PumpkinBox Game Platform

Rami Awar - 201600300, Karl Hayek - 201601287
American University of Beirut
Beirut, Lebanon
rba13@mail.aub.edu, kch05@mail.aub.edu

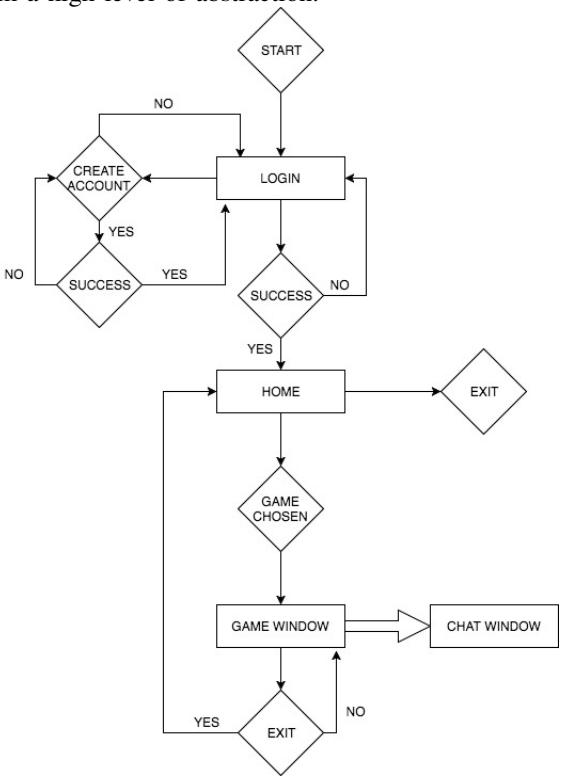
I. Introduction

A. Problem Statement

The goal behind this application is connecting a client to the PumpkinBox server, allowing them to connect with their friends, playing games or chatting. This is done with maximum security in mind to protect user privacy, and maximum scalability in case there was a plan to take this to the market at some point. (As time progressed, emphasis on security and scalability slightly declined due to time constraints)

II. Application Flow

The following is a flow diagram describing our application flow from a high level of abstraction.



Each element will be further elaborated after the neccessary Server/Client classes are discussed so as to be able to reference methods used to accomplish various tasks.

A. Functionality

Our application provides the user with the ability to signup and login first and foremost. During signup, we have implemented a feature validation engine in regex, that provides visual feedback to the user while entering information. After logging in, the client application is passed an authentication token to authenticate with the server during future transactions.

After logging in, the user is sent to a home page, on which a live online friends list, buttons to view personal profile, add friends, check friend requests, and access game hubs are displayed. Clicking on a friend name in the list opens up a chat window that allows the user to send a message to another user. There is no chat history implemented yet, although the foundation is laid out; the messages are all stored in the database, with the corresponding dates and a flag to check whether or not they were received and read.

The requests screen displays incoming friend requests, with an option to accept or reject each. The add friend screen allows the user to add friends by username (email). We chose the email identifier approach for simplicity; implementing a custom username based approach would require more complicated queries relative to our current implementation. The my profile button takes the user to the personal profile screen which consists of 2 tabs, one for viewing recent game activity, friends list, and total experience level. The second tab displays a modifiable friends list, modifiable user info, and the same experience level progress bar.

Finally, the gamehub includes some bar charts to view global and local stats concerning one game, and a tab for global chats and online users. This was not fully implemented due to time constraints. The gamehub offers the user two options: Play offline, or Play with a friend. Choosing to play offline would open a game window to play a match against an AI. Choosing play against a friend opens a new window that allows the user to choose which friend they want to start a game against.

B. Extra Features

- Validation system: checks for non empty text-fields, strong passwords consisting of characters and digits, and email validation.
- Realtime persistent connections to fetch notifications and objects.
- Artificial intelligence in games (tic tac toe and checkers, although checkers is not fully integrated)
- Security and scalibility oriented design: making use of authentication tokens, sha256 password hashing, ...
-

III. Server

A. Requirements

- The server should be listening for connections indefinitely
- The server must assign a new thread to each client requesting a connection
- The server must communicate safely with the client, encrypting any sensitive information across the connection and decrypting at the endpoints
- The server must connect to the PumpkinBox database
- The server must check the authenticity of each client in each client request by using authentication tokens

B. Implementation

The Server class is meant to be run on a private hosted server running Java 1.8. It has no GUI since no client must have access to the sensitive information present in it such as database credentials and connections to other clients.

The Server class has a ServerSocket and port number member variables for obvious reasons, and an acceptConnections core method. The acceptConnections method instantiates the ServerSocket at the defined port number, and listens in an infinite loop for new connections, assigning and starting a ServerThread for each new connection request.

The ServerThread class is responsible for handling each client connection, parsing requests following a well formatted API, and accessing the database. More information on the API later in the document.

The parseRequest method in ServerThread decodes the client request and executes whatever instructions are needed. For example, for SIGNUP requests, the server hashes the password and stores it in the database along with the entered username(email), firstname, and lastname.

In our main method inside the Server class, we instantiate a new Server and call acceptConnections(). This runs our multi-threaded server indefinitely.

C. Methods

The Server class must be able to perform several functions which are neccessary to the application such as checking the authenticity of users, adding users, editing profile information, fetching friends, fetching new messages, ... What follows is a list of functions that the server performs. More details concerning the technical details of the functions like their input and output data (API) are covered in APPENDIX A.

- The server class must have a method to sign up a user, checking if he already exists and then choosing to add him to the users database or not.
- The server class must have a method to login a user, checking if he already exists, checking the authenticity of his token after confirming that his hashed password matches the one saved in the database.
- The server class must be able to get any incoming friend requests after confirming the authenticity of the active user's authentication token. To avoid redundancy, any request to the server excluding signup and login, must have a user ID along with a valid authentication token. This is to avoid having the user send his credentials upon every request.
- The server class must be able to get the list of friends belonging to a given user. (Offline and online - All friends)
- The server class must have a method to fetch the recent game activity of a certain player.
- The server class must have a method to get the friends list of a friend (not the logged in user) but only after the privacy privileges are checked.
- The server class should be able to get the activity list of a friend (not the logged in user) but only after the privacy privileges are checked.

IV. Client

A. Requirements

- The client should start with a login screen
- The client should be able to communicate with the PumpkinBox server via the designed API
- The client must update the GUI where needed
- The client should encrypt sensitive data before sending to the server, ensuring safe communication
- The client should provide an access token with the API request after logging in, in order for the request to be handled

B. Implementation

The client class contains one function for now, which is the sendLoginData function. This function sends login data to the server, and returns a LoginResponseObject which is an object we created to encapsulate the status code, the server response, and the authentication token. This client function maintains a non-persistent connection and closes the socket after it is done.

Calling this client function is done by accessing this static method (sendLoginData).

V. Chat Server

The chat server plays an essential role in managing all realtime operations such as friend requests and friend request notifications, messages and message notifications, game invitations, ... This server connects persistently to a chat client which is constructed in the home controller class.

VI. Chat Client

A. Requirements

- The client should update the online friends list every second
- The client should be able to send messages to friends
- The client must update the GUI in a live manner
- The client should check for new messages and notifications received every second

VII. Game Server

A. Requirements

- Piping game moves from one user to another.

This is achieved by saving each game move in a MessageObject with a sender_id, receiver_id, game name, and game move.

VIII. Game Client

A. Requirements

- Send game moves to GameServer
- Receive game moves related to current client from GameServer

IX. Database

The database is hosted on "34.206.52.140:3306/pumpkinbox" which is an amazon lightsail private hosting server. The MySQL server at this URL runs continuously, and is accessed by us through SSH through the MySQLWorkbench.

So far, we have only designed the pumpkinbox users table in the following manner:

ID: integer - primary key - autoincrements
firstname: varchar(45)
lastname: varchar(45)
password: varchar(255)
email: varchar(255)
authtoken: varchar(255)
expiration: varchar(255)

X. Games

A. Gamehub

When the user clicks on a game from the home page, a new window that contains the games hub is displayed. This window contains an option to play the selected game with a player - who has to be online - from the users friends list, and another option to play the game offline. Furthermore, the game hub contains the games global standings, the players history of scores in this game, and the global chat for this game, where the user can challenge a player who is online. The statistics part of the game hub show the individual statistics of the player (number of wins, losses and ties) and the global statistics for all players.

B. Games

We have written two games for the project, Tic Tac Toe, and Checkers, and wrote AIs for both, whose intelligences range from medium to high, meaning that it is manageable to beat them, however doing so requires some thinking. We decided to make a game hub for each hub to centralize all of the things related to the game in this hub. Also, we decided to work on no more than two games to polish them, give them good AIs, and make them more fun.

The Tic Tac Toe AI works by deciding its first move according to the first move of the user (or if the AI has the first move, it chooses between a corner cell and the center cell). For the rest of the game the AI makes at each turn: a move that results in a winning combination; if cannot find one then it checks if it can make a move to block a possible winning combination of the opponent; if not then it makes a random move. The Checkers AI works by checking all possible next three moves at every turn, and choosing the best outcome. This is done by building a binary tree of height 4 than contains all possible moves as children. Picking the best move from this decision tree is done using the MinMax algorithm, which picks the move in the binary tree that has the highest assigned score for the AI and the lowest assigned score for its opponent. The Checkers game was more challenging to implement because we made it so that a piece is made a king if it reaches an end of the board, which makes it able to move both up and down. Furthermore, we made it so that if a piece kills a piece of the opposite type and can kill a second one in a row, it is made possible for it to do so.

For the two games, we used non-persistent connections for sending and receiving moves between players. In order to prevent the GUI from stalling while waiting for the response from the server, we run a thread dedicated to the move sending and receiving functions. These functions initialize a connection, send/receive a move with an ID that identifies the sender and another that identifies the user, and then close the connection.

XI. API

We have constructed an API for the client and server to communicate efficiently. Each request made by the client to the server follows a certain format: VERB - SECRET - CONTENT

VERB is the action type to be performed by the server. The different types of VERBs are LOGIN, SIGNUP, UPDATE, GET, and MESSAGE. The parseRequest function takes care of the rest.

SECRET is the username,password encrypted combination in case of LOGIN and SIGNUP requests, or an authentication token which is provided by the server after logging in (in case of regular requests).

CONTENT contains the request content in the case of signing up (firstname, lastname), or the command required in case of UPDATE, GET, and MESSAGE.

We have also implemented a class called CODES that simulates HTTP response codes in a way, in order to make status code handling easier. Each code simply responds to a string. It's a very simple class consisting of public static strings only.

More information available in Appendix A.

XII. Organization

This section will describe how the whole code is organized in terms of packages, classes, fxml files, and css stylsheets. Utility packages are directly under the src folder, like time, security, database, api, validation, etc. All UI components on the other hand are inside a ui package in the src folder. Some of these components are also utilities such as draggable(to make a stage draggable), and icons(to import icons from font-awesome libraries with ease). Components relating to scene are in their

own packages, but only the main scene which is the login scene or starting point of our program is located directly under the ui package. Each scene from an organizational perspective is divided into a main class (for testing purposes only so we don't have to navigate the app starting from login to reach the scene we are working on), a controller class, an fxml file and possible a css stylesheet. The controller class is responsible for handling all ui functionalities, like what clicking a button does or filling a text field does and so on. It is from this class that we send requests to the server via our API for example, after pressing the login button. The fxml files reference this controller in their root element as an fx:controller attribute (XML attribute that is). CSS stylesheets are either imported from the previous controller class when loading the new scene or referenced inside an XML attribute (Depending on number of references in a certain stylesheet).

XIII. Suggested Schedule

We plan on finishing the main gui, friends listing and functionality, messaging and chat handling, and some games for testing by the end of this week. After that we plan to add the security layer which encrypts outgoing and incoming connections and decrypts at the endpoint.

XIV. Remaining Tasks

What remains is making a user-friends table, implementing the friends system, a private messaging interface between users, implementing the multiplayer functionality in games, implementing a more secure communication protocol by encrypting and decrypting using a private key available at the server and the client, designing what remains of our client/server API, and taking the whole project online by hosting the server on our private server in addition to the MySQL server.

XV. Division of Tasks

Rami took care of the database, gui development, server-client side development and API development.
Karl took care of the game development and integrating the games with the gui.

XVI. Future Plans

We plan on making the gui more modular, in ways such as making each game preview item an fxml element in its own with its own properties, thus making is spawnable from code. We also plan on integrating a tabs menu that keeps friend chats in the main stage and not in a seperate window.

XVII. Running the program

First, we must run the Server class, then the ChatServer class, then GameServer class. After that, you must make sure that you are connected to the internet in order to access our publicly hosted database. After that, running the main class launches the program.

XVIII. Appendix A

A. API Documentation - Static Requests

- LOGIN:

  Client sends "LOGIN username—password" - Server returns 2 string objects - The first is CODE.OK or CODE.NOT_FOUND, and the second is the authentication token or CODE.NOT_FOUND. The second redundant NOT_FOUND code is sent for consistency. Sometimes redundancy is better. CODE.OK indicates that the user is found, and the authentication token must be used by the user in subsequent requests to access the server.

- SIGNUP:

  Client sends "SIGNUP username—password firstname—lastname" - Server returns 1 string object: CODE.OK or CODE.INSERTION_ERROR or CODE.ALREADY_EXISTS, the first indicates that insertion into the database has been successful, while the second indicates otherwise, and the third indicates that this username(email) already exists. We handle each case accordingly.

- Rest will be implemented during this week.

B. ChatServer/Client API Documentation

- NOTIFICATION:

  Chat server sends a notification to the client which needs to be added to the client notification queue. This notification queue in turn is linked with a scheduled thread that updates the client gui with a notification when a message is received.

- MESSAGE:

  Chat server sends a message to the client, including sender id and message content. The client then decides on which screen to display it, and whether or not to send a notification (If chat window with this specific sender is already open or not).

- Rest will be implemented during this week.

XIX. References

1) Oracle JavaFX documentation
2) Server-Client Chat Application by Almas Baimagambetov (inspired our implementation of Server/Client classes)
3) JFoenix JavaFX library documentation