

# EVENTAK

*An event creating and organizing system for event planners.*

**Mariam Maarouf**

Spring 2016

Advanced Programming Applications

## PURPOSE

This application is designed to help **event planners** create, manage, and keep records of events, venues, and contacts. For the purpose of this project, the user will be able to manage two types of events: private parties, and concerts.

## DETAILED DESCRIPTION

The user this system is designed for will be able to:

- Create events of type Party or Concert.
- Set and edit details for those events (for example: sponsors and their payment methods, venue, date, etc)
- Produce reports for several purposes (for example: a complete summary of the event, the guest list, number of tickets sold if applicable, all events for a certain month/year/week/day, etc).
- Add venues, either separately, or for certain events.
- Add contacts, either separately, or for certain venues, or entertainment groups (band, DJ, show, etc).
- Generate invitations for private parties.
- Generate and keep track of sold tickets of different tiers (“VIP”, “Standard”, etc) for concerts.
- Estimate cost per person, depending on the budget set for a private party.

# PHASE ONE

## BASIC ARCHITECTURE AND FUNCTIONALITY

### CLASSES

Regular classes:

- Event ***abstract***
  - Parent class for all event types (currently Concert and Party).
  - Attributes: name, venue, guest list, date, budget, sponsors.
  - Methods: getters and setters, add guest (checks for uniqueness of guest inside the guest list), get budget per person (calculates based on number of guests added, throws an exception if not enough info is provided or no guests have been added), get sponsor by ID (index in sponsors list).
  - Abstract methods: invite guest (generate invitation and add guest), summarize (produce a String summary).
- Ticketing ***interface***
  - The interface that allows an event type to implement the ticketing system. Tickets are divided into tiers (for example, VIP and Standard) set by the user. Each tier has its own price, limit, and available number of units.
  - Methods: add ticket tier, change ticket tier price, change ticket tier limit (add/subtract or set a new one), buy tickets, get all available tickets, get all available tickets of a certain tier, and generate a ticket of a certain tier.
- Concert
  - A type of event that can be created.
  - Extends **Event** and implements **Ticketing**.
  - Extra attributes: list of entertainment, hash of ticket tiers (every tier corresponds to its price, limit, and available units in an array), and 2D list of payments (each row: payment method, amount paid, how many tickets, tier of tickets).
  - Extra methods: add and get entertainment, get tickets sold (in general or a certain tier), and get total money made.
- Contact
  - Can be created independently or within a Venue/Entertainment.
  - Attributes: name, list of emails, list of phones.

- Methods: getters and setters, as well as a custom **.toString()**
- Cost
  - Created to manage budgets and/or actual cost (in the future).
  - Attributes: total, number of people.
  - Methods: setters and getters, as well as a method to get cost per person that throws an exception if not enough info is provided.
- Entertainment
  - A class for entertainment objects/groups, such as bands, shows, DJs, etc.
  - Attributes: name, contact (type: Contact), and time (in which they will perform in a certain event).
  - Methods: setters and getters, as well as a **.toString()** implementation.
- Guest
  - Object for guests.
  - Extends **Contact**.
  - Extra attributes: RSVP (boolean, indicates if they're attending. Default is true), address (optional), and comment (optional).
  - Extra methods: setters and getters, as well as a **.toString()** implementation, and a method to reverse RSVP (if true, make false. If false, make true).
- Party
  - Another type of events.
  - Extends **Event**.
  - Extra attributes: list of highlighted people (example: bride, groom), list of entertainment, and type of event (String description).
  - Extra methods: setters and getters, as well as implementations for the abstract methods in Event.
- Payment
  - A tool someone (for now, only sponsors) can *have*. It stores payment methods, either simple (one column, example: "Cheque") or detailed (two columns, method and detail, example: "Visa Card" "119281931").
- Run **test class/main**
  - Contains list of contacts, venues, concerts, and parties that will be initialized from a file later in phase two.
- Sponsor
  - A special type of contact who can pay.
  - Extends **Contact**.
  - Extra attributes: payment (new, and public, so its methods can be

accessible).

- Extra method: a **.toString()** implementation.
- Venue
  - An object for the place where an event is held.
  - Attributes: name, address, phones, contact (type: Contact).
  - Methods: setters and getters, as well as a **.toString()** implementation.

Exceptions: (all unchecked)

- AlreadyExistsException
  - For when something already exists. Thrown when a guest had already been added to an event.
- NotAddedException
  - Thrown if someone is trying to access/edit an attribute that hasn't been added yet.
- NotEnoughInfoException
  - Thrown if someone is trying to retrieve a summary and not all the factors/attributes needed are available.
- OutofLimitException
  - Thrown if the user attempts to buy more tickets than the available limit.

## SAMPLE OUTPUT

By running the application, the user is met with a main menu (in class **Run**), as follows:

1. Create Party
2. Create Contact
3. List Parties
4. List Contacts
5. Add Contact
6. Add Venue
7. List Contacts
8. List Venues
9. Exit the program

They then choose one (it keeps prompting them for an answer/choice until one is valid). The creation choices all prompt the user for the least possible constructors needed to create an object, creates it, then adds it to the **ArrayList<OBJECT>** corresponding to the type.

The list choices print a list of the `.toString()` values (or names in case of parties/concerts) along with their ID (index) in the `ArrayList<OBJECT>` object.

Aside from running the application, if functions are directly called from `main()`, a sample input could be something like this:

```
// MAIN MENU
System.out.println("Welcome!");

Party woo = new Party("PARTY NAME", "PARTY TYPE");
Guest foo = new Guest("GROOM");
Guest moo = new Guest("John Doe");
Venue awe = new Venue("VENUE NAME", "VENUE ADDRESS", "+22390213184");
woo.addHighlight(foo); // GROOM is now the highlight of the party called PARTY NAME
woo.setDate(2016, 5, 1, 17, 00); // Date is set to the 1st of May 2016 at 5PM
woo.addVenue(awe);
woo.addSponsor("Rich Person", "1291314143", "richperson@richpeople.com"); // first sponsor, ID = 0
woo.getSponsorById(0).payment.addMethod("Visa Card", "10921931981931");
System.out.println(woo.inviteGuest(moo)); // generates invite, handles its own error
try{
    woo.addGuest(moo); // adding a guest we've already added
}
catch(AlreadyExistsException ex){
    System.out.println(ex.getMessage());
}
woo.addEntertainment("Awesome Band", new GregorianCalendar(2016, 5, 1, 19, 00));
woo.setBudget(2000);
System.out.println(woo.summarize());
```

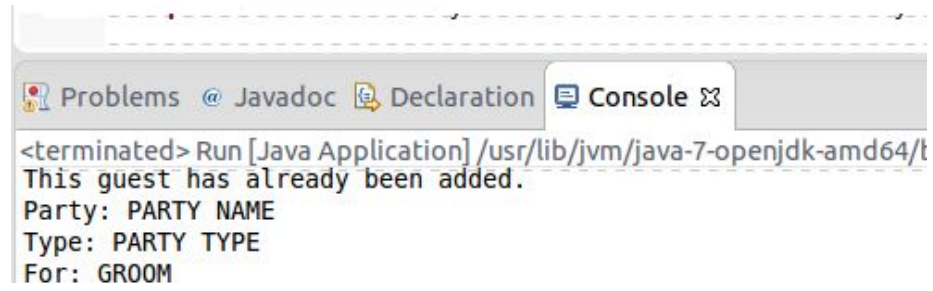
Output:

```
<terminated> Run [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/bin/java (Apr 10, 2016, 12:14:04 AM)
Welcome!
~~~~~
Dear John Doe,
You have been invited to attend a PARTY TYPE party by GROOM. RSVP, regrets only.
Venue : VENUE NAME
Date: Wed Jun 01 17:00:00 EET 2016
~~~~~
This guest has already been added.
Party: PARTY NAME
Type: PARTY TYPE
For: GROOM
Sponsor(s):
ID: 0
Name: Rich Person
Emails: [ richperson@richpeople.com ]
Phones: [ 1291314143 ]
Payment Methods:
Visa Card      10921931981931

Date: Wed Jun 01 17:00:00 EET 2016
Venue:
Venue name: VENUE NAME
Address: VENUE ADDRESS
Phones: [ +22390213184 ]
Contact: No contact has been added to this venue yet.
Budget:2000
Entertainment:
Name: Awesome Band
~~~~~
```

(cont.)

(cont.)



The screenshot shows a console window with a tab bar at the top containing 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. The output text in the console is as follows:

```
<terminated> Run [Java Application] /usr/lib/jvm/java-7-openjdk-amd64/t  
This guest has already been added.  
Party: PARTY NAME  
Type: PARTY TYPE  
For: GROOM
```



# PHASE TWO

## FILE I/O CAPABILITY

### CHANGE SUMMARY

Added the ability to write object ArrayLists in the main class **Run** that contain the working list of venues, parties, concerts, and contacts (and, by default, all the other objects those objects link to inside) to external files, one for every type of ArrayLists.

### CLASS CHANGE DETAILS

#### Run.java:

Added two functions:

```
initialize(){
    // initializes all objects needed for the program to work
    // from their corresponding files
    // file paths are '/tmp/ATTRIBUTENAME.ser'
    // Function is called as soon as program runs
}

save(){
    // saves all objects needed for the program to work
    // to their corresponding files
    // function is called when the program safely exits (when
    // the user chooses option 9 in the main menu)
}
```

#### Rest of the classes:

Changed all of them to implement **Serializable**, and as a result, added serialVersionUID to every one of them, as well as made sure there were empty/no arg constructors for the deserialization to work.

## **SAMPLE OUTPUT**

Same as before, the only difference is that created events/contacts/etc are now saved in the program directory and are initialized instead of “evaporating” after every run of the program.

# PHASE THREE

## Graphical User Interface using JavaFX

### CHANGE SUMMARY

Provide a GUI using JavaFX for the application. The GUI is then compiled into a runnable .jar file (eventak.jar). Functionality covered by the GUI:

- Create Party
- Manage Parties
- Create Concert
- Manage Concerts
- Create Contact
- Manage Contacts

Each with all the details included (list of entertainment, owners of parties, etc).

### CLASS CHANGE DETAILS

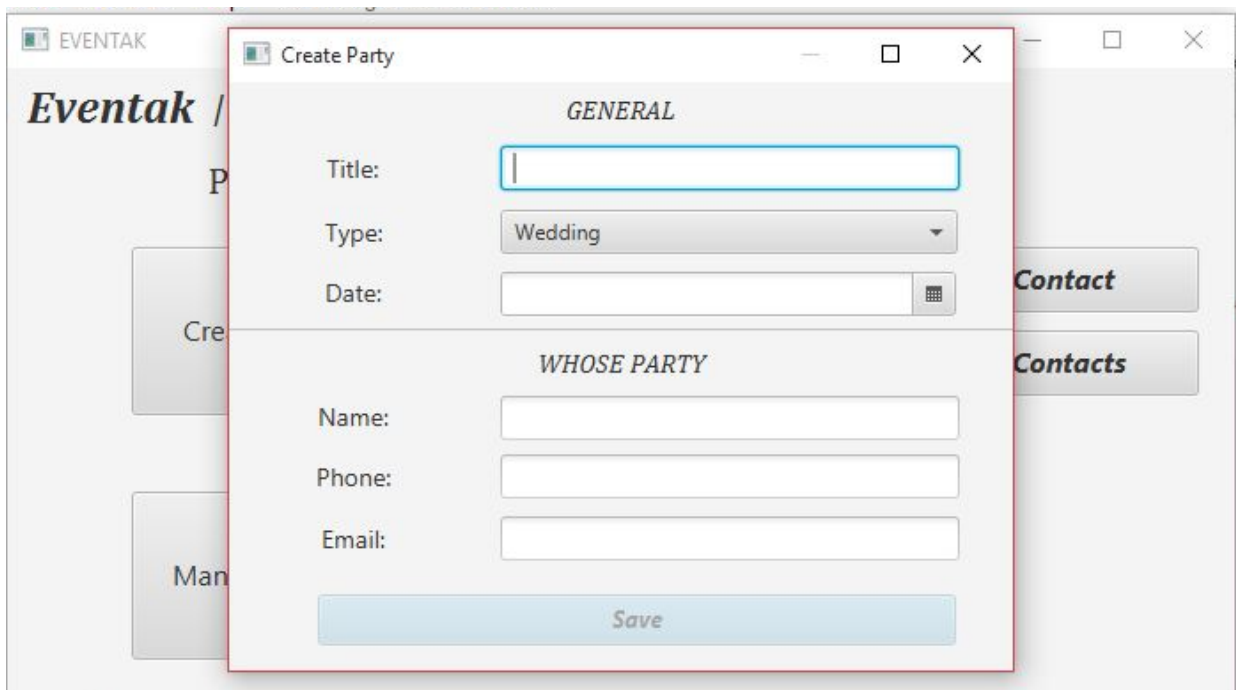
Added files:

- **Controllers:**
  - AddConcertController
  - AddContactController
  - AddEntertainmentController
  - AddGuestController
  - AddPartyController
  - AddSponsorController
  - EditGuestController
  - MainController
  - ManageConcertController
  - ManageContactController
  - ManageOwnersController
  - ManagePartyController
  - ManageSponsorsController
  - ManageTicketsController
  - ShowPartyController
- **FXML files corresponding to all controllers.**

## SCREENSHOTS

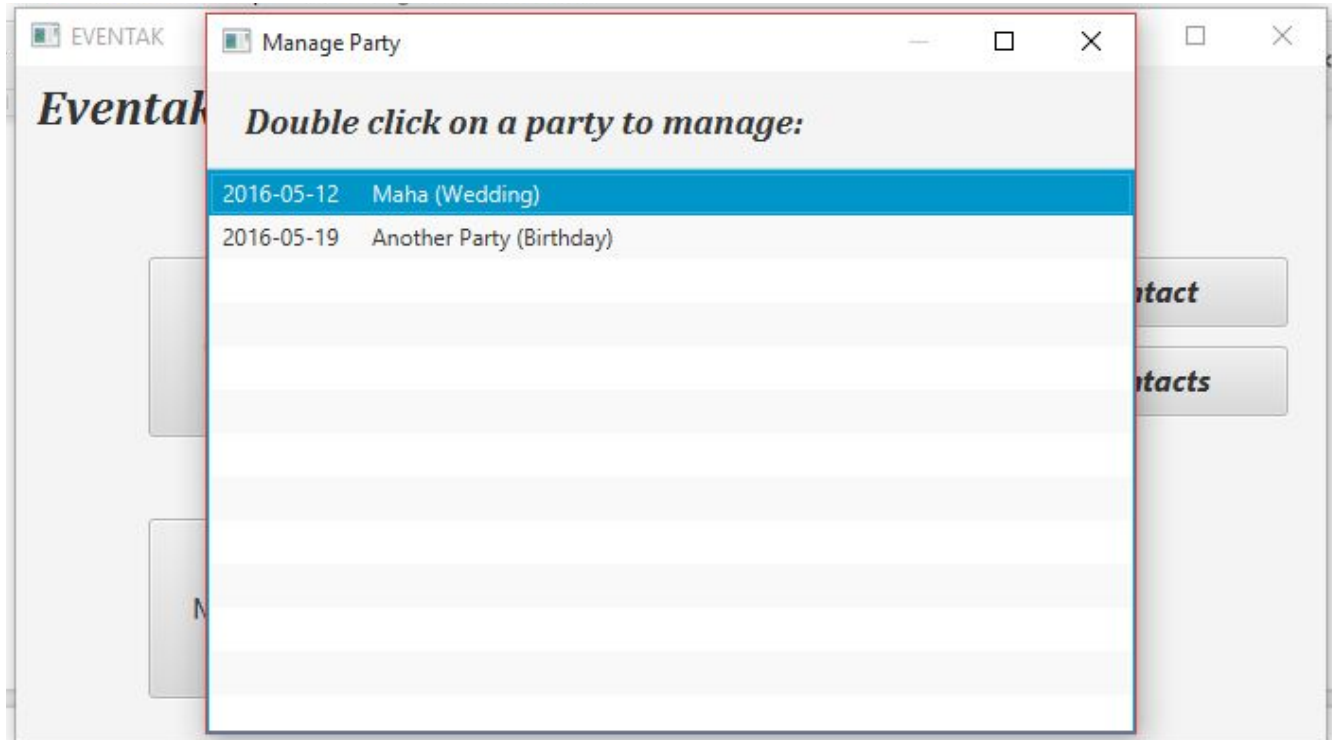


Main view.

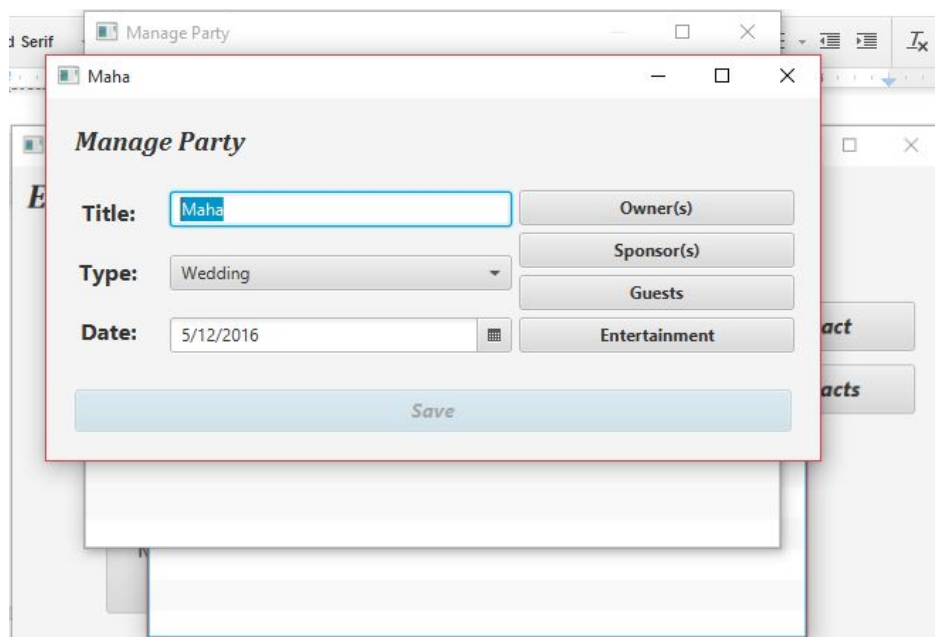


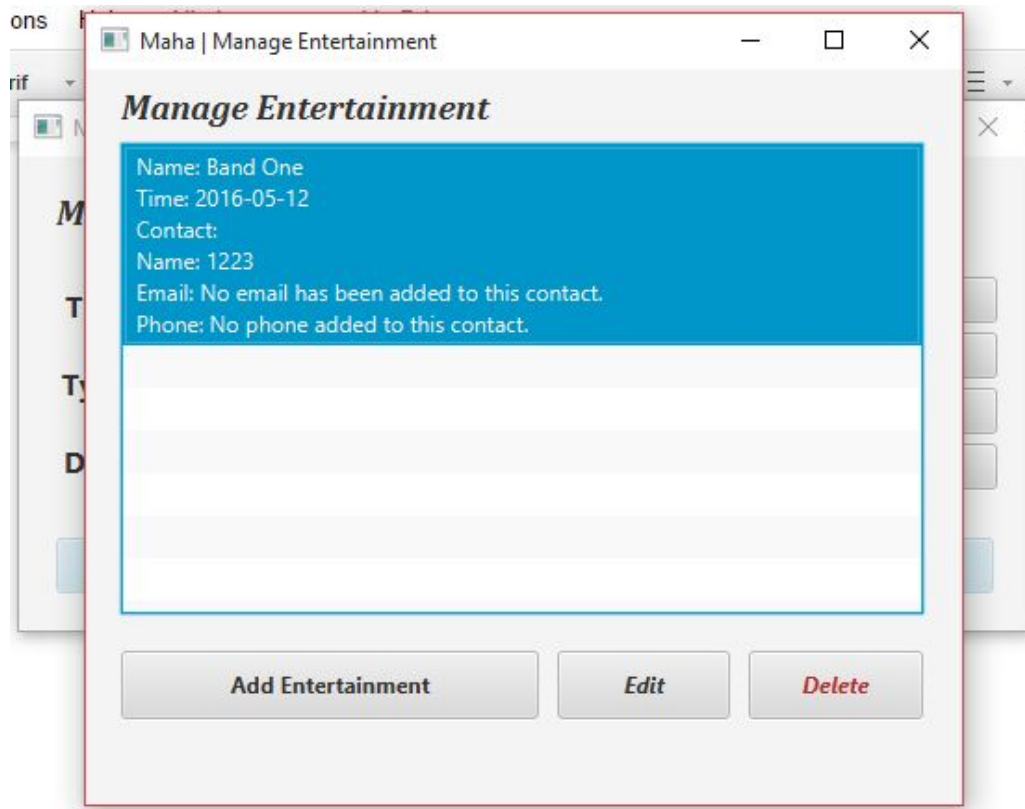
(prev) Add Party.

(next) Manage Parties.



(next) Show/Edit/Manage a single party.





(prev) Manage Entertainment.

Those are the main “looks” of the app, with minor changes between those and the rest (for example, there are different fields required for Concerts than Parties, etc.

# PHASE FOUR

## REMOTE ACCESS CAPABILITIES

### CHANGE SUMMARY

None. The GUI application is standalone. In this phase, I created a server app called eventak-server.jar, and a client app called eventak-client.jar. The server app contains all the classes from the previous phases, while the client app only contains the interface/doesn't need any files to run.

Both apps communicate over **localhost:8000**.

The server app can be run using the command:

```
$ java -jar eventak-server.jar
```

The client app is used to **buy tickets** from a certain **concert** and **ticket tier**. The server app relies on files in the same dir (concerts.ser to be exact) as a form of simplistic database. To emulate this, a concert with ticket tiers can be created using the main GUI app, then the server can run and the client can buy tickets.

### CLASS CHANGE DETAILS

SERVER:

A GreetingServer class that:

- Opens a socket/listener on port 8000.
- When a client connects, it gets the input stream (a comma-delimited string with the concert name, ticket tier, and number of tickets requested).
- The server checks the database from the concerts.ser file and returns an error if:
  - There's a mistake in the format of the request (the number field is not a number for example).
  - The concert does not exist in the database.
  - The concert exists, but the tier does not exist.
  - The concert and the tier exist, but the tickets requested are out of the limit set.
- If successful, the server returns a message with "SUCCESS" and the money spent

on those tickets (according to the database price).

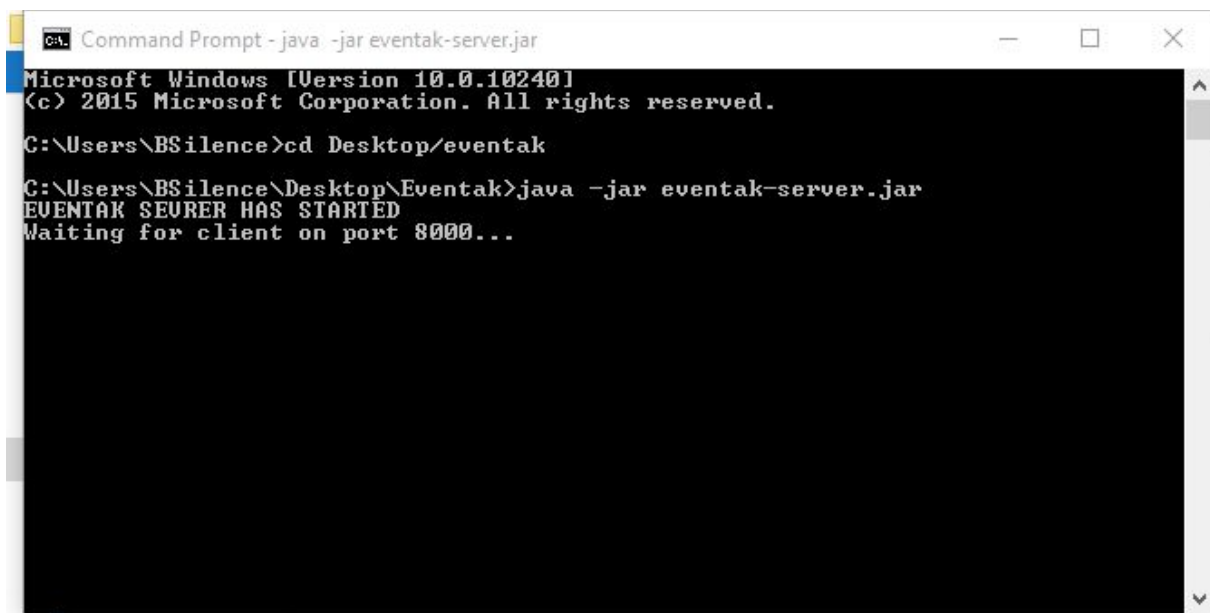
- After that is done, the server listens for more requests.

CLIENT:

A Main.fxml file and a corresponding MainController that:

- Displays messages and replies from the server in a non-editable textarea.
- Allows the user to enter the name, ticket tier and number of tickets.
- Validates the presence of the three of them, as well as the format of the number of tickets (is a number when parsed).
- If valid, opens a connection on localhost:8000 and sends a request to the server with the data in the format the server expects.
- Gets a reply, displays it in the textarea, then closes the connection.

## SCREENSHOT



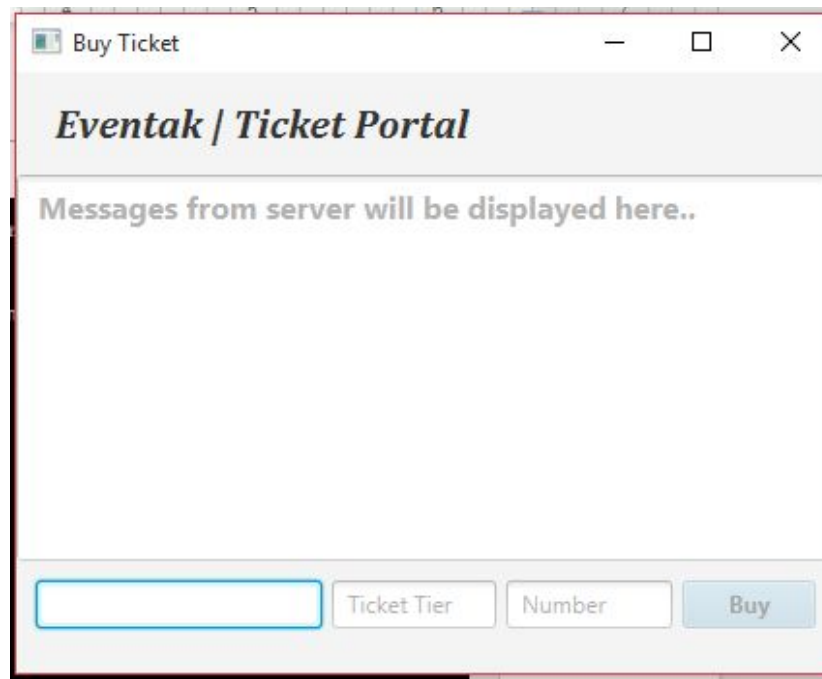
```
Command Prompt - java -jar eventak-server.jar
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\BSilence>cd Desktop/eventak

C:\Users\BSilence\Desktop\Eventak>java -jar eventak-server.jar
EVENTAK SEURER HAS STARTED
Waiting for client on port 8000...
```

Initial state of server.



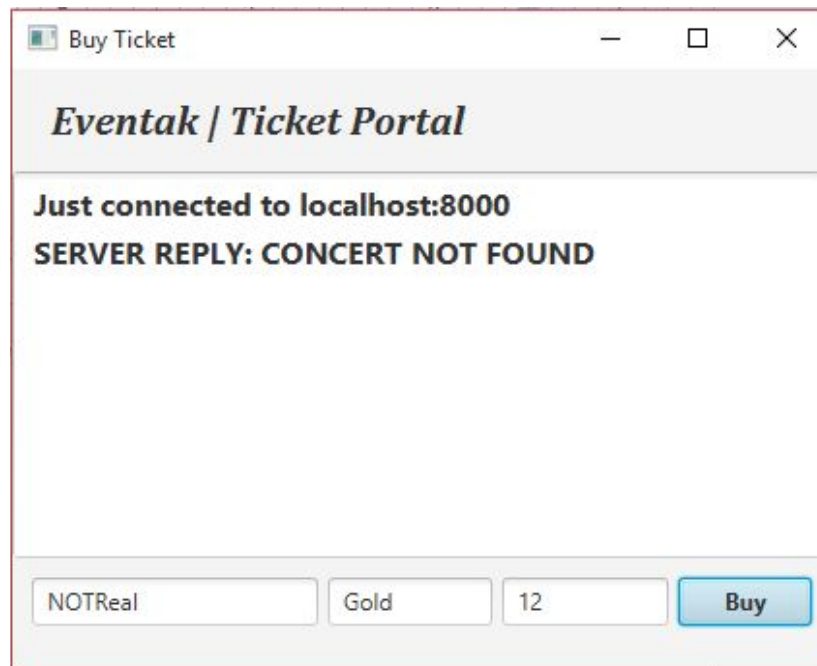


Initial state of client.

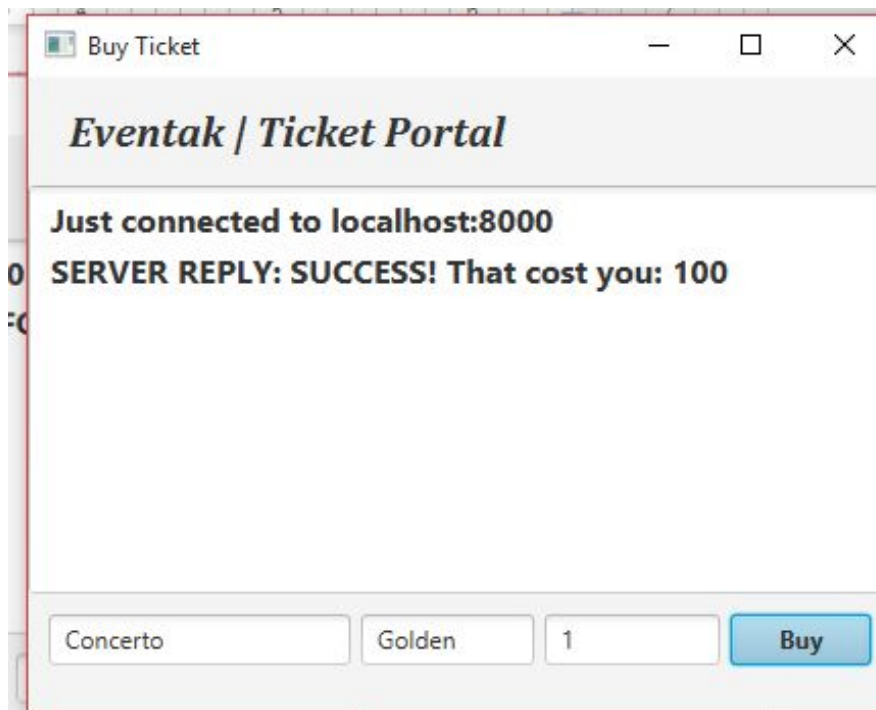
```
Command Prompt - java -jar eventak-server.jar
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\BSilence>cd Desktop/eventak
C:\Users\BSilence\Desktop\Eventak>java -jar eventak-server.jar
EVENTAK SEURER HAS STARTED
Waiting for client on port 8000...
Just connected to /127.0.0.1:21214
NOTReal,Gold,12
Waiting for client on port 8000...
```

Server after 1 request.



Client after 1 request (failed).



Client after 1 request (success).