UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

LIGHTUP

MULTIPLAYER DRAW-AND-GUESS GAME

GROUP PROJECT

Supervisor: Dr. TRAN Giang Son

Submitted by Team LightUp

PHAM Gia Phuc	**---**	**BI10-138**
TRAN Dac Minh	**---**	**BI10-113**
DO Hoang Quan	**---**	**BI10-147**
DANG Gia Linh	**---**	**BI10-101**
PHAM Duc Thang	**---**	**BI10-159**

FEBRUARY 9, 2022
TEAM LIGHTUP

Contents

Acknowledgements

Like many new technologies, digital gaming started as something out of the world. Since the first game appeared in 1962, video games existed almost entirely as novelties passed around by programmers and technicians with access to computers, primarily at research institutions and large companies. The history of video games transitioned into a new era early in the decade, however, with the rise of the commercial video game industry.

Digital games have become a remarkable cultural phenomenon in the last ten years. The casual games sector especially has been growing rapidly in the last few years. A casual game is a video game targeted at a mass market audience, as opposed to a hardcore game, which is targeted at hobbyist gamers. Casual games may exhibit any type of gameplay and genre. They generally involve simpler rules, shorter sessions, and require less learned skill. They don't expect familiarity with a standard set of mechanics, controls, and tropes.

Most casual games have:

Fun, simple gameplay that is easy to understand

Simple user interface, operated with a mobile phone tap-and-swipe interface or a one-button mouse interface

Short sessions, so a game can be played during work breaks, while on public transportation, or while waiting in a queue anywhere

Often, familiar visual elements, like playing cards or a Match 3 grid of objects

Aiming to give people a hilarious, funny and memorable experience after the hard-working times, we LightUp team have worked on developing the LightUp – a multiplayer draw-and-guess game on the website platform, which is really easy for anyone with any electrical devices such as smartphone, tablet and computer, etc. who want to have the joyful times.

Specially, we would like to express our special thanks of gratitude to our teacher Dr. TRAN Giang Son and Dr. NGHIEM Thi Phuong who gave us the opportunity to do this wonderful project on the topic Multiplayer Draw and Guess game, which also helped us in enriching experiences.

- Team LightUp -

Glossary

Node.js	is an open-source, cross-platform runtime environment that allows developers to create server-side and networking applications. Node.js applications are written in JavaScript and provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js
Npm (Node Package Manager)	is an online repository for the publishing of open-source Node.js projects and a command line utility for installing packages, managing versions and dependencies
WebSocket	WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. WebSocket is distinct from HTTP
Server	waits for client request messages, processes them when they arrive, and responds to the web browser with an HTTP response message (e.g., "HTTP/1.1 200 OK" for success, "HTTP/1.1 401 Unauthorized", … etc.). A response can be many things, but the most common response from and API is JSON (JavaScript Object Notation) file
Client	is everything in a web application that is displayed or takes place on the end user device. It also sends requests (like GET, POST, DELETE, UPDATE, … etc.) to the server
JSON	JavaScript Object Notation is a standard way of formatting data using syntax from JavaScript
API (Application Programming Interface)	Code that allows a client to interact with a server
HTML (HyperText Markup Language)	is the standard markup language for documents designed to be displayed in a web browser.
CSS (Cascading Style Sheets)	is a style sheet language used for describing the presentation of a document written in a markup language such as HTML
JavaScript	is a high-level, often just-in-time compiled language that conforms to the ECMAScript standard.[14] It has dynamic typing, prototype-based object-orientation, and first-class functions.
PHP (Hypertext Preprocessor)	is a general-purpose scripting language geared towards web development

SQL (Structured Query Language)	is a domain-specific language used in programming and designed for managing data held in a relational database management system (RDBMS), or for stream processing in a relational data stream management system (RDSMS)
Firebase	Is a platform developed by Google for creating mobile and web applications It was originally an independent company founded in 2011.
URL (Uniform Resource Locator)	is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it
HTTP	HyperText Transfer Protocol
protocol	is an established set of rules that determine how data is transmitted between different devices in the same network
network module	is a software module that implements a specific function in a network stack, such as a data link interface, a transport protocol, or a network application
UML (Unified Modeling Language)	is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.
UI	User Interface
`var port = 8000;`	Line(s) of code to handle function(s) in the program
Event name	Some explanations of features in libraries

List of Figures

Introduction

1.1. Context and motivation

LightUp is a fun and exciting drawing game that we all know and love. It is a perfect game if you want to have fun with your friends and family. It is an Online PVP game, also a Cross-Platform Multiplayer game. Furthermore, it has various insane characters, and simply adding your own packages with words allowed us to spend our free time to enjoy it. Our app, Light up is inspired by “skribbl.io” but a better upgraded of it

With our learning experience, we will build a multiplayer game using WebSocket. WebSocket will enable us to create event-based server-client architecture. The messages are passed between all connected browsers instantly. We will combine the Canvas drawing, JSON data packing, and several techniques learned in the university to build a draw and guess game.

1.2. Objective

Our objective for this project is to develop and preliminary design a game with a purpose that with potential to build a dataset for drawing based image retrieval field. To achieve this, we:

- Develop a user-friendly interface to allow users to easily color sketches.
- Propose an entertaining game, which employs a drawing-based image retrieval paradigm in the gameplay.

1.3. Thesis Structure

The rest of the thesis will be structured as the following sections:

- Section 1: Introduction about project's objective.
- Section 2: Methodology.
- Section 3: Summarize the results and talk about future improvement.

Materials and Methods

2.1. Functional requirements

- Users can join a room to play the game
- Users can choose a word then scribble to describe it
- Users can guess what another user is drawing
- Customize character
- Login to save point
- Set time and rounds

2.2. System analysis and design

A. Use Case Diagram

A UML use case diagram is the primary form of system/software requirements for a new software program underdeveloped.

These use cases specify the expected behavior of a Player, the main user of the app and an Admin, who have responsibilities for maintaining the app.

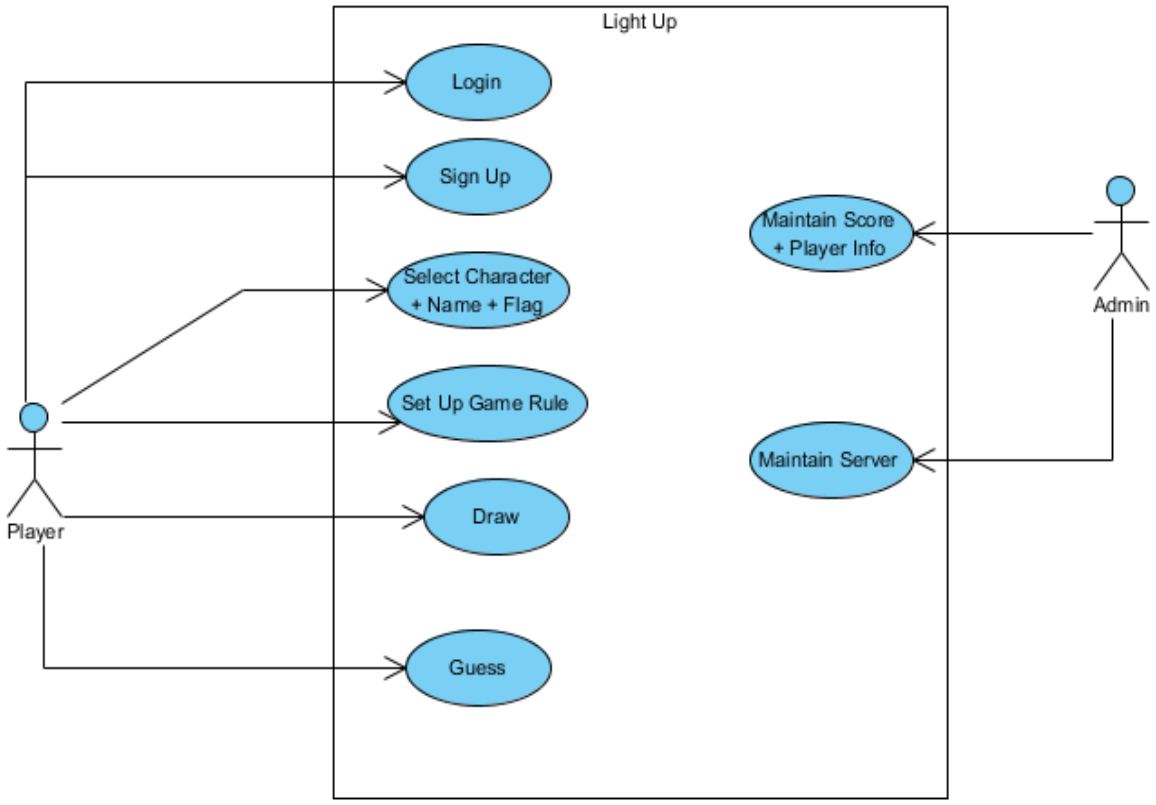

Figure 1. Use-Case diagram

B. Sequence Diagram

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario. The following diagrams describe the most important functions in this project:

2.2.1 Sign Up

Brief Description

This Use-case describes how a Player sign-up for a new account.

Flow of Events

Basic Flow

This use case starts when the player wishes to sign-up.

a) The Player fills in the sign-up form.
b) The form will check if the input.
c) The server checks if an account exists, based on the Inputs.

Alternative Flows

Invalid Input

If invalid input format, the sign-up form will show error, Player can then input again or cancel, at which the use case ends.

Account Exists

If in the Basic Flow, the Player enters an existing account the system displays an error message. The Player can choose to either return to the beginning of the Basic Flow or cancel the sign-up, at which point the use case ends.

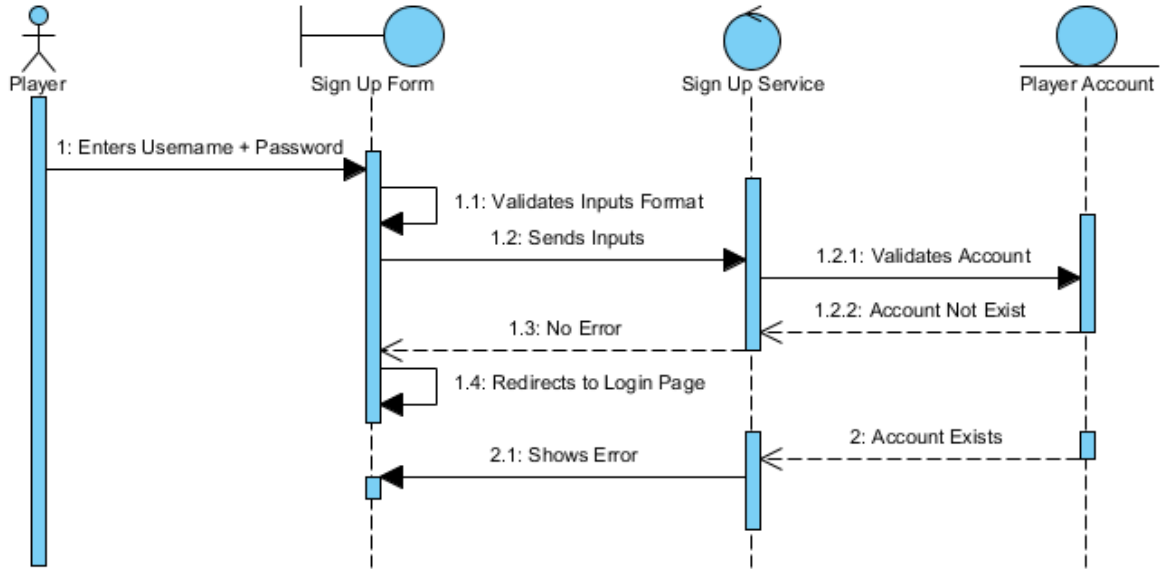

Figure 2.1. Sign Up sequence diagram

2.2.2 Sign In

Brief Description

This Use-case describes how a Player logs in to the App.

Flow of Events

Basic Flow

This use case starts when the Player wishes to sign-in to the Web App.

a) The Player enters his/her Username and password.

b) The server validates the entered username and password based on the Player Account database and logs the Player into the app.

Alternative Flows

Invalid Username/Password

If in the Basic Flow, the Player enters an invalid Username and/or password, the Login form displays an error message. The Player can choose to either return to the beginning of the Basic Flow or cancel the log-in (play as guest), at which point the use case ends.

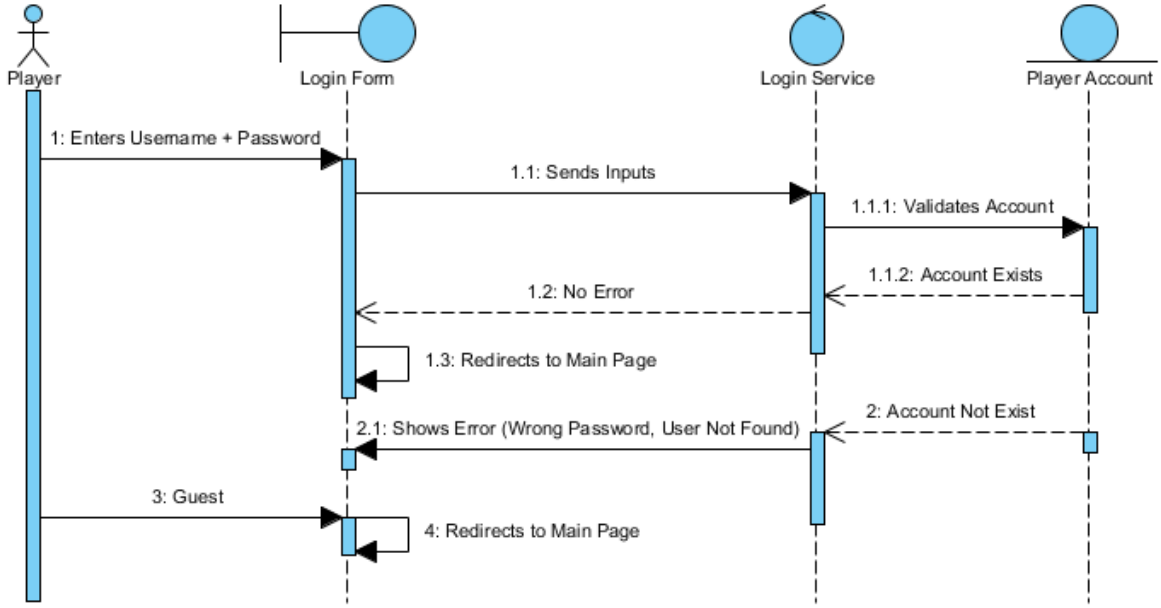

Figure 2.2. Sign In sequence diagram

2.2.3 Select Character, Name and Flag

Brief Description

This Use-case describes how a Player customize Character + Name + National Flag.

Flow of Events

Basic Flow

This use case starts when the Player is redirected to the main page (index).

a) The Player chooses his/her character, enters the name, chooses the flag.
b) The server added player info to the firebase database when the Player clicked play button.

Alternative Flows

Player Logged In

If in the Basic Flow, the Player logged in with his/her account, the page will initialize player info with his/her username and national flag.

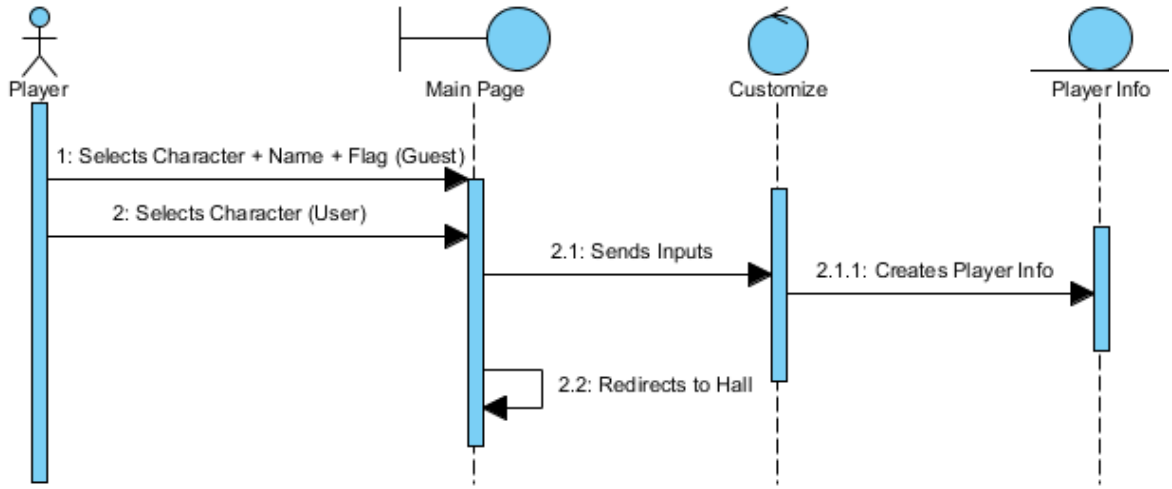

Figure 2.3. Select Character, Name and Flag sequence diagram

2.2.4 Set Game Rule

Brief Description

This Use-case describes how a Player setup rules before starting a game.

Flow of Events

Basic Flow

This use case starts when the Player is redirected to the Hall

a) The First Player to enter the Hall will set up rules for the game (time limit, rounds or add custom words).
b) The server added game rule info to the firebase database when the Player clicked the start game button.

Alternative Flows

None.

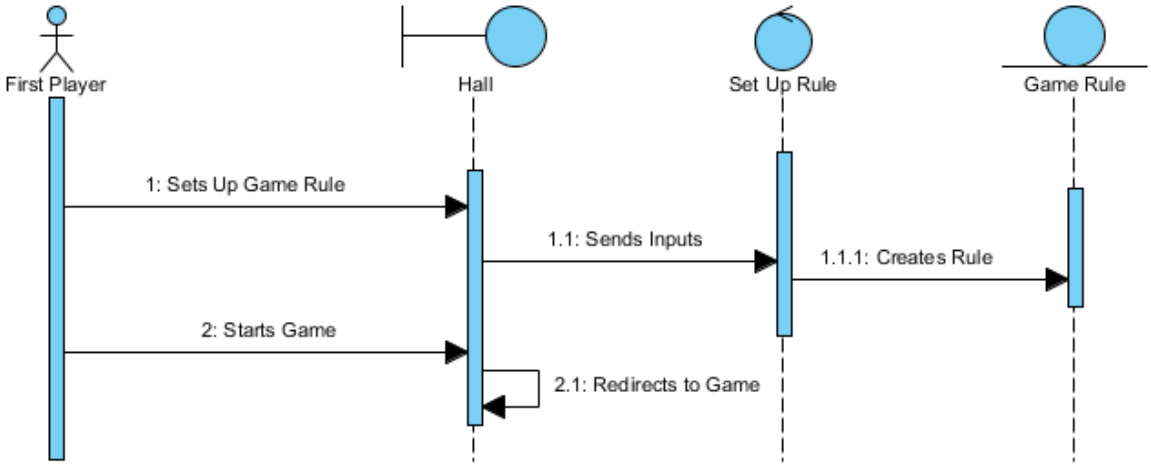

Figure 2.4. Set game rule sequence diagram

2.2.5 Draw and Guess

Brief Description

This Use-case describes how the Players play the game.

Flow of Events

Basic Flow

This use case starts when the Players are redirected to the Room (Game Room).

a) The Drawer chooses between 3 random generated words to draw each round.

b) The other Players guess the word through the chat, if one Player guesses the word right, that Player gets a point, the round will end.

Alternative Flows

Nobody Guess It Right

No one gets a point, the round will end when it hits time limit.

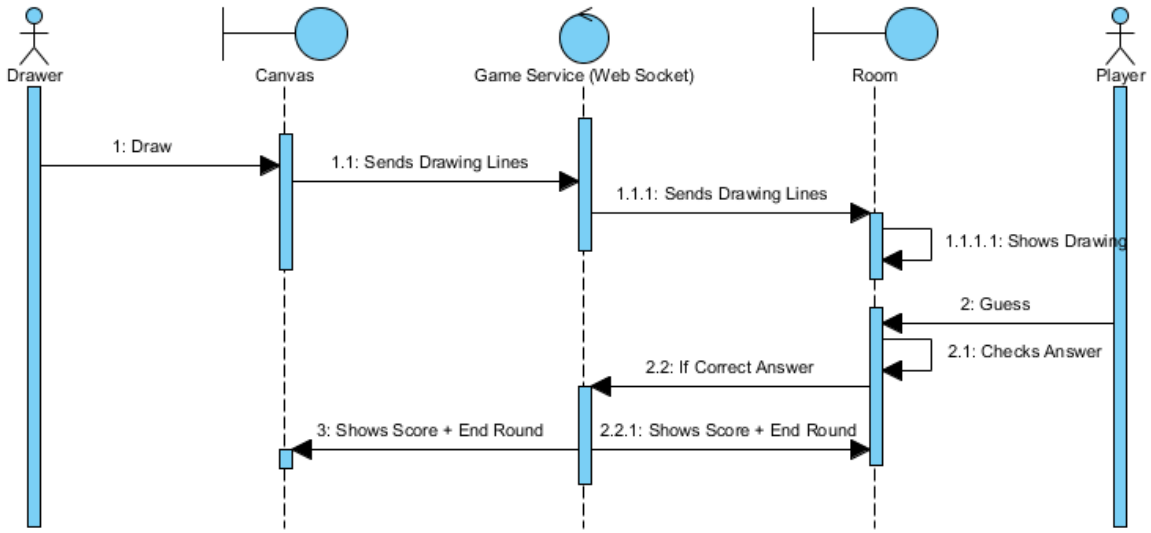

Figure 2.5. Draw and Guess sequence diagram

C. Class Diagram

A class diagram models the static structure of a system. It shows relationships between classes, objects, attributes, and operations.

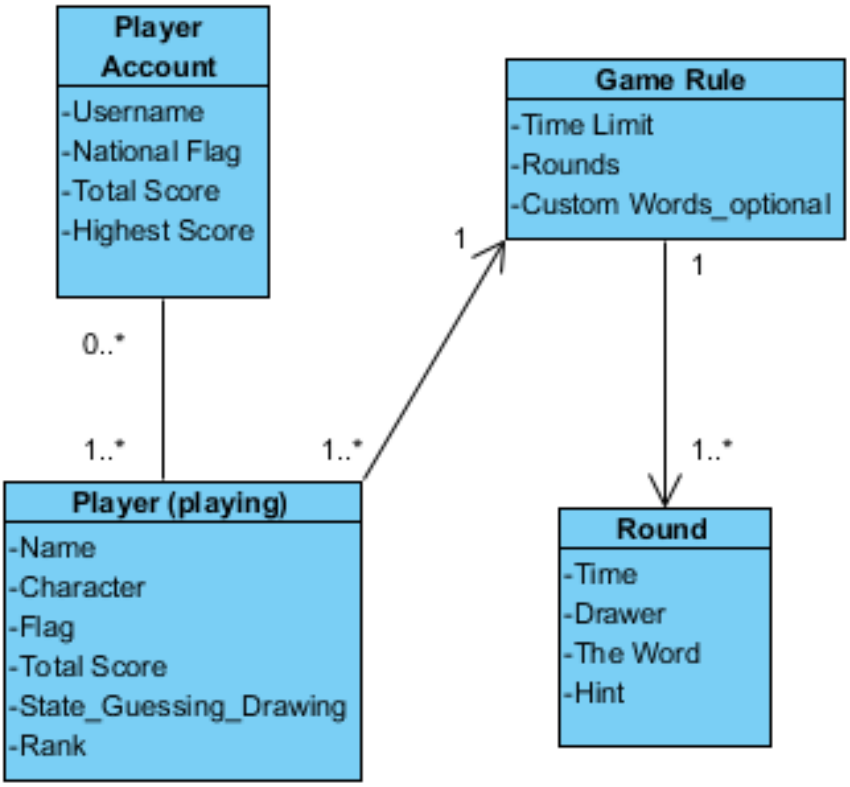

Figure 3. Class diagram

The Diagram shows the relationships between objects in the system.

Player Info object is created when the data object is retrieved from the server or when data is input from Guest. The Data object will contain Username, Password and National Flag of the Player Account, in case the Player logged in. Then Player Info will be initialized with total score = 0, rank = 1 and state = 'Guessing' apart from the Username, Password, and Flag. These 3 attributes will be updated constantly during the game process.

At the end of the game, the app will add Total Score of the Player Info to the Saved Total Score and compare between the new Highest Score to the Saved Total Score to update Player Account stored in the database.

The Player Info object is reset every time a game ends.

Game Rules are updated on the database when a Player sets up the game rules in the Hall page. And a Game Round object data will be generated based on the Game Rules Info and reset each time a round ends. A Game Round object will hold the information about the Word, the Hint and the Role (who the Drawer is) apart from the Time limit retrieved from the Game Rules.

D. Database Diagram

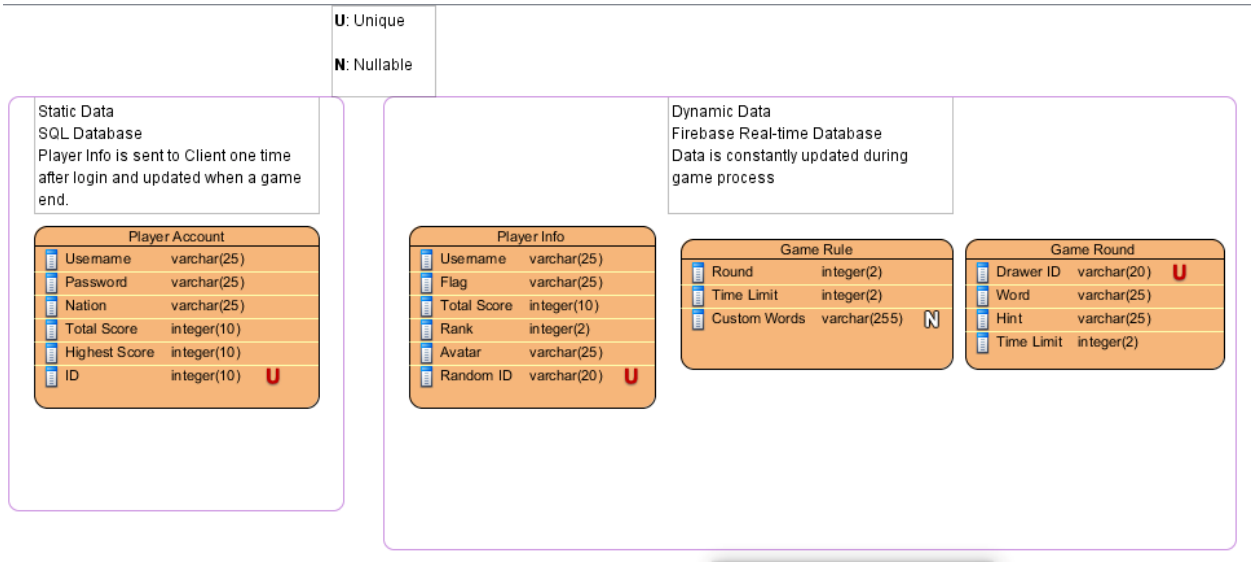

Figure4. Database diagram

E. System Architecture

Our project uses NodeJS to host a server.

Our main purpose is to create an interactive game that can handle many users interacting with the others, for that reason, we use the Web Socket and its services along

with client-side JavaScript functions. In this way, we are able to make and accelerate the interaction between users and make changes in the page without reloading every time.

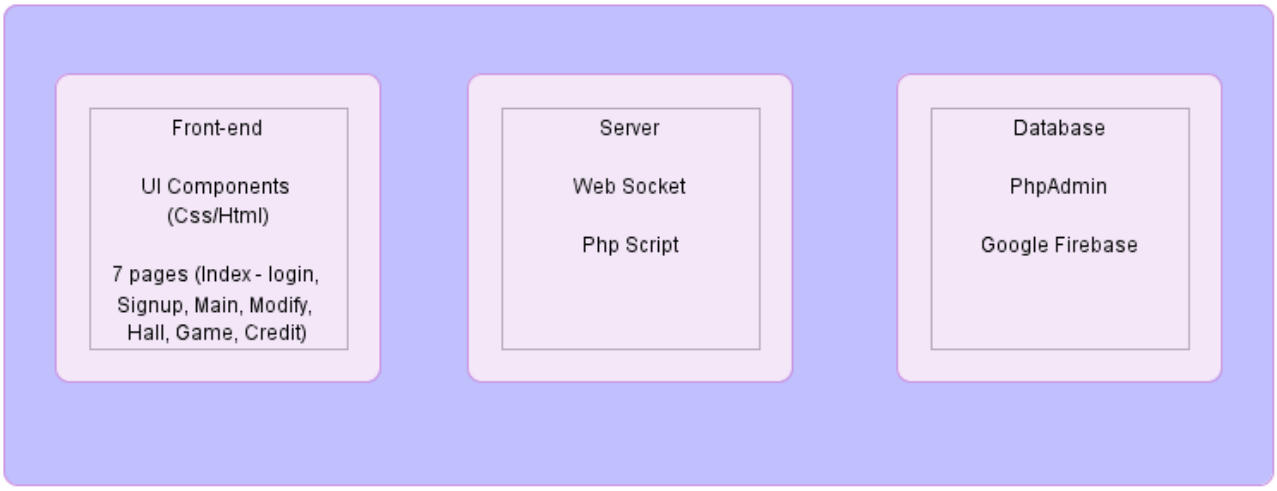

Figure 5. System Architecture

The figure above depicts the components of the system, which is the web app in our case. Players will interact with the app through 7 pages of the website. In normal order, Players will be directed through the Login page – Main page – Hall page and Game page. The Sign-up page can be accessed from the Login page, the Credit and Modify page can be accessed from Main.

We test our program on Local Server (Node Js) because it has some libraries that strongly support Web Socket (both Server and Client libraries).

Php scripts are also used to connect to the Sql database which stores player accounts information.

The Database (Back-end) of the app consists of both Sql Database and Firebase real-time database. We use phpAdmin to manage the Sql Database and Firebase Console to test and keep track of the data communications.

2.3. Implementation

A. Material for development

2.3.1. Node.js

We build server logic on top of this environment. The **WebSocket server** does not necessarily run on Node.js. There are different server-side implementations of the

WebSocket protocol. We chose **Node.js** because it uses **JavaScript**, and we are familiar with it.

2.3.2. WebSocket server

We created a simple server logic that initialized the **WebSocket library** and listened to the connection event. In Node.JS, different functions are packed into modules. When we need functionality in a specific module, we use require to load it. We load the **WebSocket module** and then initialize the server using the following code in the server logic:

```
var wsServer = require("ws").Server;

var server = new wsServer({ port: port });
```

Since the **ws** module is managed by **npm**, it's installed inside a folder called **node_modules**. When we require a library with only the name, the Node.js runtime looks for that module in the **node_modules** folder. We used **8000**

```
var port = 8000;
```

as the **server's port number**, with which a client connects to this server. We may choose a different port number, but we have to ensure that the chosen port number is not overlapped by other common server services.

```
server.on('connection', function(socket) {

        console.log("A connection established");

});
```

The connection event comes with a socket argument. We will need to store this socket later because we use this object to interact with the connecting client.

B. Front-end

The front-end of the website is built by HTML and CSS. We aim to build a web app that can be played on many devices: Desktop, Laptop, Tablet, Phones, for that reason, CSS is a must to build a responsive web page. To be more specific we set up CSS according to devices width:

Max width > 1200px for Desktops

Max width < 1200px for Laptops

```
@media (max-width: 1199px) {
        .u-section-1 .u-sheet-1 {
                min-height: 772px;
        }
```

Max width < 992px for Tablets

```
    @media (max-width: 991px) {
        .u-section-1 .u-sheet-1 {
                min-height: 1690px;
        }
```

Max width < 768px for Landscape Phones

```
@media (max-width: 767px) {
        .u-section-1 .u-sheet-1 {
                min-height: 2275px;
        }
```

Max width < 576px for Portrait Phones

```
@media (max-width: 575px) {
         .u-section-1 .u-sheet-1 {
                min-height: 2281px;
        }
```

C. Index page

2.3.3. User Interface

We control the layout of the main page by utilizing blocks (<section> and tags).

The main page has 3 layers: the default layer, the flags layer, which will be toggled when you click the flags icon (254 flags, scrollable) and the profile layer which will be toggled when you click the button on the upper right corner of the page, the images below will demonstrate our idea of the layout but not our final version:

Index page by default:

Figure 6.1. Index page by default

The “flags” layer:

Figure 6.2. “Flags” layer

The “Profile” layer:

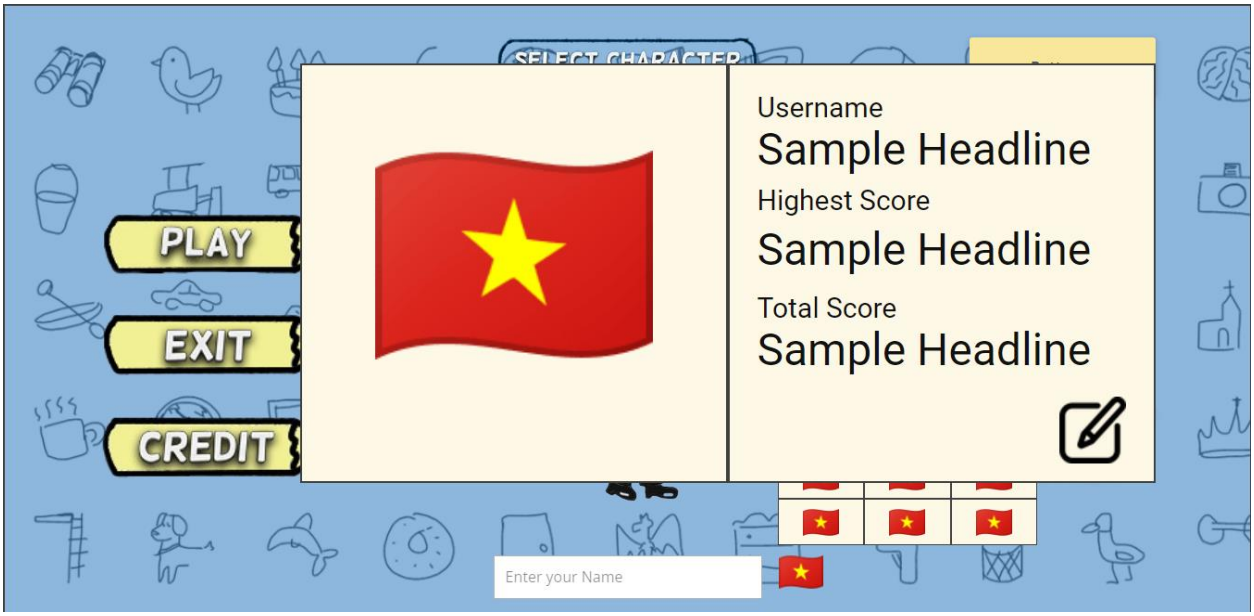

Figure 6.3. “Profile” layer

2.3.4. Functionalities

The main page functionalities can be summarized as customizations. In the main page you can select character (avatar) using the left-right arrows. You can also select

your national flag and your name if you're logged in as a guest. In case you logged in, your flag and name will be initialized based on your account info.

The default info would be Character: Trump, Flag: Viet Nam and Name: blank, in case the name is left blank, the web page will generate a random name.

Some of them can be counted as:

```
"Sky Bully",
"Steel Foil",
"Jawbone",
"Cool Iris",
"Tusk",
"Albatross",
"Bad Bunny",
"Titanium",
"Delivery Boy",
"Dug Funnie",
"Gunsly Bruce Lee",
"Harry Dotter",
"Manchu Man",
"Mighty Mafia",
"Anonymous Names",
"Homely Sharpshooters",
"Investor Relations",
"CobraBite",
"Bleed",
"Fisheye",
"Sidewalk Enforcer",
"Falcon",
"Bullet",
```

Figure 7. Random names

The three buttons on the left: the “play button” will send your character info to the firebase database; “this info” will be used during the playing process. Note that the firebase database will be deleted when the game ends, it’s different from the player account database which is static. The “exit button” will redirect you to google.com, at first, we plan to close the tab when you click the button but that functionality is not supported by modern browsers anymore. The “Credit button” will just redirect you to a page which shows our brief info

2.3.5. Page Content

All of the Images, Drawings and many of the button icons are ourselves designed, and they look cool!

There are 10 characters to choose from.

2.3.6. Data process

The data sent to firebase will be in JSON format, luckily google firebase has some very powerful libraries to support the data manipulation.

Import libraries by using module type JavaScript:

```
import { initializeApp }
import { getDatabase, ref, push, onValue, set }
        from "https://www.gstatic.com/firebasejs/9.6.1/firebase-database.js";
```

After connecting to **Firebase**, we can then add player info to the database.

The **push()** function is used to generate random id for each player on the database.

After pushing player info to the database, we get the id list to decide which player is the first player to enter the room. The first player will be the one who sets up game rules and starts the game in the waiting room.

```
// Initialize Firebase
const app = initializeApp(firebaseConfig);
const db = getDatabase(app);
var reference;

function addPlayer(player, img) {
    reference = ref(db, 'rules/');

    set(reference, {
        name: player,
        avatar: img,
    });
    onValue(ref(db, 'players/'), (snapshot) => {
        var data = Object.keys(snapshot.val());
        console.log(data);
    });
};
```

D. Waiting room

2.3.7. User Interface

The waiting room has one default layer. In the page there are two main sections: the form on the left and the grid layout on the right. As shown in the image below, our game only supports 9 players playing in the same room.

However, the image is not our final version:

Waiting room:

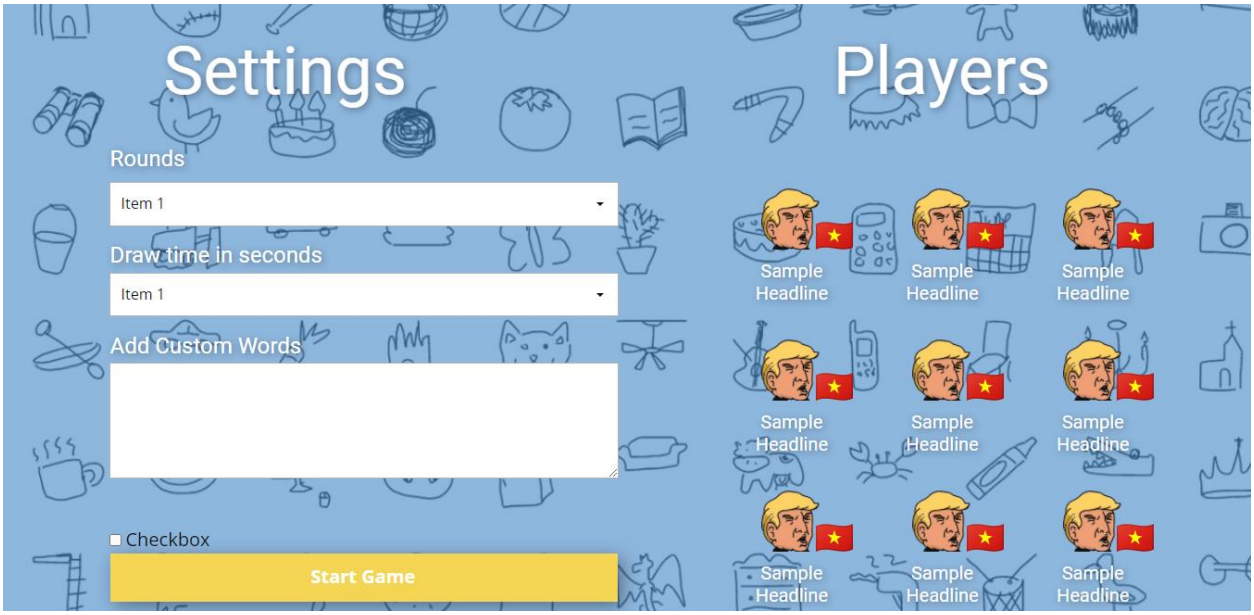

Figure 8. Waiting room

2.3.8. Functionalities

The grid layout on the right of the page shows players info in format **(avatar + flag) + playername**.

The form on the right is used to set up the game rules, the first player can set up the number of play rounds, time limit each round and he/she can also add custom words (optional) to the game dictionary, these words are not surely used in the game. The checkbox is used to toggled Exclusive use of custom words which means the custom words will definitely be used in the game.

The default settings are: Rounds: 3, Draw time: 60 seconds, Custom Words: blank, Check box (toggle): unchecked (off).

Only the first player can set up the game rules, the form is disabled for other players.

All players will be redirected to the game room when the first player hits ‘start’.

2.3.9. Data Process

Player info is retrieved from firebase database when the page loads, and the game rules are also sent to the database in JSON format when the first player start the game:

```
function setupRules(round, time, words, toggle) {
        reference = ref(db, 'rules/');
      set(reference, {
            rounds: round,
            limit: time,
            custom: words,
            exclusive: toggle,
            });
};
```

E. Gaming room

2.3.10. Creating a client that connects to server and get the total connections count

We built a client that established a **WebSocket connection** to the **server** that we built in the last section. The **client** will print any messages that are received from the **server** to the console panel in the **Inspector of Developer Tools.**

- Establishing a WebSocket connection

```
var socket = new WebSocket(wsUrl);
```

The **url argument** is a string with the WebSocket URL. In our example, we run our server locally. Therefore, the URL we used is **ws://127.0.0.1:8000**, where **8000** represents the port number of the server to which we are connecting. It is 8000 because the server was listening to port 8000 when we built the server-side logic.

- About WebSocket server client events, the following table lists

The events we will use to deal with WebSockets:

Event name	Description
Onopen	This is fired when a connection to the server is established
Onmessage	This is fired when any message from the server is received
Onclose	This is fired when the server closed the connection
Onerror	This is fired when there is any error in the connection

- Sending message to all connected browsers

  Once the server gets a new connection event, we send the updated count of the connection to all clients. We just need to call the **sendAll** function in the server instance with a string argument as the message. The following code snippet sends a server message to all connected browsers:

```
var message = "a message from server";
server.sendAll(message);
```

  We defined two classes, **User** and **Room**, in a game.js file, which we use to manage all the connected sockets.

- Defining class and instant instance methods

  In JavaScript, object-oriented programming is done by using functions and prototypes. When we create a room instance by calling new **Room()**, the browser clones all properties and methods in **Room.prototype** to the instance.

- Handling a newly connected user

  For each connected user, we need to interact with them via an events handler. We add the user object into an array for easy management. We need to handle the **onclose event** when a user disconnects. To do this, we remove that user from the array.

- Exporting modules

  After defining our classes in the game.js file, we exported them. By **exporting** them to the **module**, we can **import** them in the other file by using the **require method**, as follows:

```
var User = require('./game').User;
var Room = require('./game').Room;
```

- Sending messages to the client

  WebSocket has the ability to send messages from the server to a user. Normally, the client requests the server and then the server responds. In a socket server, all users are connected, so messages can be triggered and sent in both directions.

```
Room.prototype.send = function(message) {
        for (var i=0, len=this.users.length; i<len; i++) {
                this.users[i].socket.send(message);
```

```
        }
};
```

Then listen the message on the client:

```
// on message event
WebSocketGame.socket.onmessage = function(e) {
        console.log(e.data);
};
```

2.3.11. Build chat function with WebSockets

We want to build a chat room where users can type a message in their respective browsers and send the message to all the connected users instantly.

- Sending a message to the server

  We add an **input text field** for the users to type some text there and send it out, the user input a message and then the text is sent as a message to the WebSocket server. The server will then forward the message to **all connected browsers**. Once a browser receives the message, it displays it in the chat area. In this case, the users are connected to the instant chat room once they load the web page. The server will then print the **received message** in the terminal.

```
$("#send").click(sendMessage);

$("#chat-input").keypress(function(event) {
        if (event.keyCode === 13) {
                sendMessage();
        }
});

function sendMessage() {
        var message = $("#chat-input").val();
        wsGame.socket.send(message);
        $("#chat-input").val("");
}
```

- Sending a message from the client to the server

  In order to **send a message** from the client to the server, we call the following send method in the WebSocket instance:

```
WebSocketGame.socket.send(message);
```

In the following code snippet from our example, we get the message from the input text field and **send it to the server**:

```
var message = $("#chat-input").val();
WebSocketGame.socket.send(message);
```

- Receiving a message on the server side

  On the **server side**, we need to handle the message we just sent from the client. We have an event named message in the connection instance in the **WebSocket node.js library**. We can **listen** to the connection message event to receive a message from each client connection.

- Sending every received message on the server side to create a chat room

  The server could **receive** messages sent from browsers. However, the server does nothing except print the received messages in the terminal. We learned how a server sends the connection count to all the connected clients. We also learned how the client sends a message to the server. Therefore, we add some logic to the server to send the messages out by combining these two techniques to let the server send the received messages to all the connected users.

```
user.socket.on("message", function(message){
        console.log("Receive message from " + user.id + ": " + message);
        // send to all users in the room.
        var msg = "User " + user.id + " said: " + message;
        room.sendAll(msg);
});
```

2.3.12. Marking a shared drawing whiteboard with Canvas and WebSocket

We want a **shared sketchpad**. Anyone can draw something on the sketchpad and all others can view it. We learned how messages are **communicated** between clients and servers. We will go further and send drawing data.

- Building a local drawing sketchpad

  Before we work with data sending and server handling, we focus on making a drawing whiteboard. We use the **Canvas** to build a local **drawing sketchpad**. We created a local drawing pad. This is like a whiteboard where the player can draw in the Canvas by dragging the mouse. However, the drawing data is not sent to the server yet all drawings are only displayed locally.

  Firstly, create a canvas in the html file

```
<canvas id="drawing-pad"></canvas>
```

Then, replace the **wsGame global object** with the following variable at the top of the JavaScript file:

```
var WebSocketGame = {
        // indicates if it is drawing now.
        isDrawing : false,
        // the starting point of the next line drawing.
        startX : 0,
        startY : 0,
}

// canvas context
var canvas = document.getElementById('drawing-pad');
var ctx = canvas.getContext('2d');
```

- Drawing with Canvas

  When we draw something on the computer, it often means that we **click** on the **Canvas** and **drag the mouse (or pen)**. The line is drawn **until** the **mouse button** is **up**. Then, the user clicks on another place and drags again to draw lines. In our example, we have a Boolean flag named **isDrawing** to indicate whether the user is drawing. The **isDrawing flag** is **false** by default.

  When the mouse button is **at a point**, we **turn the flag to true**. When the mouse is moving, we draw a line between the moved point and the last point when the mouse button was. Then, we set the isDrawing flag to false when the mouse button is up. This is how the drawing logic works.

- Sending the drawing to all connected clients

  To go further, it is necessary to **send** the **drawing data** to the server and let the server send them to **all connected players**.

  To achieve it, we have created two constants in game.js file:

```
// Constants
var LINE_SEGMENT = 0;
var CHAT_MESSAGE = 1;
```

After that, the following code is added at the beginning of the **Room.prototype.addUser** method:

```
    this.users.push(user);
    var room = this;
    // tell others that someone joins the room
    var data = {
            dataType: CHAT_MESSAGE,
            sender: "Server",
            message: "Welcome " + user.id + " joining the party. Total connection: " +
this.users.length,
    };
    room.sendAll(JSON.stringify(data));
```

Moreover, **JSON-formatted string** is used for communicating both drawing actions and chat messages with the usage of:

```
    user.socket.on("message", function(message){
            console.log("Receive message from " + user.id + ": " + message);

            // construct the message
            var data = JSON.parse(message);
            if (data.dataType === CHAT_MESSAGE) {
                    // add the sender information into the message data object.
                    data.sender = user.id;
            }

            // send to all clients in room.
            room.sendAll(JSON.stringify(data));
    });
```

on the message event handler.

When handling the message on the **client-side**, we **converted** the **JSON-formatted string** back to a **data object**. If the data is a **chat message**, we then displayed it as the chat history, otherwise, it is being **drawled in the Canvas** by using **parse**:

```
    var data = JSON.parse(e.data);
```

- Defining a data object to communicate between the client and the server

In order to communicate correctly between the server and clients when there is a lot of data packed into one message, we have to define a data object that both the client and server understand.

There are several properties in the data object. The following table lists the properties and why we need them:

Property name	**Reasons**
dataType	This is an important property that helps us to understand the entire data. The data is either a chat message or drawing line segment data
sender	If the data is a chat message, the client needs to know who sent the message.
message	If the data is a chat message, the client needs to know who sent the message.
startX	When the data type is a drawing line segment, we include the xy coordinates of the starting point of the line.
startY	
endX	When the data type is a drawing line segment, we include the x/y coordinates of the ending point of the line.
endY	

- Packing the drawing into JSON for sending work

  We used the **JSON.stringify function** when we stored a JavaScript object into a **JSON-formatted string** in the local storage. Now, we need to send the data in string format between the server and the client. We use the same method to pack the drawing lines data into an object and send it as a **JSON string**

```
var data = {};
wsGame.socket.send(JSON.stringify(data));
```

- Recreating the drawing after receiving them from other clients

  The **JSON parsing** often comes as a pair of **stringify**. When we receive a message from the server, we have to parse it to the JavaScript object. The following code on the client side parses the data and either updates the chat history or draws a line based on the data.

2.3.13. Building a multiplayer draw-and-guess game

We built an instant **chat room** earlier. Also, we built a **multiuser sketchpad**. Now we **combine** these two techniques and build a **draw-and-guess game.** A draw-and-guess game is a game in which **one player** is **given a word** to draw. All **other players** do not know the word and **guess the word according to the drawing**. The one who draws and who correctly guesses the word earns points.

We need a few more constants to determine different states during the game play.

```
GAME_LOGIC : 2,

// Constant for game logic state

WAITING_TO_START : 0,

GAME_START : 1,

GAME_OVER : 2,

GAME_RESTART : 3,
```

When the client receives a message from the server, it **parses** it and **checks** whether it is a chat message or a line drawing. We have another type of message named **GAME_LOGIC** for handling the game logic. The game logic message contains different data for **different game states**.

Finally, there is one last step in the **client-side logic**. We want to **restart** the game by sending a restart signal to the server. At the same time, we clear the drawing and chat history. To do this, we add the lines inside the **websocket.js** file:

```
// restart button

$("#restart").hide();

$("#restart").click(function() {

    canvas.width = canvas.width;

        $("#chat-history").html("");

        $("#chat-history").append("<li>Restarting Game.</li>");


        // pack the restart message into an object.

        var data = {};

        data.dataType = WebSocketGame.GAME_LOGIC;
```

```
            data.gameState = WebSocketGame.GAME_RESTART;
            WebSocketGame.socket.send(JSON.stringify(data));
            $("#restart").hide();
    });
```

The server side was just in charge of sending any incoming message to all connected browsers. This is not enough for a multiplayer game. The server will act as the game master that controls the game flow and determination of the winning condition. We extend the **Room** class with **gameRoom** that can handle the game flow in game.js file.

This is the constructor function of a new class called **gameRoom**, which initializes **game logic**:

```
    function gameRoom() {
        // current turn
        this.playerTurn = 0;
        this.wordList = ["one"];
        this.currentAnswer = undefined;
        this.currentGameState = WAITING_TO_START;

        // send game state to all players
        var gameLogicData ={
                        dataType: GAME_LOGIC,
                           gameState: WAITING_TO_START
        };
        console.log(this);
        this.sendAll(JSON.stringify(gameLogicData));
    };
```

This way, the gameRoom keeps the original **room's addUser** function, and we can add extra logic and some features like randomly defining a drawer or a word in playing time.

At last, we export the gameRoom class so that other files, such as server.js, can access the gameRoom class:

```
module.exports.gameRoom = gameRoom;
```

In addition, in **server.js**, we must call our new **gameRoom constructor** instead of the generic Room one.

* Game flow controlling:

There are several states in the game flow. Before the game starts, the connected players are waiting for the game to start. Once there are enough connections for the multiplayer game, the server sends a game logic message to all the players to inform them of the start of the game.

When the game is over, the server sends a game over state to all the players. Then, the game finishes and the game logic halts until any player clicks on the restart button. Once the restart button is clicked, the client sends a game restart state to the server instructing the server to improve the game. Then, the game starts again.

2.3.14. Extra functionalities

1. **Styles**

   Css is used to improve its visual outlook with some decorative images.

2. **Better drawing**

   To give user more choices when playing with the game, we added some features which contain:

- Color of drawing lines

  The game provides 6 constant colors: white, black, red, blue, green, yellow and a color picker to customize the picture

- Thickness of drawing lines

  By default, the thickness of the pen is 2 with the color of black, however the drawer can change the thickness with a wide range: from 1 to 100, this will increase the joy of expressing the imagination, we believe.

- Shape drawing

  Yet, just the normal lines are not enough, therefore, we give users the ability to draw the straight line, circle and polygons. This way, it is easier for the drawer to create the greatest picture in play-time.

3. Bonus features

Since not every time, every drawing lines are correct or fit the imagination of drawer, we also provide the feature of undoing action – drawing line in this case – and the ability to clear erase everything in the canvas pad, thus, it is better for drawer to fix or even reconstruct the imagination of the masterpieces.

Besides, we also added some functions to help stabilize the working process of the game like take and restore the snapshot of the canvas drawing pad.

Result and Discussion

3.1. Result

We have made a casual Web game with the following features:

- Able to choose the names and Character for gaming room
- Player can set the rule as well as the list of playing words
- Player can change the way of drawing with multiple choices of color and shape drawing
- Connecting the server from a browser
- Chatting and guessing in playroom

3.2. User Interface

These following images are screenshots of our LightUp game running on a Dell laptop 18.5 inches with Chrome browser.

A. Index page

Figure 9.1. Log In

Figure 9.2. Log in

Figure 9.3. Index page by default

Figure 9.4. Select character and flags

Figure 9.5. User profile

Figure 9.6. Leaderboard

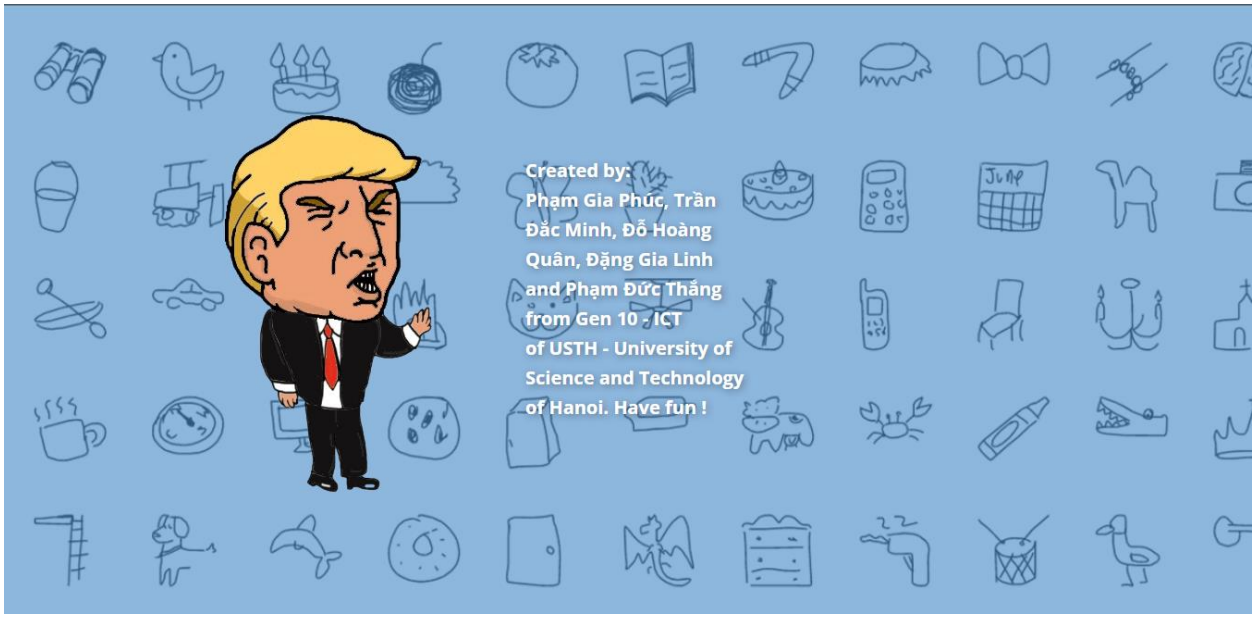

Figure 9.7. Credit button

B. Waiting room

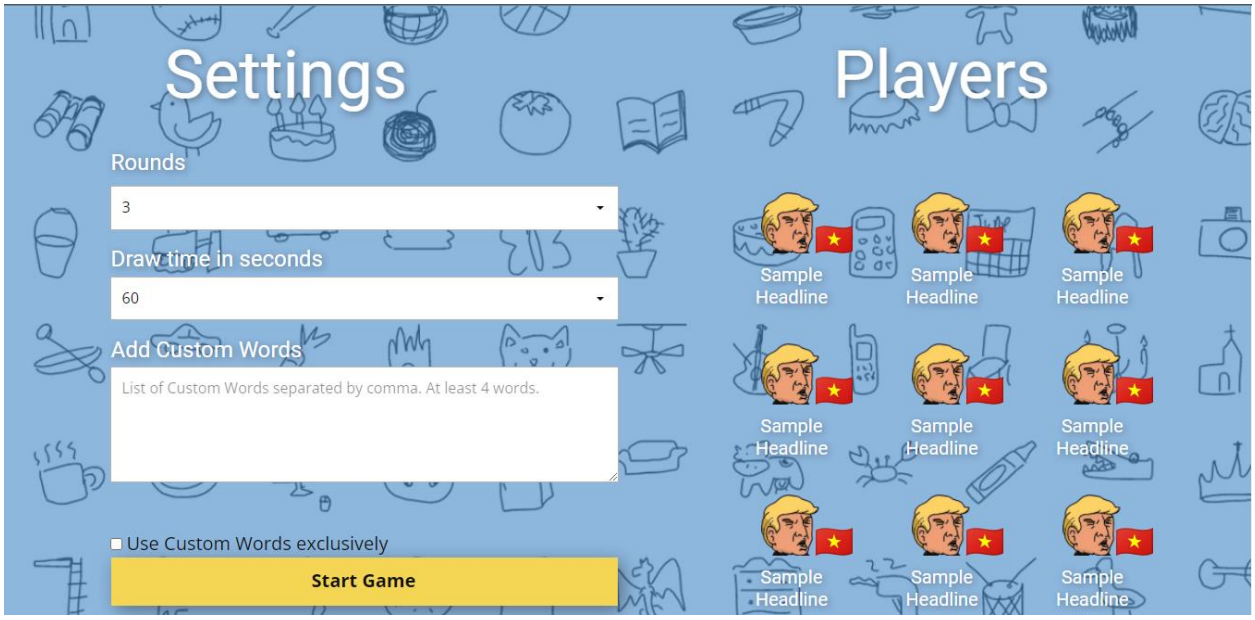

Figure 10. Waiting room

C. Gaming room

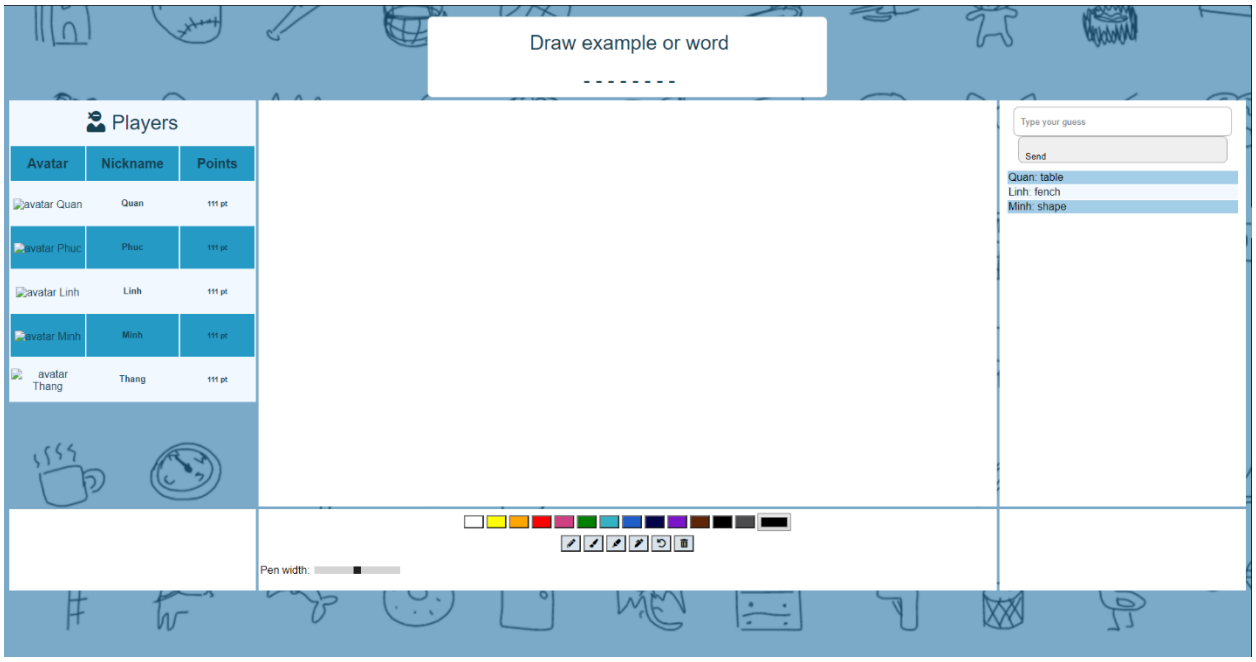

Figure 11. Gaming room

Conclusion and Future developments

4.1. Conclusion

In conclusion, we have built a multiplayer game with the help of WebSocket. WebSocket enables us to create event-based server-client architecture. The messages are passed between all connected browsers instantly. We have combined the Canvas drawing, JSON data packing, and several techniques to build the draw-and-guess game.

The main components of the application have been implemented:

- **Name and Character choosing function**: Player can choose the name, character and avatar which are shown in the game
- **Rule set function**: Player can set custom words for the word list when playing to have the best experiences
- **Drawing and chatting function**: Players can draw in multiple ways, share the masterpieces as well as chatting with others.

4.2. Future developments

- Grading system
- More stable in play time
- Optimize and update the game
- Update database to store information of the game
- Other platforms optimization, such as Mobile platform
- More administrative features like kick somebody out of the room
- More game rooms, custom game rounds, more timing choices
- Sharing options

References

Gaming place:

http://lightup.42web.io/

Game developing documents:

https://www.w3schools.com/html/

https://www.w3schools.com/js/default.asp

https://www.w3schools.com/php/default.asp

https://www.w3schools.com/css/default.asp

https://www.w3schools.com/sql/default.asp

Node.js WebSocket documents:

https://www.w3schools.com/nodejs/nodejs_intro.asp

https://github.com/nodejs/node

https://nodejs.dev/learn

https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

https://spring.io/guides/gs/messaging-stomp-websocket/

Firebase database documents:

https://www.npmjs.com/package/firebase

https://firebase.google.com/docs/web/setup

https://firebase.googleblog.com/

https://firebase.google.com/firebase

UML diagram tool:

https://www.tutorialspoint.com/uml/uml_standard_diagrams.htm

https://www.visual-paradigm.com/tutorials/