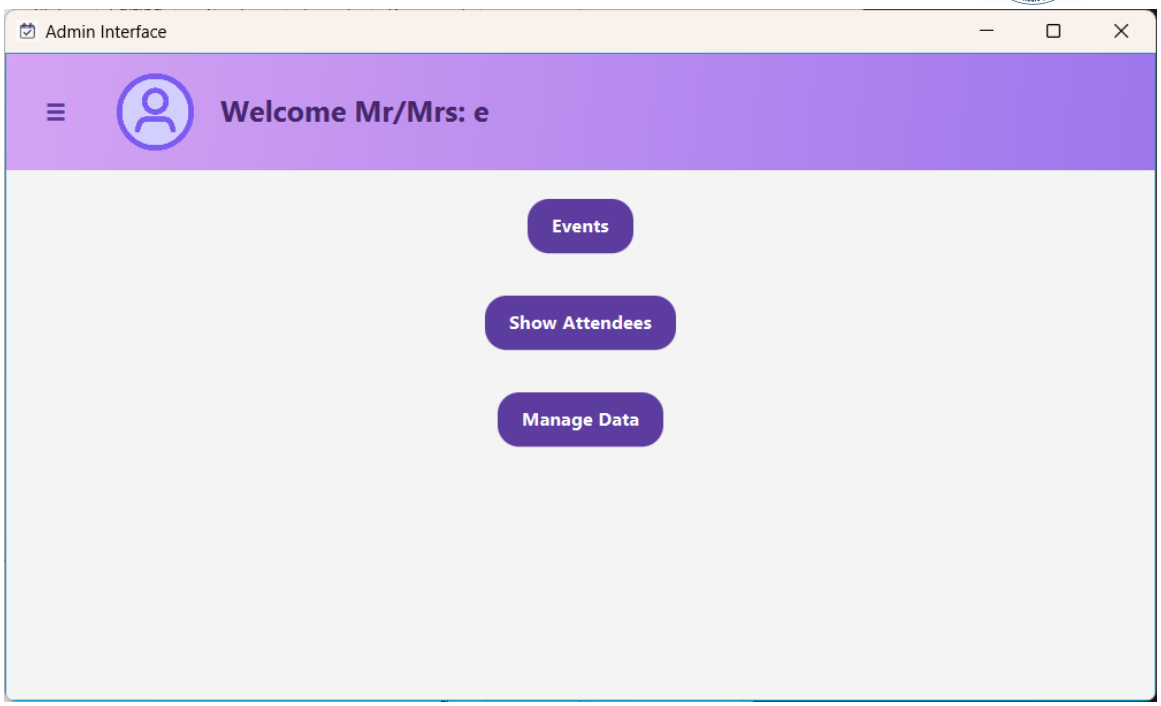

ARE YOU SURE???????
yes
no

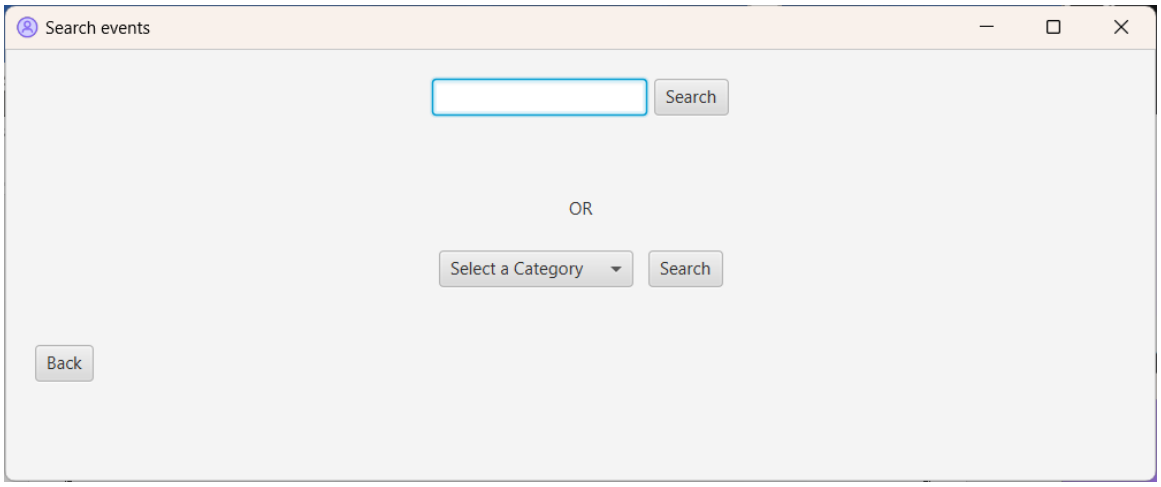

Event Details
Success
Number of Purchased Tickets: 3
Event ID: E74411536412800
Event Name: reeee
Event Room:
Price Payed: 37.5
Balance: 13.0
Ok


MY ACCOUNT
Username: j
Email: j
ID: A74411535104300
Age: 17
Date of Birth: 16/2/2008
Interests:
standup comedy
plays
workshops
Back

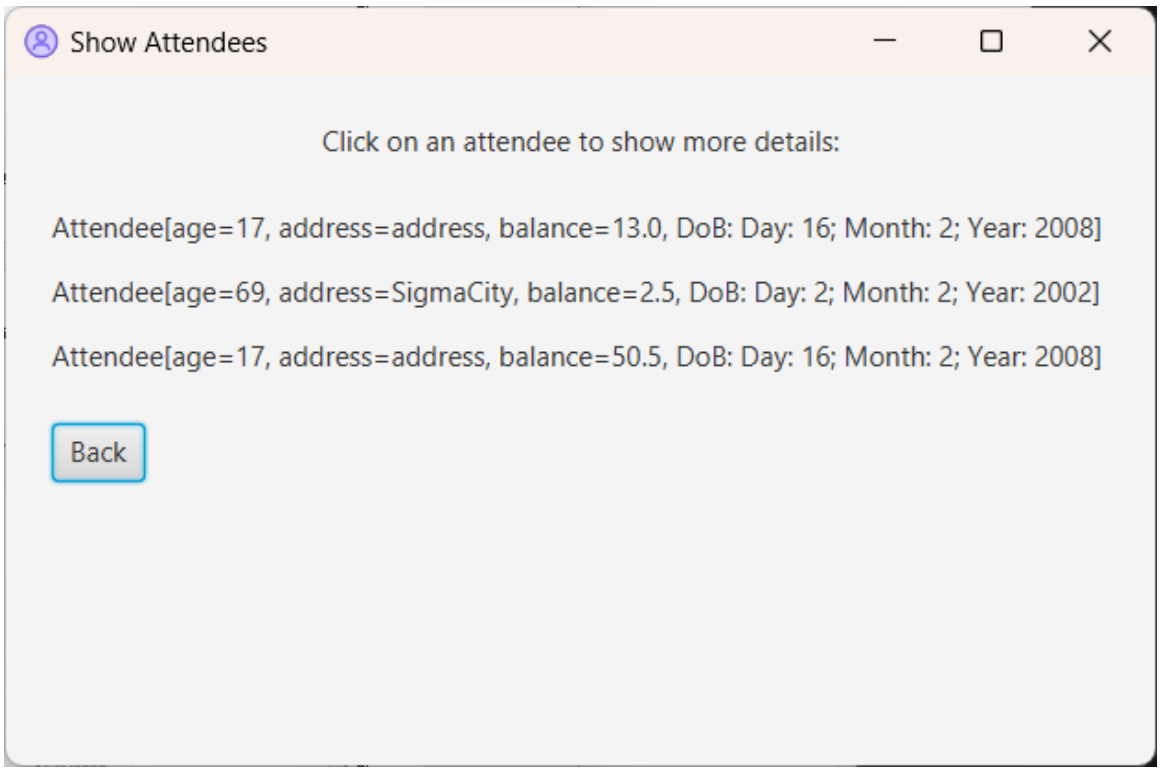


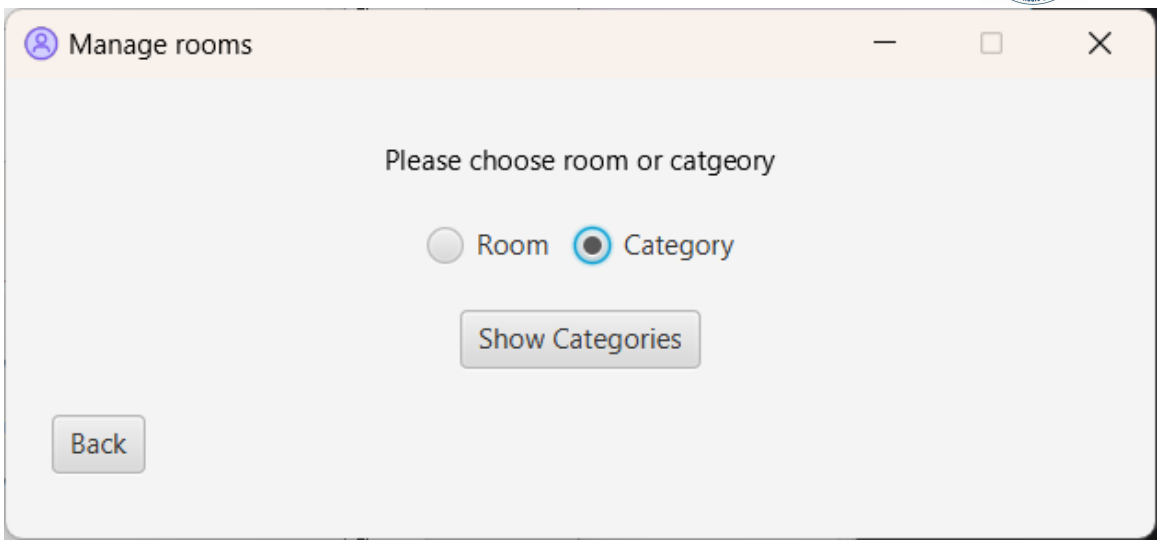
2.3 Admin Interface

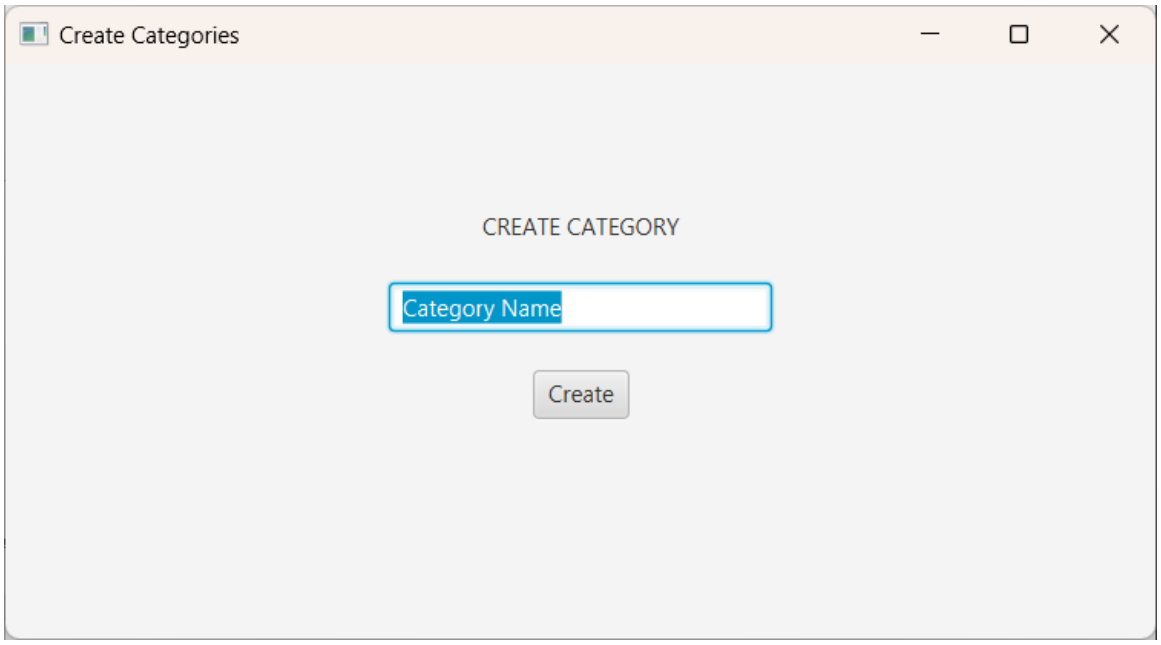

Admin Interface
Welcome Mr/Mrs: e
Events
Show Attendees
Manage Data

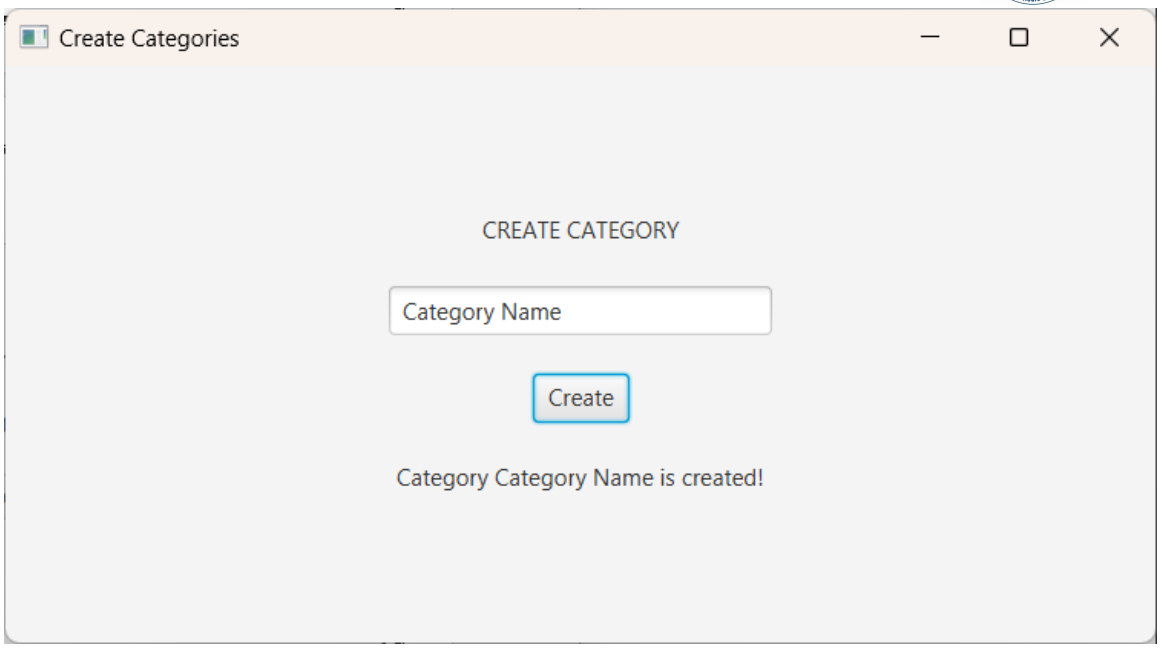

Search events
Search
OR
Select a Category
Search
Back


Search events
Search
OR
plays
Search
reeee
16/2/2008
at time:null
weeee
16/2/2008
at time:null
noice
16/2/2008
at time:null
seeeeeee
16/2/2008
at time:null
champions
16/2/2008
at time:null
Back


Show Attendees
Click on an attendee to show more details:
Attendee[age=17, address=address, balance=13.0, DoB: Day: 16; Month: 2; Year: 2008]
Attendee[age=69, address=SigmaCity, balance=2.5, DoB: Day: 2; Month: 2; Year: 2002]
Attendee[age=17, address=address, balance=50.5, DoB: Day: 16; Month: 2; Year: 2008]
Back


Manage rooms
Please choose room or catgeory
Room
Category
Show Categories
Back


Create Categories
CREATE CATEGORY
Category Name
Create


Create Categories
CREATE CATEGORY
Category Name
Create
Category Category Name is created!