UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI



LIGHTUP

**MULTIPLAYER DRAW-AND-GUESS GAME**

**GROUP PROJECT**

February 9, 2022

TEAM LightUP

https://github.com/Lib3Rt9/LightUp

**Supervisor: Dr. TRAN Giang Son**

**Submitted by Team LightUp**

**PHAM Gia Phuc**

**TRAN Dac Minh**

**DO Hoang Quan**

**DANG Gia Linh**

**PHAM Duc Thang**

**---**

**---**

**---**

**---**

**---**

**BI10-138**

**BI10-113**

**BI10-147**

**BI10-101**

**BI10-159**

Contents

[**Acknowledgements** 3](#_Toc95142203)

[**Glossary** 3](#_Toc95142204)

[**List of Figures** 3](#_Toc95142205)

[**Introduction** 3](#_Toc95142206)

[**1.1.** **Context and motivation** 3](#_Toc95142207)

[**1.2.** **Objective** 3](#_Toc95142208)

[**1.3.** **Thesis Structure** 4](#_Toc95142209)

[**Materials and Methods** 4](#_Toc95142210)

[**2.1.** **Functional requirements** 4](#_Toc95142211)

[**2.2.** **System analysis and design** 4](#_Toc95142212)

[**2.2.1** **Use Case Diagram** 4](#_Toc95142213)

[**2.2.2** **Sequence Diagram** 4](#_Toc95142214)

[**2.2.3** **Class Diagram** 4](#_Toc95142215)

[**2.2.4** **System Architecture** 4](#_Toc95142216)

[**2.3.** **Implementation** 4](#_Toc95142217)

[**2.3.1** **Installing material for development** 4](#_Toc95142218)

[**2.3.2** **Index page** 5](#_Toc95142219)

[**2.3.3** **Waiting room** 5](#_Toc95142220)

[**2.3.4** **Gaming room** 6](#_Toc95142221)

[**Result and Discussion** 10](#_Toc95142222)

[**3.1.** **Result** 10](#_Toc95142223)

[**3.2.** **Discussion** 10](#_Toc95142224)

[**3.3.** **User Interface** 10](#_Toc95142225)

[**Conclusion and Future developments** 10](#_Toc95142226)

[**4.1.** **Conclusion** 10](#_Toc95142227)

[**4.2.** **Development** 10](#_Toc95142228)

[**References** 10](#_Toc95142229)

# **Acknowledgements**

We would like to express our special thanks of gratitude to our teacher Dr. Tran Giang Son who gave us the opportunity to do this wonderful project on the topic Multiplayer Draw and Guess game, which also helped us in enriching experience.

# **Glossary**

* **Node.js:** is an open-source, cross-platform runtime environment that allows developer 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.

# **List of Figures**

# **Introduction**

## **Context and motivation**

With our learning experience, we will build a multiplayer game using WebSockets. WebSockets 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.

## **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.

## **Thesis Structure**

The rest of the thesis will be structured as 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**

## **Functional requirements**

* User can join a room to play the game.
* User can choose a word then scribble to describe it.
* User can guess what another user is drawing.

## **System analysis and design**

### **Use Case Diagram**

* + 1. **A**
       1. **A**
    2. **B**
       1. **B**

### **Sequence Diagram**

* + 1. **A**
       1. **A**
    2. **B**
       1. **B**

### **Class Diagram**

* + 1. **A**
       1. **A**
    2. **B**
       1. **B**

### **System Architecture**

* + 1. **A**
       1. **A**
    2. **B**
       1. **B**

## **Implementation**

### **Installing material for development**

* + 1. **Install 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 **WebSockets protocol**. We chose **Node.js** because it uses **JavaScript**, and we are familiar with it.

* + 1. **Creating a Websocket server**

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

<code>

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**

<code>

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.

<code>

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.

* + 1. **Websocket**
    2. **Please don’t delete this line to keep the layout**

### **Index page**

* + 1. (**contains information for index page)**
    2. **Identify the need for data**
    3. **Data resource**
    4. **Data process: Edit and convert data**
    5. **Please don’t delete this line to keep the layout**

### **Waiting room**

* + 1. **Break Down Steps**
    2. **Server Side**
       1. Host a local Server with NodeJS and Express
       2. Embedded a map with Google Maps JavaScript API
       3. Geocoding Service
       4. Google Cloud Storage
    3. **Please don’t delete this line to keep the layout**

### **Gaming room**

* + 1. **Creating a client that connects to server and get the total connections count**

We build a client that established a **WebSockets 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

<code>

var socket = new WebSocket(wsUrl);

The url argument is a string with the WebSockets 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 closes 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:

<code>

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

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

* Sending messages to the client

WebSockets have 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);

};

* + - 1. **Host a local server with NodeJS and Websocket**
         1. Before we go further about hosting a local Sever, we would like to briefly define a few important terms and concepts:
      2. **Create a Websocket server**
    1. **Building chatting 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.

<code>

$("#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:

<code>

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 room.

var msg = "User " + user.id + " said: " + message;

room.sendAll(msg);

});

* + 1. **Marking a shared drawing whiteboard with Canvas and WebSockets**

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.

<code>

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 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. We also add five buttons with 5 different colors. The player can choose the color when drawing.

* Sending the drawing to all connected clients
* Defining a data object to communicate between the client and the server
* Packing the drawing into JSON for sending work
* Recreating the drawing after receiving them from other clients

# **Result and Discussion**

## **Result**

## **Discussion**

## **User Interface**

# **Conclusion and Future developments**

## **Conclusion**

## **Development**

# **References**