Lab 11: *REST API* Frontend

Multiplayer HiLo

REST API, Client

IMPORTANT NOTE: Install Node JS for Backend

Table of Content: *Implementing a REST Client for Multiplayer HiLo*

Related Labs:

- REST API- Server Endpoints | (Backend)

Lab Introduction

Prerequisites

Lab: REST API - Server Endpoints. This is the second lab in the REST API sequence. You must have your Backend service running on localhost and listening on port 3000 to start this lab.

Motivation

Build a multiplayer HiLo game that users play from the browser.

Goal

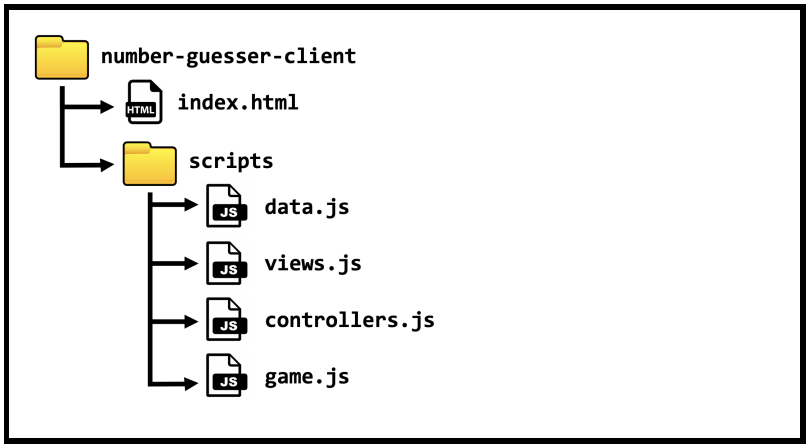
Design & Implement a frontend client for a Number Guessing Game REST API

Learning Objectives

- Async/Await & fetch
- DOM manipulation
- REST Clients
- JSON data
- MVC software architecture
- Frontend app logic (Design & Develop)

Client-side Architecture:

Start this project by making a project folder where all your assets & scripts will be organized. Create all necessary files and folders as illustrated below.



Concepts

REST Architecture

REST architecture is a common approach for backend services to interface with other backend or frontend apps.

- **API:** a collection of pre-implemented functions or methods a developer can import or invoke
- **REST:** REpresentational State Transfer. Refers to using the stateless HTTP protocol to provide access to an API
- **Endpoints:** When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include a URL of a server or serviceI

Async/Await

Asynchronous JavaScript functions implemented using Promises. It invokes the function and does not wait for the result like in default synchronous functions. When the promise is resolved it triggers a callback.

- **fetch:** Common asynchronous function to send an HTTP request from the browser's JavaScript Run Environment to a backend REST API service
- **then:** Common asynchronous callback function that triggers when a promise is resolved

---

JSON

JavaScript Object Notation. Is a format of encapsulating and serializing data for transmitting between one application to another. The REST API backend service will transmit response data as JSON which can be treated as a JavaScript object in the JSRE.

Document Object Model (DOM)

Document Object Model is the JavaScript representation of the browser viewport's HTML elements. The DOM allows us to programmatically render HTML from JS code. This allows apps to overwrite HTML elements and thus build SPAs. (Single Page Apps)

Iteration 0: Design Game Client of REST API

'Approach' → Plan phase

Goal #0: Specify the views, controls, and logic of client app

Approach: Pencil & paper

Before coding a client app, you should first plan out views, controls, and logic you intend to use and how those client interfaces with backend service and how the client interface with the user

---

'Apply' → Do phase

Design Steps

Step 1: Create Mockups of your views.

1	**Number Guessing Game** New Game Join Game	**Main Menu (View)** *Starting point for application* • Option 1: New Game *(goto 2)* • Option 2: Join Game *(goto 3)*
2	**Number Guessing Game** Start: 0 End: 1000 Start	**New Game Menu (View)** *Inputs: min & max values* *Request a new game from REST API* • Option 1: Start *(goto 4)*
3	**Number Guessing Game** Game ID: GameID Join	**Join Game Menu (View)** *Inputs: game id* *Request a get game from REST API* • Option 1: Join *(goto 4) or (goto 1) on error*
4	**Number Guessing Game** **Game ID: GuTOM7tm-** Min: 0 Max: 1000 Submit	**Game (View)** *Inputs: guess* *Request a guess from REST API* • Option 1: Submit *(goto 4) or (goto 5) on win*
5	**Number Guessing Game** **Game ID: GuTOM7tm-** Game Over: You win! Replay Main Menu	**Game Over Menu (View)** *After game ends whether player won/lost* • Option 1: Replay *(goto 4)* • Option 2: Main Menu *(goto 1)*

'Assess' → Test phase

Once you understand the mockups, then start incrementally building the project!

Iteration 1: Display → ' Main Menu' view

'Approach' → Plan phase

Goal #1: Display the guessing game's main menu

Approach: HTML, Javascript

From the mockup, the Main Menu offers two options: create a new game and join an existing games

---

'Apply' → Do phase

Steps

- **Step 1**: index.html → define a view element & import JavaScript scripts
- **Step 2**: views.js → define function that renders HTML into the view element
- **Step 3**: views.js → When the window loads it will invoke the main menu.

Step 1 (HTML): *index.html* → Define body

Import the JavaScript scripts and define a div element for the JavaScript to display the game's views

index.html

```
<body>
   <h1>Number Guessing Game</h1>
   <div id='view'></div>

   <script src='./scripts/data.js'></script>
   <script src='./scripts/views.js'></script>
   <script src='./scripts/game.js'></script>
   <script src='./scripts/controllers.js'></script>
</body>
```

Step 2 (JS): ***views.js*** → mainMenu()

In the views script, define a mainMenu method that displays the HTML options in the view div element.

views.js

```
const mainMenu = function(){
   const view = document.getElementById('view')
   const html =    `<section>
                       <button id='new-game-button'>New Game</button>
                       <button id='join-game-button'>Join Game </button>
                   </section>`
   view.innerHTML = html;
}
```

Step 3 (JS): ***views.js*** → window.onload

The views script should display the main menu when the browser window loads..

views.js

```
window.onload = mainMenu;
```

'Assess' → Test phase

Open the index.html in the browser and the main menu should render.

Main Menu (View)

Iteration 2: Display → 'New Game' view

'Approach' → Plan phase

Goal #2: Setup a controller that launches the 'New Game' view from 'Main Menu' view

Approach: JavaScript EventListeners, DOM manipulation

JavaScript defines functions that overwrite the HTML in the view div and each time a new game view is displayed, set up its buttons with event listeners and callback functions.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: views.js → define a function that renders the 'new game' menu to HTML view
- **Step 2**: controllers.js → define a function that maps callbacks to HTML IDs
- **Step 3**: controllers.js → define a function that adds an event listener to an HTML button
- **Step 4**: views.js → refactor main menu function to add an event listener to button

Step 1 (JS): *views.js* → newGameMenu()

In views, define a newGameMenu function that overwrites the HTML view div.

views.js

```
const newGameMenu = function(){
   const view = document.getElementById('view');
   const html =    `<section>
                       Start: <input id='min-value' type=number value=0>
                       End:   <input id='max-value' type=number value=1000>
                       <button id='start-game-button'>Start</button>
                   </section>`
   view.innerHTML = html;
}
```

Step 2 (JS): *controllers.js* → getCallbacks()

In controllers, define a function that maps callbacks to button IDs.

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu
});
```

Step 3 (JS): ***controllers.js*** → addController()

In controllers, define a function that takes in a variable number of button IDs & sets their event listeners.

controllers.js

```
const addController = function(...buttonIDs){
   const callbacks = getCallbacks();
   for (let id of buttonIDs){
       const button = document.getElementById(id);
       button.addEventListener('click', callbacks[id]);
   }
}
```

Step 4 (JS): ***views.js*** → Refactor: mainMenu()

Refactor mainMenu function so that it adds a controller to the new-game button

views.js

```
const mainMenu = function(){
   const view = document.getElementById('view')
   const html =    `<section>
                       <button id='new-game-button'>New Game</button>
                       <button id='join-game-button'>Join Game </button>
                   </section>`
   view.innerHTML = html;
   addController('new-game-button');
}
```

'Assess' → Test phase

Open the index.html in the browser and the main menu should render.

Click on the 'New Game' button

After click, 'New Game' menu should display

Iteration 3: Fetch Backend → 'New Game' API

'Approach' → Plan phase

Goal #3: Use fetch on the backend REST API to create a new guessing game

Approach: Fetch, Await/Async, REST API

Setup game data & a method that has the backend create a new game & display its response in console.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: data.js → Initialize centralized location for storing game data
- **Step 2**: controllers.js → functions to get the min & max values from inputs
- **Step 3**: game.js → method to request new game from backend API
- **Step 4**: views.js → display response from backend service
- **Step 5**: controllers.js → map callback to button id to create game on backend service
- **Step 6**: views.js → refactor: newGamemenu to set event listener on button

Step 1 (JS): *data.js* → Declare game variables

Initialize centralized location to store game data accessible by game, controller, and views

data.js

```
let gameID;          //Game ID for Guesser Game
let min;             //Min possible number
let max;             //Max possible number
let gameover;        //Manage game state
```

Step 2 (JS): *controllers.js* → getMin(), getMax()

Define functions to get the min & max values from inputs.

controllers.js

```
const getMin = () => document.getElementById('min-value').value;
const getMax = () => document.getElementById('max-value').value;
```

Step 3 (JS): *game.js* → startGame()

Method to send a request for a new game to the backend API, then display the game view to console.

game.js

```
async function startGame(){
    min = getMin();
    max = getMax();
    const url = `http://localhost:3000/api/game/new?start=${min}&end=${max}`;
    const response = await fetch(url);
    const data = await response.json();
    gameID = data.gameID;
    viewGame();
}
```

Step 4 (JS): ***view.js*** → viewGame()

display the data from the backend service to the console.

views.js

```
const viewGame = function(){
    console.log(gameID, min, max);
}
```

Step 5 (JS): ***controllers.js*** → Refactor: getCallbacks()

In controllers, map callbacks to the corresponding button IDs.

controllers.js

```
const getCallbacks = () => ({
    'new-game-button': newGameMenu,
    'start-game-button': startGame
});
```

Step 6 (JS): ***views.js*** → Refactor: newGameMenu()

Refactor newGameMenu function so that it adds a controller to the start-game button

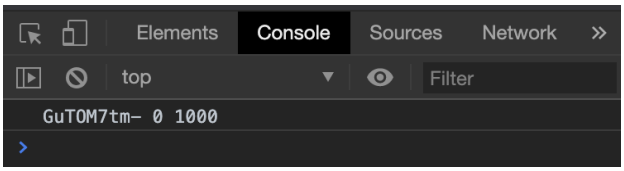
views.js

```
const newGameMenu = function(){
    const view = document.getElementById('view');
    const html =    `<section>
                        Start: <input id='min-value' type=number value=0>
                        End:   <input id='max-value' type=number value=1000>
                        <button id='start-game-button'>Start</button>
                    </section>`
    view.innerHTML = html;
    addController('start-game-button');
}
```

'Assess' → Test phase

Open the index.html in the browser and the main menu should render.

Enter Min & Max values then click the 'Start" button



In console, game data from backend service
Note: Game ID varies each time

Iteration 4: Display → 'Join Game' view

'Approach' → Plan phase

Goal #4: Setup a controller that launches the 'Join Game' view from 'Main Menu' view

Approach: JavaScript EventListeners, DOM manipulation

Use JavaScript to render on the HTML view, set the main menu button with event listeners and callback functions to update the game display.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: views.js → function to render the 'join game' menu to HTML view
- **Step 2**: controllers.js → Refactor getCallbacks: to add callback function & button id
- **Step 3**: views.js → Refactor: mainMenu to set controller on 'join-game-button'

Step 1 (JS): *views.js* → joinGameMenu()

In views, define a joinGameMenu function that overwrites the HTML view div.

views.js

```
const joinGameMenu = function(){
   const view = document.getElementById('view');
   const html =    `<section>
                       Game ID: <input id='room-code' type=text value='GameID'>
                       <button id='find-game-button'>Join</button>
                   </section>`
   view.innerHTML = html;
}
```

Step 2 (JS): *controllers.js* → Refactor: getCallbacks()

In controllers, map new callbacks to the new corresponding button IDs.

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu,
   'start-game-button': startGame,
   'join-game-button': joinGameMenu
});
```

Step 3 (JS): ***views.js*** → Refactor: mainMenu

Refactor mainMenu function so that it adds a controller to the join-game button

views.js

```
const mainMenu = function(){
   const view = document.getElementById('view')
   const html =    `<section>
                       <button id='new-game-button'>New Game</button>
                       <button id='join-game-button'>Join Game </button>
                   </section>`
   view.innerHTML = html;
   addController('new-game-button', 'join-game-button');
}
```

'Assess' → Test phase

Open the index.html in the browser and the main menu should render.

Click on the 'Join Game' button

After click, 'New Game' menu should display

Iteration 5: Fetch Backend → 'Get Game' API

'Approach' → Plan phase

Goal #5: Use fetch on the backend REST API to join an existing guessing game

Approach: Fetch, Await/Async, REST API

Setup a method that uses the backend to get data for an existing game via its Game ID to join.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: controllers.js → function to get value for game id from HTML inputs
- **Step 2:** game.js → requests to backend to get data for existing game with its id, handle failure
- **Step 3**: controllers.js → refactor getCallbacks to add new callbacks and button IDs.
- **Step 4**: views.js → refactor 'joinGameMenu' to setup event listener on buttons

Step 1 (JS): *controllers.js* → getGameID()

Define a function to get the existing game 's ID value from HTML input.

controllers.js

```
const getGameID = () => document.getElementById('room-code').value;
```

Step 2 (JS): *game.js* → findGame()

Request backend to get game data. If successful then set app data and display, otherwise return to menu

game.js

```
async function findGame(){
   gameID = getGameID();
   const response = await fetch(`http://localhost:3000/api/game/${gameID}`);
   const data = await response.json();
   if (data.success){
       min = data.start;
       max = data.end;
       gameover = data.gameover;
       viewGame();
   }
   else{
       mainMenu();
   }
}
```

Step 3 (JS): ***controllers.js*** → Refactor: getCallbacks()

In controllers, map new callbacks to the new corresponding button IDs.

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu,
   'start-game-button': startGame,
   'join-game-button': joinGameMenu,
   'find-game-button': findGame
});
```

Step 4 (JS): ***views.js*** → Refactor: joinGameMenu()

Refactor joinGameMenu function so that it adds a controller to the find-game button

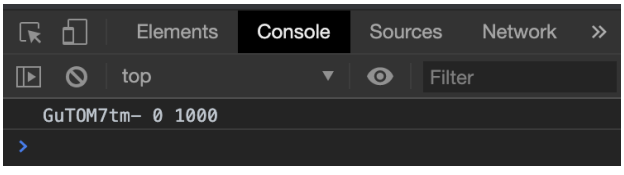
views.js

```
const joinGameMenu = function(){
   const view = document.getElementById('view');
   const html =    `<section>
                       Game ID: <input id='room-code' type=text value='GameID'>
                       <button id='find-game-button'>Join</button>
                   </section>`
   view.innerHTML = html;
   addController('find-game-button');
}
```

'Assess' → Test phase

Open the index.html in the browser and the main menu should render. Create a game, and copy its ID, then refresh the page and try to join that game using the game id.

Enter the Game ID of an existing game. Then click 'Join' button



In console, game data from backend service Note: Game ID varies each time

Iteration 6: Display → Game View

'Approach' → Plan phase

Goal #6: Display game data in the HTML view

Approach: JavaScript EventListeners, DOM manipulation

Refactor the game view method to display game data and controls in the HTML view.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: view.js → refactor 'view game' method to render HTML into the view

Step 1 (JS): *view.js* → Refactor: viewGame()

Define 'viewGame' method to render HTML into the view

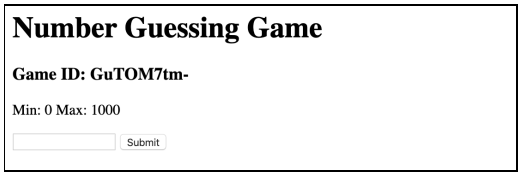
views.js

```
const viewGame = function(){
   const view = document.getElementById('view');
   const html =  `<section>
                     <h3>Game ID: ${gameID}</h3>
                     <p> Min: ${min}  Max: ${max}</p>
                     <input type=number id='guess-input'>
                     <button id='submit-guess-button'>Submit</button>
                     <ul id='clues-list'></ul>
                  </section>`
   view.innerHTML = html;
}
```

'Assess' → Test phase

Open index.html & join that existing game using the same game id. It should display game view

Enter the Game ID of an existing game. Then click 'Join' button



Game View shows data
Note: Game ID varies each time

Iteration 7: Fetch Backend → Submit Guess

'Approach' → Plan phase

Goal #7: Use fetch on the backend REST API to submit a guess

Approach: Fetch, Await/Async, REST API, String Query

Setup a method that uses the backend to submit guesses to an existing game.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: controllers.js → get the user's guess from HTML
- **Step 2:** game.js → requests to backend to submit guess and handle the result
- **Step 3**: views.js → Function that adds a clue to the game view's clue list
- **Step 4**: controllers.js → map new callbacks to the new corresponding button IDs.
- **Step 5**: views.js → Refactor viewGame function to adds a controller to the submit-guess button

Step 1 (JS): *controllers.js* → getGuess()

Define a function to get the user's guess value from HTML input.

controllers.js

```
const getGuess = () => document.getElementById('guess-input').value;
```

Step 2 (JS): *game.js* → submitGuess()

Request backend to submit a guess. If successful then either display a clue or show game over message.

game.js

```
async function submitGuess(){
   const guess = getGuess();
   const url = `http://localhost:3000/api/game/${gameID}/guess?guess=${guess}`;
   const response = await fetch(url);
   const data = await response.json();
   if (data.success){
       switch(data.guess){
           case  "correct": alert("You win!");  break;
           case "gameover": alert("You lose!"); break;
           default: viewClue(data.guess, guess)
       }
   }
}
```

Step 3 (JS): ***views.js*** → viewClue()

Function that adds a clue to the game view's clue list

views.js

```
const viewClue = function(clue, guess){
   clueList = document.getElementById('clues-list');
   clueList.innerHTML += `<li>${guess} is ${clue}</li>`
}
```

Step 4 (JS): ***controllers.js*** → Refactor: getCallbacks()

In controllers, map new callbacks to the new corresponding button IDs.

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu,
   'start-game-button': startGame,
   'join-game-button': joinGameMenu,
   'find-game-button': findGame,
   'submit-guess-button': submitGuess
});
```

Step 5 (JS): ***views.js*** → Refactor: viewGame()

Refactor viewGame function so that it adds a controller to the submit-guess button

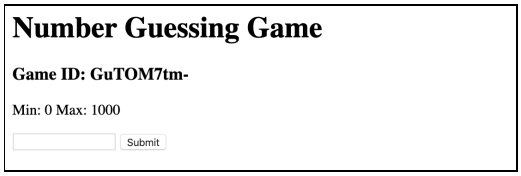
views.js

```
const viewGame = function(){
   const view = document.getElementById('view');
   const html =  `<section>
                        <h3>Game ID: ${gameID}</h3>
                        <p> Min: ${min}  Max: ${max}</p>
                        <input type=number id='guess-input'>
                        <button id='submit-guess-button'>Submit</button>
                        <ul id='clues-list'></ul>
                    </section>`
   view.innerHTML = html;
   addController('submit-guess-button');
}
```

'Assess' → Test phase

Open index.html & join that existing game using the same game id. It should display game view

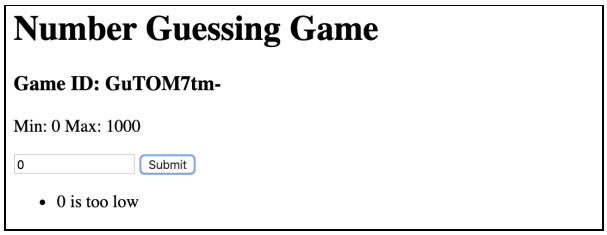
Enter the Game ID of an existing game.
Then click 'Join' button



Game View shows data
Note: Game ID varies each time

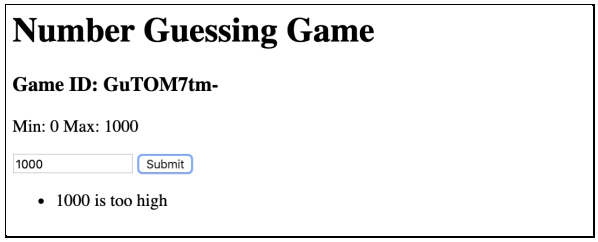
Test: Submit Guess "Too Low"

Submit a guess: **0**



Test: Submit Guess "Too High"

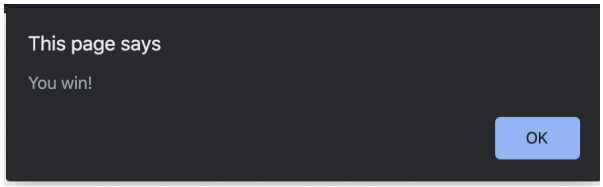
Submit a guess: **1000**



Test: Submit Guess "Correct"

Submit a guess: **Number**

Note: Use the console log from server



Test: Submit Guess "Gameover"

Submit a guess: **Any Number**

Note: Must test this condition after a win

Iteration 8: Display → Game Over

'Approach' → Plan phase

Goal #8: Display Over message in the HTML view

Approach: JavaScript EventListeners, DOM manipulation

Define a gameOverMenu function to display game over in the HTML view.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: views.js → function that overwrites HTML view with Game Over
- **Step 2**: game.js → Refactor submitGuess function to use the gameOverMenu function
- **Step 3**: views.js → Refactor viewGame function to use the gameOverMenu function

Step 1 (JS): views.js → gameOverMenu()

Define 'gameOverMenu' method to render HTML into the view

views.js

```
const gameOverMenu = function(result){
   const view = document.getElementById('view');
   const html =    `<section>
                       <h3>Game ID: ${gameID}</h3>
                       <p>Game Over: ${result}</p>
                       <button id='reset-game-button'>Replay</button>
                       <button id='quit-game-button'>Main Menu</button>
                   </section>`
   view.innerHTML = html;
}
```

Step 2 (JS): game.js → Refactor: submitGuess()

Refactor submitGuess function to use the gameOverMenu function instead of alerts.

game.js

```
async function submitGuess(){
   const guess = getGuess();
   const url = `http://localhost:3000/api/game/${gameID}/guess?guess=${guess}`;
   const response = await fetch(url);
   const data = await response.json();
   if (data.success){
       switch(data.guess){
           case  "correct": gameOverMenu("You win!");  break;
           case "gameover": gameOverMenu("You lose!"); break;
           default: viewClue(data.guess, guess)
       }
   }
}
```

Step 3 (JS): view.js → Refactor: viewGame()

Refactor viewGame function to invoke the gameOverMenu if the gameover variable is true

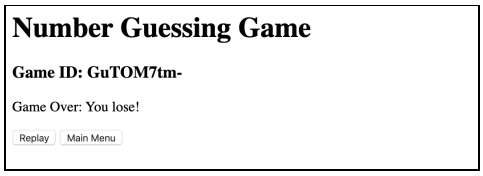
views.js

```
const viewGame = function(){
   const view = document.getElementById('view');
   const html =  `<section>
                       <h3>Game ID: ${gameID}</h3>
                       <p> Min: ${min}  Max: ${max}</p>
                       <input type=number id='guess-input'>
                       <button id='submit-guess-button'>Submit</button>
                       <ul id='clues-list'></ul>
                  </section>`
   view.innerHTML = html;
   addController('submit-guess-button');
   if (gameover) gameOverMenu('You lose!');
}
```

'Assess' → Test phase

Open index.html & join that existing game using the same game id. It should display game view

Enter the Game ID of an existing game. Then click 'Join' button



Game Over since number guessed Note: Game ID varies each time

Iteration 9: Fetch Backend → Reset Game

'Approach' → Plan phase

Goal #9: Use fetch on the backend REST API to reset a game

Approach: Fetch, Await/Async, REST API, String Query

Setup a function that requests the backend to reset a game.

'Apply' → Do phase

JavaScript Steps

- **Step 1**: game.js → requests to backend to reset game and then update game view
- **Step 2**: controllers.js → map new callbacks to the new corresponding button IDs.
- **Step 3**: views.js → Refactor gameOver function to adds a controller to the reset-game button

Step 1 (JS): game.js → Refactor: resetGame()

Requests to backend to reset game and then update game view

game.js

```
async function resetGame(){
   const url = `http://localhost:3000/api/game/${gameID}/reset`;
   const response = await fetch(url);
   const data = await response.json();
   if (data.success){
       gameover = data.gameover;
       viewGame();
   }
}
```

Step 2 (JS): controllers.js → Refactor: getCallbacks()

map new callbacks to the new corresponding button IDs

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu,
   'start-game-button': startGame,
   'join-game-button': joinGameMenu,
   'find-game-button': findGame,
   'submit-guess-button': submitGuess,
   'reset-game-button': resetGame
```

```
});
```

Step 3 (JS): views.js → Refactor: gameOverMenu()

Refactor gameOver function to adds a controller to the reset-game button

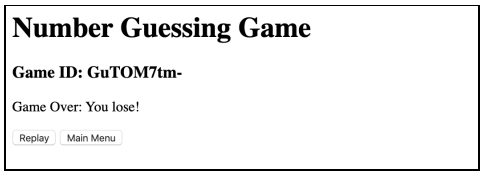
views.js

```
const gameOverMenu = function(result){
   const view = document.getElementById('view');
   const html =    `<section>
                       <h3>Game ID: ${gameID}</h3>
                       <p>Game Over: ${result}</p>
                       <button id='reset-game-button'>Replay</button>
                       <button id='quit-game-button'>Main Menu</button>
                   </section>`
   view.innerHTML = html;
   addController('reset-game-button');
}
```

'Assess' → Test phase

Open index.html & join that existing game using the same game id. It should display game view

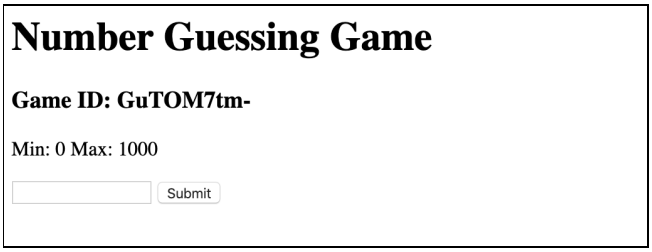
Enter the Game ID of an existing game.
Then click 'Join' button



Game Over since number guessed
Note: Game ID varies each time

Test: Reset Game

Click the Replay button



Game should then return to Game View with a reset game.

Iteration 10: Quit Game

'Approach' → Plan phase

Goal #10: Quit game and go back to Main Menu

Approach: JavaScript EventListeners, DOM manipulation

Define a controller that overwrites the Game Over view with the Main Menu view

---

'Apply' → Do phase

JavaScript Steps

- **Step 1**: controllers.js → map new callbacks to the new corresponding button IDs.
- **Step 2**: views.js → Refactor gameOver function to adds a controller to the quit-game button

Step 1 (JS): *controllers.js* → Refactor: getCallbacks()

map new callbacks to the new corresponding button IDs

controllers.js

```
const getCallbacks = () => ({
   'new-game-button': newGameMenu,
   'start-game-button': startGame,
   'join-game-button': joinGameMenu,
   'find-game-button': findGame,
   'submit-guess-button': submitGuess,
   'reset-game-button': resetGame,
   'quit-game-button': mainMenu
});
```

Step 2 (JS): *views.js* → Refactor: gameOverMenu()

Refactor gameOver function to adds a controller to the quit-game button

views.js

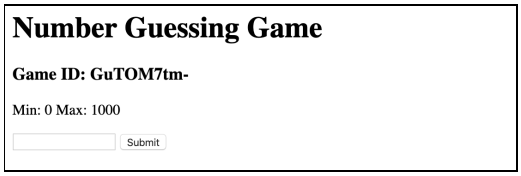
```
const gameOverMenu = function(result){
   const view = document.getElementById('view');
   const html =    `<section>
                       <h3>Game ID: ${gameID}</h3>
                       <p>Game Over: ${result}</p>
                       <button id='reset-game-button'>Replay</button>
                       <button id='quit-game-button'>Main Menu</button>
                   </section>`
```

```
    view.innerHTML = html;
    addController('reset-game-button', 'quit-game-button');
}
```

'Assess' → Test phase

Open index.html & join that existing game using the same game id. It should display game view

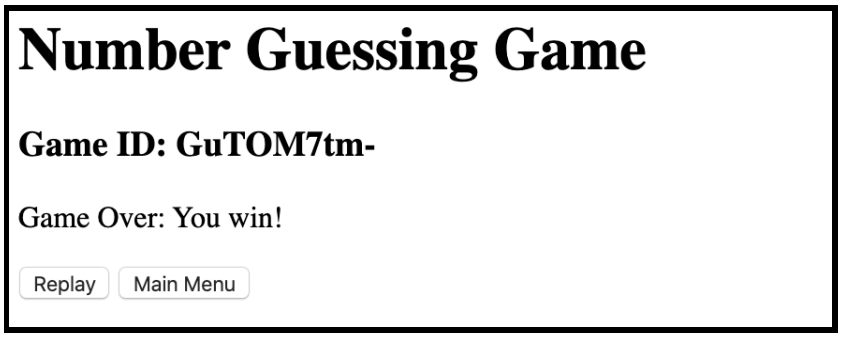
Enter the Game ID of an existing game.
Then click 'Join' button



Game View shows data
Note: Game ID varies each time

Test: Win Game

Play the game until you get the number



Test: Quit Game

From game over menu select 'Main Menu'

Conclusions

Final Comments

In this lab you implemented a frontend REST client for a Multiplayer Number Guessing Game. This lab covered: async/await fetches, DOM manipulations, HTTP JSON responses,

Future Improvements

- Style the frontend app
- Add Usernames for a game and show name & win record as a scoreboard
- Synchronize users together and display who is playing (via. web sockets)
- Store the games in a database
- Deploy into production on heroku

Lab Submission

Compress your project folder into a zip file and submit it to Moodle.