Lab 07: *REST Client* **Frontend**

Quiz Game + Leaderboard

REST Client App (HTTP + Fetch API) and JavaScript Modules

REQUIRES: HTTP PROTOCOL

Table of Content: *Quiz Game + Leaderboard*

Lab Introduction

Prerequisites

HTTP Protocol. Software Requirements: Chrome Web Server or Node http-server; to host HTML.

Motivation

Learn the Fetch API to build a REST Client & JS Modules to better organize an App's codebase.

Goal

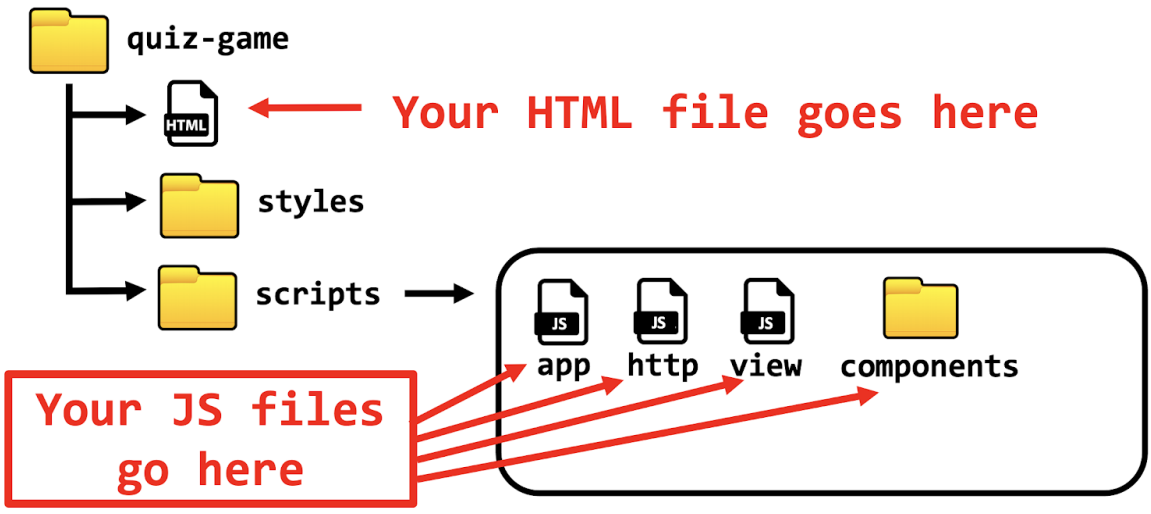
Build a Quiz game that requests trivia from a web server & maintains a shared leaderboard for all players.

Learning Objectives

- Fetch API to send requests to REST Endpoints
- REST Client architecture for developing a distributed application
- Asynchronous JavaScript with async/await functions
- JavaScript Modules with import/export statements for improved modularity of codebase
- Classless CSS libraries that attractively style an HTML document without classnames
- Scheduling & Clearing Timed Events
- Using tDocument Object Model (DOM) to implement a Single Page Application (SPA)

Project Architecture:

Start this project by downloading the starter files from github. See the project structure below.



Download Starter files:
https://github.com/scalemailted/quiz-game/archive/master.zip

REST APIs, REST Clients & Fetch

A full stack app shares data between clients & servers via HTTP requests/responses. A browser is a REST client when it depends on HTTP Requests for data. A REST API refers to the data that a server provides via HTTP requests.

- **HTTP Request:** Browsers requests data from web servers using HTTP. HTTP Requests have four common methods: GET (read), PUT (update), POST (create), DELETE (delete)
- **REST Endpoint:** The URLs on a server that listen for HTTP requests and return back data.
- **Fetch:** An asynchronous JavaScript API that allows browsers to send HTTP requests.

Asynchronous JavaScript

JavaScript provides non-blocking functions which allow for other code to execute while the asynchronous function waits for its process to complete. This is an important feature for networking-related tasks, so that your code does not wait until the server response before proceeding to its next tasks.

- **async/await:** Any function may be declared as asynchronous with the async keyword. The statement whereby an I/O operation occurs is declared with the await keyword.
- **Promise object**: Asynchronous functions return Promise objects. This allows for the function to execute and move to the next task. The promise object is later resolved when the async function completes, thereby resolving the promise. Callbacks can be triggered when a promise is resolved.

JavaScript Modules

Modules allow imports between JS files without linking the script into the HTML. This provides for more readable, browsable, scalable code. Modules require the HTML to be hosted by HTTP. Modules have their own scope that is not part of the global scope or function scope.

- **module:** HTML document would link a script with the type of "module".
- **import:** The `import` keyword allows a script to access functions or objects in its module scope
- **export:** The `export` keyword provides external access to functions/objects. These are returned as an anonymous object containing all the exported functions/objects.
- **default:** The `default` keyword exports a single function/object along with its name

Goal 0: JavaScript Modules - Crash Course

Part 0: Project Directory for Modules Tester

Part 1: Declare a module in HTML & use HTTP to host the HTML (index.html)

```
<html>
   <head>
       <script type="module" src="foo.js"></script>
   </head>
</html>
```

Part 2: Declare an Export on a function/object (bar.js)

```
const bar = { name:'bar', type:'spam', age:0 };
export default bar;
```

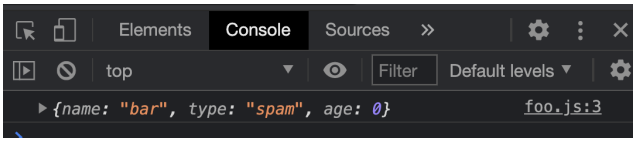
Create an Object or Function & Export it from bar.js

Part 3: Declare an Import on a function/object (foo.js)

```
import bar from './bar.js';
console.log(bar);
```

Import the default exported function in foo.js

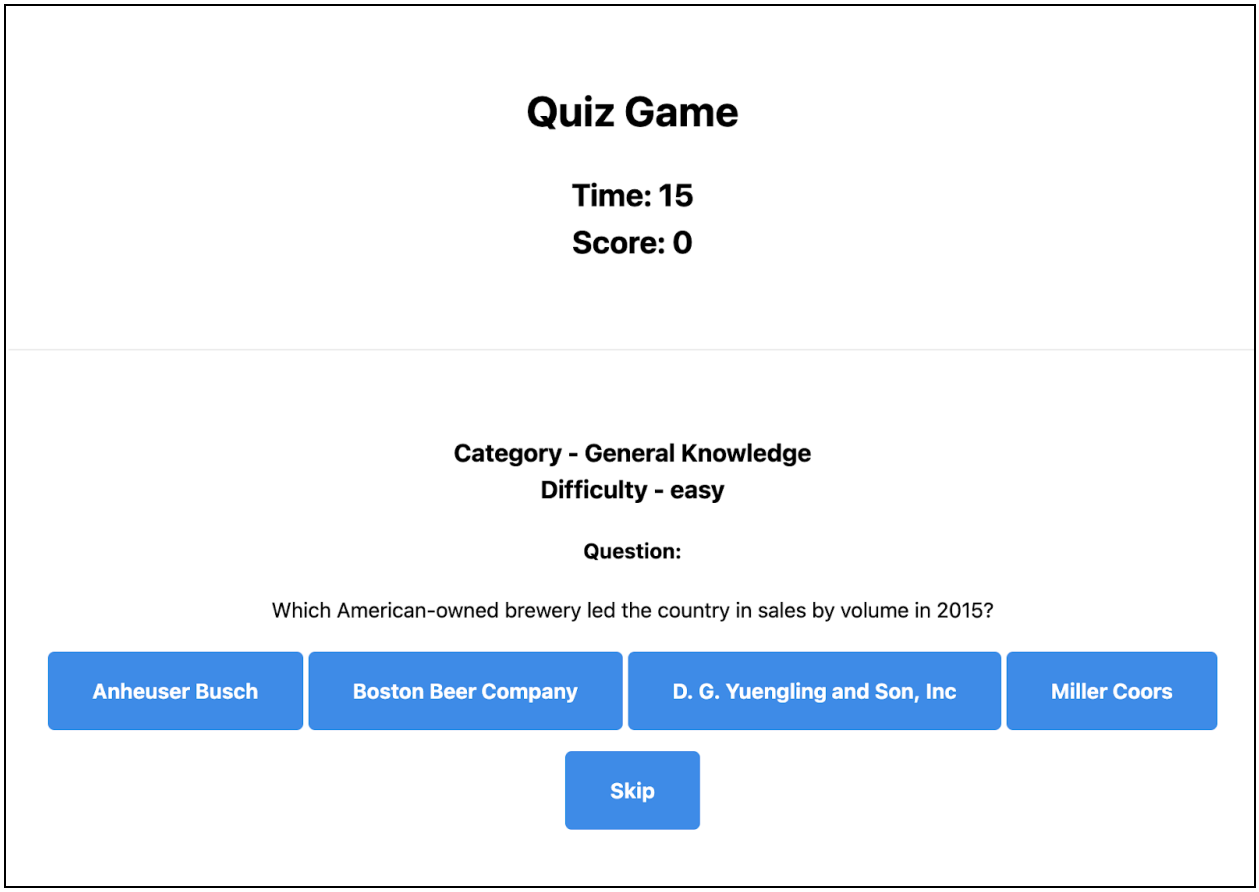
Part 4: Use HTTP to Open index.html (Chrome Web Server or Node http-server)



NOTE: Do not advance through this lab until you get this example project working!

Part 1:

Quiz Game - (GET Requests, DOM, Events)



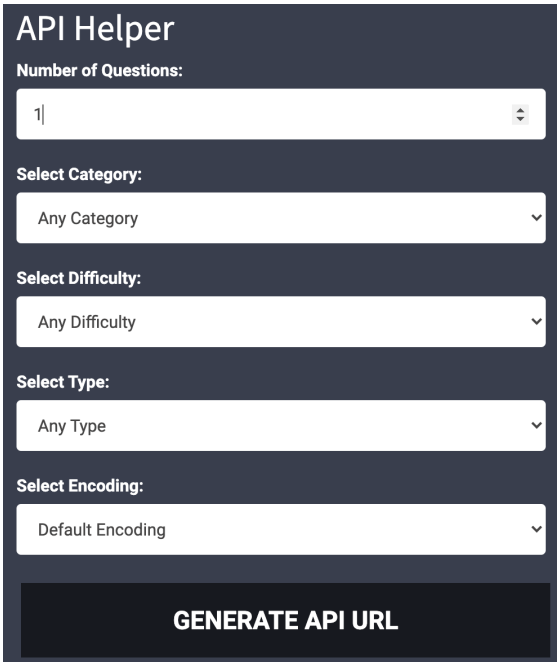
Goal 1-0: Trivia API

Summary:

The Open Trivia Database provides a completely free JSON API for use in programming projects. Use of this API does not require an API Key, just use a URL (with SearchParams) within your own application to retrieve trivia questions.

More Info: https://opentdb.com/api_config.php

API Overview:



The Trivia API offers 5 different parameters:

- Number of Questions
- Category for Questions
- Difficulty of Questions
- Type of Questions: True/False or Multichoice
- Data Encoding for Sending into Browser

Their website allows an easy URL generator such that you select the parameters you want and it generates the REST Endpoint.

Since the entire REST API uses GET Requests, you can test the API out the browser using the URL.

Example: ***(Open the link in a Browser)***
https://opentdb.com/api.php?amount=1&category=19&difficulty=hard&type=multiple

Goal 1-1: GET Request to Web Server for Trivia data

'Approach' → Plan Phase

Use an HTTP GET Request on the web server's REST endpoints to get Trivia data into the Browser

'Apply' → Do Phase

Step 1: [HTML] Create HTML file that links CSS & JS files, and that has an element for the 'view'

index.html

```
<html>
  <head>
       <link rel="stylesheet" href="styles/mvp.css">
       <script type='module' src='scripts/app.js'></script>
  </head>
  <body>
      <header id='view'></header>
  </body>
</html>
```

Step 2: [http.js] Define exported function that sends an HTTP GET Request to a REST Endpoint

scripts/http.js → sendGetRequest

```
export const sendGETRequest = async (url) => {                    //Asynchronous function
   const options = new Object();                                  //Empty object for HTTP options
   options.method = "GET";                                        //HTTP method: GET
   const response = await fetch(url, options);                    //Send request with fetch (wait)
   const data = await response.json();                            //Extract json from response (wait)
   return data                                                    //return data as an Object
}
```

Step 3: [app.js] Import all exports from http.js as http, & initialize Trivia GET endpoint & the game state

scripts/app.js → imports and variables

```
import * as http from './http.js'                                 //Import http functions
const GET_TRIVIA = `https://opentdb.com/api.php?amount=1&difficulty=easy`;      //Trivia GET endpoint
const state = {};                                                 //Game state
```

Step 4: [app.js] Define a 'play' function to play the quiz *(for now, just print the result of GET request)*

scripts/app.js → play()

```
const playGame = async () => {                                    //PLAY function
   const json = await http.sendGETRequest(GET_TRIVIA);            //GET Request for trivia data
   console.log(json);                                             //Print trivia data
}
```

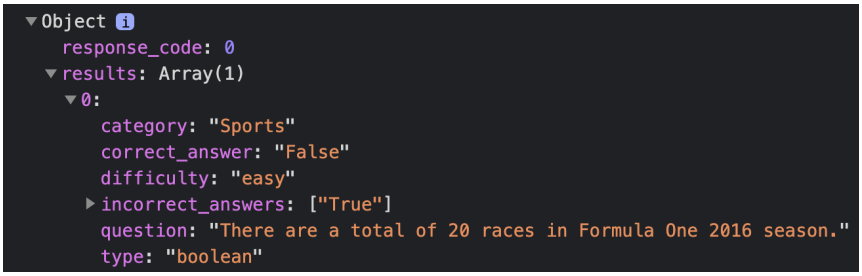
Step 5: [app.js] Define a global 'start' function to launch the app & have it execute when window loads

scripts/app.js → start()

```
window.start = async () => {                                    //START function
   playGame();                                                  //call play function
}

window.addEventListener('load', start);                         //When window loads execute start
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser. Check the dev console, where an Object containing a results array with the trivia data.



Note: Inspect the Object to see its properties.

Goal 1-2: Display Trivia Question to DOM

'Approach' → Plan Phase

Use DOM API to access the HTML element with view ID and set its inner HTML with the Question text

'Apply' → Do Phase

Step 1: [Question.js] Define function to return the HTML for Trivia Question, then export as default

scripts/components/Question.js → Question

```
const Question = (trivia) => (                                          //Function for HTML component
   `<h3>
       <div>Category -   ${trivia.category}</div>
       <div>Difficulty - ${trivia.difficulty}</div>
   </h3>
   <h4>Question:</h4>
   <p>${trivia.question}</p>`
)                                                                        //Return HTML text

export default Question;                                                 //Export Question function
```

Step 2: [view.js] Import the `Question` function from `Question.js`

scripts/view.js → imports

```
import Question from './components/Question.js';                        //Import Question function
```

Step 3: [view.js] Define a function to display new html text into the view

scripts/view.js → renderDOM

```
const renderDOM = (html) => document.getElementById('view').innerHTML = html;    //Set HTML in view
```

Step 4: [view.js] Define an exported function that renders the Play Scene

scripts/view.js → PlayScene

```
export const PlayScene = (props) => {                                   //Function for HTML view
   const {trivia} = props;                                              //Destructure properties
   renderDOM(                                                           //render the Scene's HTML to DOM
       `${Question(trivia)}`
   )
}
```

Step 5: [app.js] import all exports from `view.js` as an object named `view`

scripts/app.js → imports

```
import * as http from './http.js'                                       //Import http functions
import * as view from './view.js';                                      //Import view functions
```

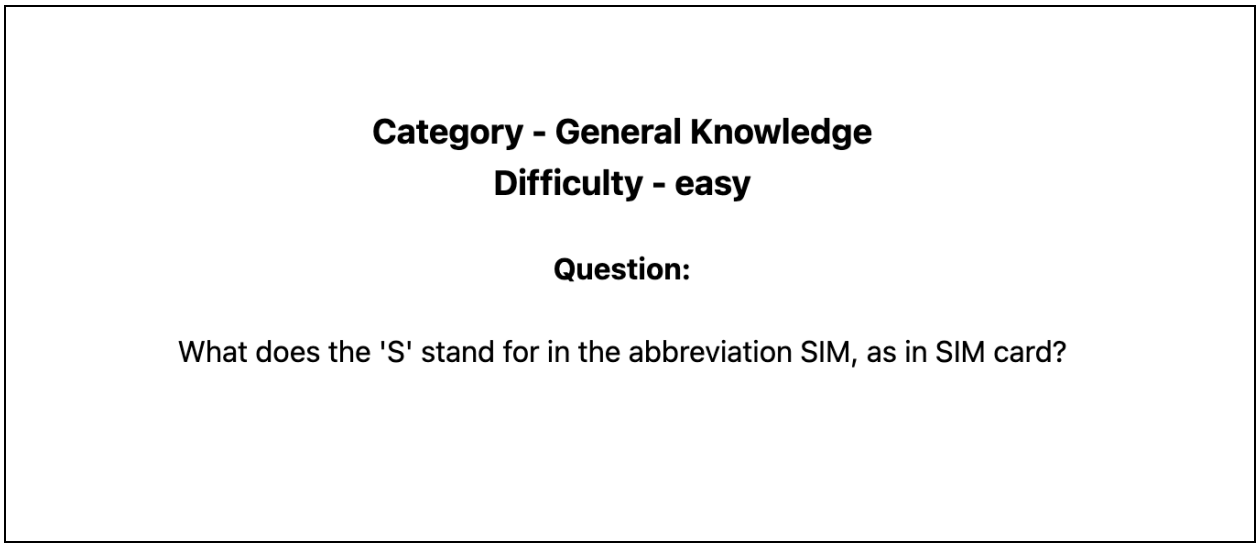
Step 5: [app.js] Refactor play function to destructure trivia object from the json results & send to view

scripts/app.js → playGame

```
const playGame = async () => {                                  //PLAY function
  const json = await http.sendGETRequest(GET_TRIVIA);           //GET Request for trivia data
  [ state.trivia ] = json.results;                              //Destructure trivia data from array
  view.PlayScene(state);                                        //Pass trivia data to view
}
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser. The Trivia Question should display in the viewport.



Goal 1-3: Display Trivia Answers to DOM

'Approach' → Plan Phase

Define an Options component for the HTML of a Trivia Options & use it from the Questions component

'Apply' → Do Phase

Step 1: [Options.js] Define a function that returns the HTML for a Trivia's Boolean Options

scripts/components/Options.js → BooleanOptions

```
const BooleanOptions = () => (                                          //Function for HTML component
   `<div>
       <button onclick="checkAnswer('True')">True</button>
       <button onclick="checkAnswer('False')">False</button>
   </div>`
)                                                                       //Return HTML text
```

Step 2: [Options.js] Define a function that returns the HTML for a Trivia's Multiple Choice Options

scripts/components/Options.js → MultiOptions

```
const MultiOptions = (trivia) => {                                      //Function for HTML component
   const options  = [trivia.correct_answer, ...trivia.incorrect_answers]; //Array with all answers
   const [a, b, c, d] = options.sort( );                                //Sort answers & destructure
   return (
       `<div>
           <button onclick='checkAnswer("${a}")'>${a}</button>
           <button onclick='checkAnswer("${b}")'>${b}</button>
           <button onclick='checkAnswer("${c}")'>${c}</button>
           <button onclick='checkAnswer("${d}")'>${d}</button>
       </div>`
   )                                                                    //Return HTML text
}
```

Step 3: [Options.js] Define a function that returns the HTML for Trivia Options, then export as default.

scripts/components/Options.js → Options

```
const Options = (trivia) => {                                   //Function for HTML component
   switch(trivia.type){                                         //Switch Options based on type
       case "boolean":  return BooleanOptions(trivia);          //Return true/false type
       case "multiple": return MultiOptions(trivia);            //Return multiple choice type
   }
}

export default Options;                                         //Export the Options function
```

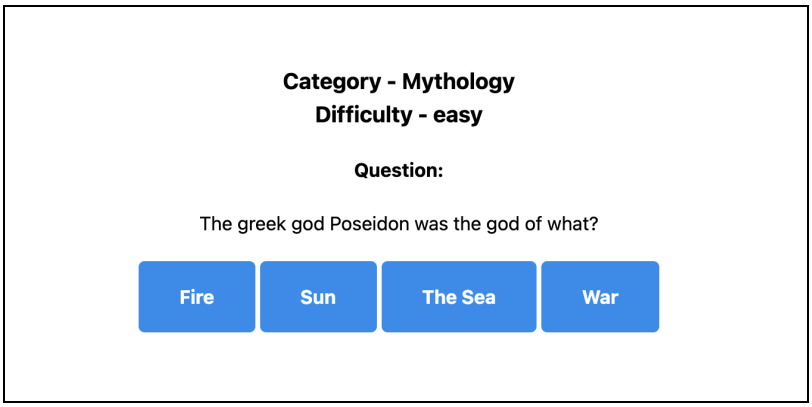
Step 4: [Question.js] Import `Options` from `Options.js` & invoke it within the `Question` function.

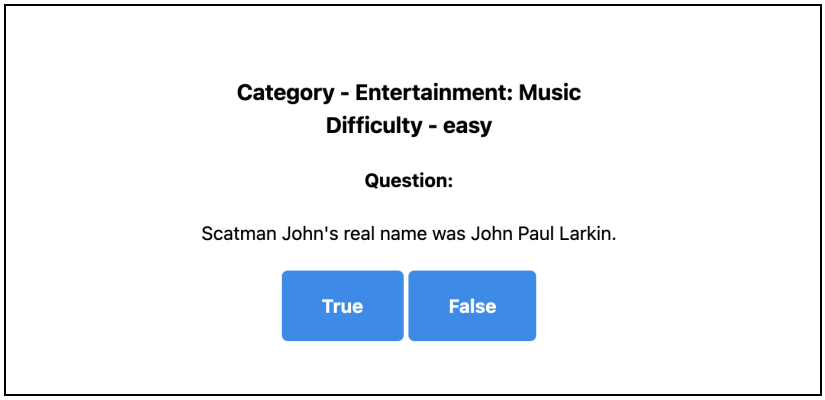
scripts/components/Question.js → Question

```
import Options from './Options.js'                                    //Import Options function

const Question = (trivia) => (                                        //Function for HTML component
  `<h3>
      <div>Category -   ${trivia.category}</div>
      <div>Difficulty - ${trivia.difficulty}</div>
  </h3>
  <h4>Question:</h4>
  <p>${trivia.question}</p>
  ${Options(trivia)}`
)                                                                     //Return HTML text

export default Question;                                              //Export Questions function
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser. The Trivia Question should display in the viewport along with Answer Options.





Goal 1-4: Timer Event & Display HUD

'Approach' → Plan Phase

Create a timer that counts down & a HUD component to show the timer & score.

'Apply' → Do Phase

Step 1: [app.js] Refactor Game state with initial properties: score, timer, interval id, trivia

scripts/app.js → state

```
const state = {
  score: 0,
  timer: 20,
  intervalId: null,
  trivia: null
};
```

Step 2: [app.js] Define a function that checks if timer is nonzero then decrement it & render play scene

scripts/app.js → countdown

```
const countdown = () => {
  if (state.timer){
     state.timer--;
     view.PlayScene(state);
  }
}
```

```
//COUNTDOWN function
//check if time remains
//decrement timer
//view render play scene
```

Step 3: [app.js] Define function to set timer, schedule a countdown interval, & call the play function

scripts/app.js → createGame

```
const createGame = () => {
  state.timer = 20;
  state.intervalId = setInterval(countdown, 1000);
  playGame();
}
```

```
//CREATE function
//set timer
//set interval id
//call PLAY function
```

Step 4: [app.js] Refactor start function to call createGame function

scripts/app.js → start

```
window.start = async () => {
  createGame();
}
```

```
//START function
//call CREATE function
```

Step 5: [HUD.js] Define function to return the HTML for HUD, then export as default

scripts/components/HUD.js → HUD

```
const HUD = (timer, score) => (                                    //Function for HTML component
   `<h1>Quiz Game</h1>
   <h2>
       <div id='time'> Time:  ${timer} </div>
       <div id='score'> Score: ${score}</div>
   </h2>
   <hr>`
)

export default HUD;                                                //Export HUD function
```

Step 6: [view.js] Import the `HUD` function from `HUD.js`

scripts/view.js → imports

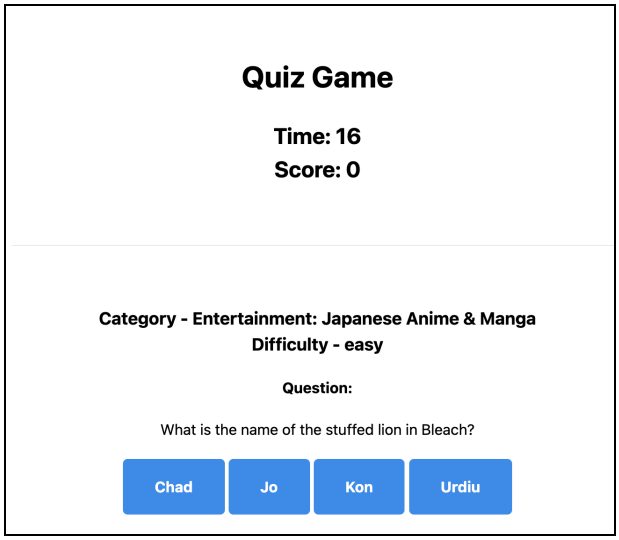
```
import Question from './components/Question.js';                   //Import Question function
import HUD from './components/HUD.js';                             //Import HUD function
```

Step 7: [view.js] Add the HUD component into the Play Scene's render call

scripts/view.js → PlayScene

```
export const PlayScene = (props) => {                              //Function for HTML view
   const {timer, score, trivia} = props;                           //Destructure properties
   renderDOM(                                                      //Render the Scene's HTML to DOM
       `${HUD(timer, score)}
       ${Question(trivia)}`
   )
}
```

'Approve' → Test Phase



Using an HTTP server, (i.e. Chrome Web Server)

Open the `index.html` file in the browser.

A Heads-Up Display (HUD) should be at top with a timer that counts down from 20 to 0.

Goal 1-5: Display Gameover to DOM

'Approach' → Plan Phase

GameOver Scene displays when the timer is 0. Gameover scene has a button to start a new game.

'Apply' → Do Phase

Step 1: [app.js] Refactor countdown to clear the Timed Interval and show a Gameover screen

scripts/app.js → countdown

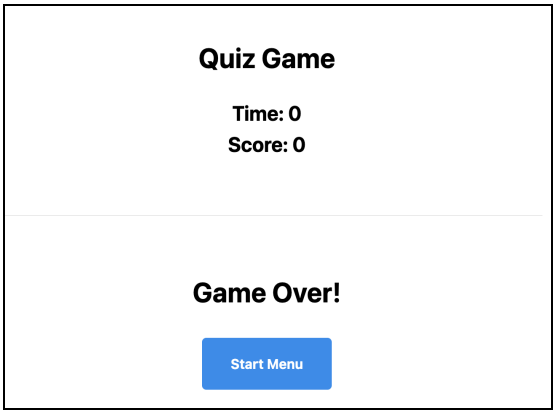
```
const countdown = () => {                                   //COUNTDOWN function
  if (state.timer > 0 ){                                    //when time remains
     state.timer--;                                         //decrement timer
     view.PlayScene(state);                                 //view render play scene
  }
  else{                                                     //when timer is 0
     clearInterval( state.intervalId );                     //stop countdown interval
     view.GameoverScene(state);                             //show gameover view
  }
}
```

Step 2: [view.js] Define a Gameover view that renders to DOM

scripts/view.js → GameoverScene

```
export const GameoverScene = (props) => {                   //Function for HTML view
   const {timer, score, trivia} = props;                    //Destructure properties
   renderDOM(                                               //render the Gameover HTML to DOM
       `${HUD(timer, score)}
       <h1>Game Over!</h1>
       <button onclick='start()'>Start Menu</button>`
   )
}
```

'Approve' → Test Phase



Using an HTTP server, (i.e. Chrome Web Server)

Open the `index.html` file in the browser.

When the timer is 0, Game Over should display.
If you click the button, a new game should start.

Goal 1-6: Display Next Trivia to DOM

'Approach' → Plan Phase

The Player may skip this question and request a new one before time expires.

'Apply' → Do Phase

Step 1: [Skip.js] Define function to return HTML for Skip button, then export as default

scripts/components/Skip.js → Skip

```
const Skip = () => (                                          //Function for HTML component
   `<div>
       <button onclick='playGame()'> Skip </button>
   </div>`
)

export default Skip;                                           //Export Skip function
```

Step 2: [view.js] Import the `Skip` function from `Skip.js`

scripts/view.js → imports

```
import Question from './components/Question.js';                    //Import Question function
import HUD from './components/HUD.js';                              //Import HUD function
import Skip from './components/Skip.js';                            //Import Skip function
```

Step 3: [view.js] Add the Skip component into the Play Scene's render call

scripts/view.js → PlayScene

```
export const PlayScene = (props) => {                         //Function for HTML view
   const {trivia, timer, score} = props;                      //Destructure properties
   renderDOM(                                                 //render the Scene's HTML to DOM
       `${HUD(timer, score)}
       ${Question(trivia)}
       ${Skip()}`
   )
}
```

Step 4: [app.js] Refactor playGame to global scope so Event Dispatcher may access it.

scripts/app.js → playGame

```
window.playGame = async () => {                                //PLAY function
  const json = await http.sendGETRequest(GET_TRIVIA);         //GET Request for trivia data
  [ state.trivia ] = json.results;                            //Destructure trivia data from array
  view.PlayScene(state);                                      //Pass trivia data to view
}
```

'Approve' → Test Phase

Use an HTTP server to serve index.html to Browser. Click the SKIP button to advance to a new question.

Goal 1-7: Submit Answer & Handle Result

'Approach' → Plan Phase

Define a checkAnswer function that determines if the player is correct or incorrect.

'Apply' → Do Phase

Step 1: [app.js] Define a global function for event dispatcher to determine if user is correct or incorrect

scripts/app.js → checkAnswer

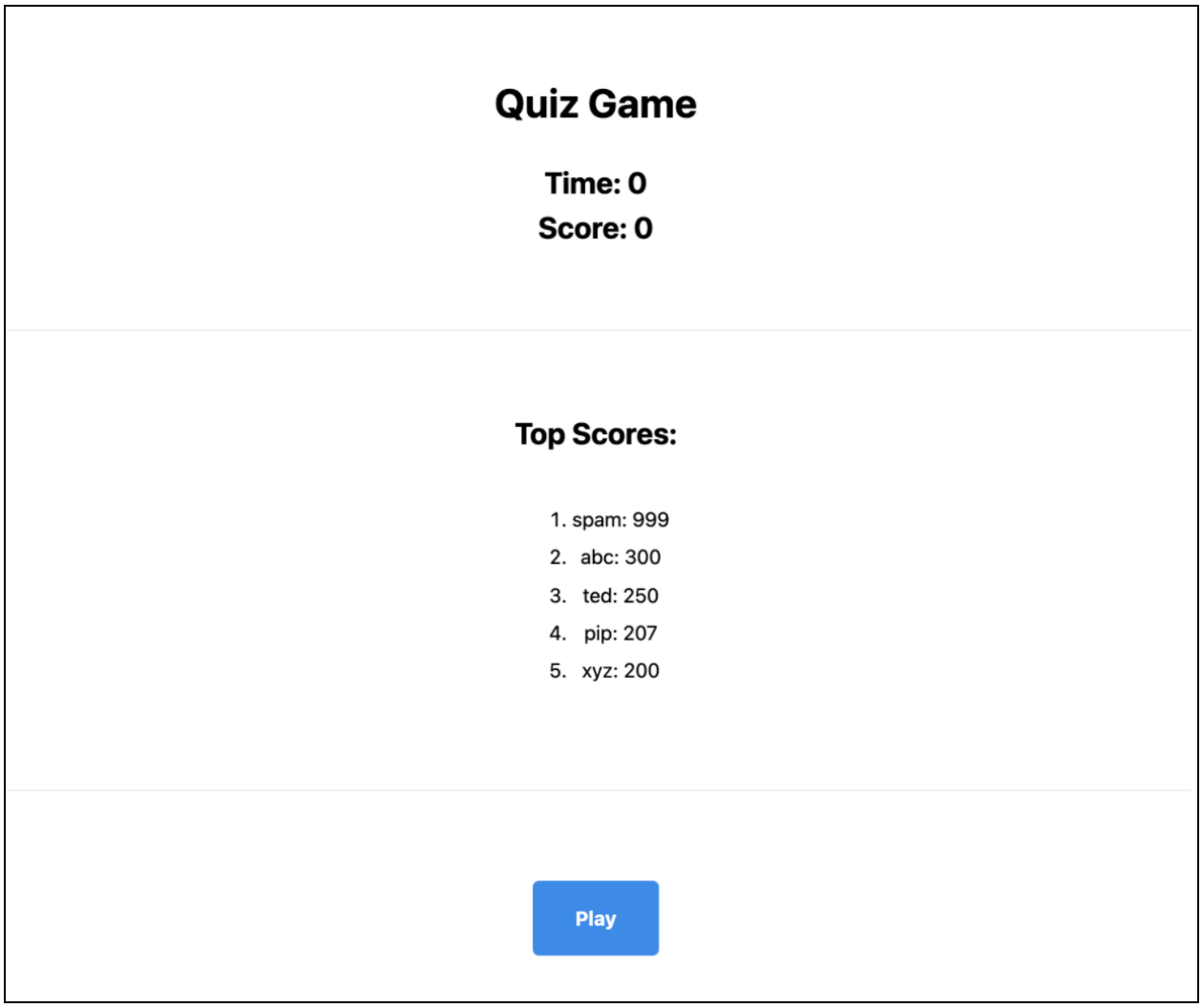
```
window.checkAnswer = (attempt) => {                    //CHECK_ANSWER function
  const answer = state.trivia.correct_answer;          //Dereference answer
  if (attempt == answer){                              //When Attempt is correct
      state.score += state.timer;                      //Add to Score based on time
      state.timer += 10;                               //Add 10 bonus seconds
      playGame();                                      //Play Next Round of Trivia
  }
  else {                                               //When Attempt is incorrect
      clearInterval( state.intervalId );              //stop countdown interval
     view.GameoverScene(state);                        //show gameover view
  }
}
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser.
Play the Game, if you get the answer correct your score goes up, if you get it wrong its game over.

Part 2:

Leaderboard - (GET Requests, PUT Requests, DOM, Events)

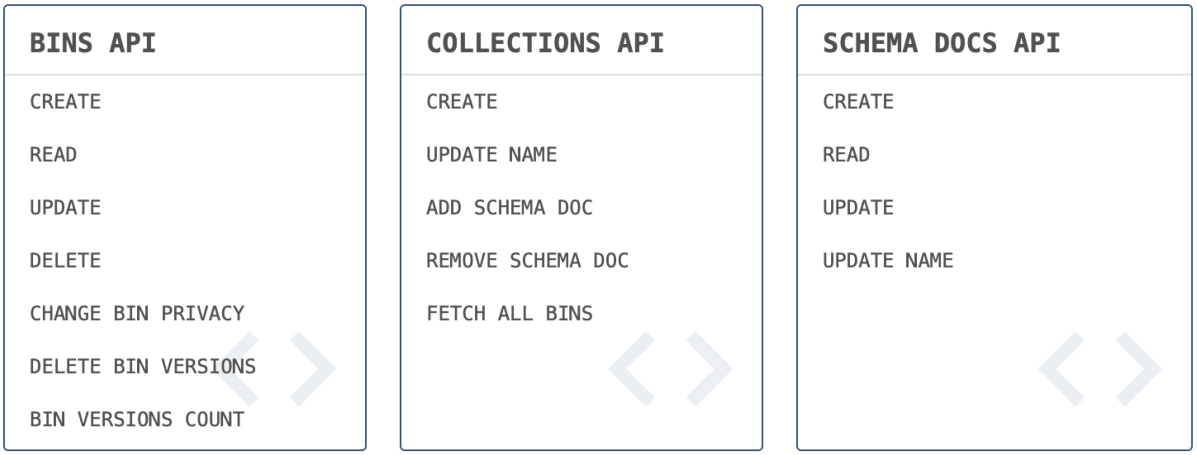


Goal 2-0: JsonBin API

Summary:
JsonBin provides a free JSON hosting service for public or private data. Note that ant data available from the browser should be public. Never use private keys in your browser code.

More Info: https://jsonbin.io/api-reference

API Overview:



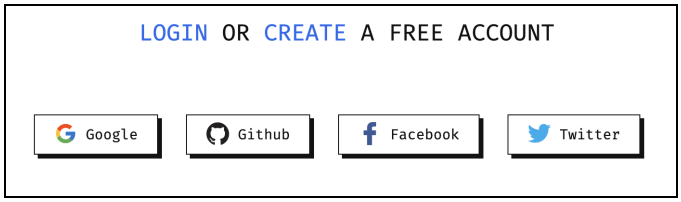
- **Bins API:** Create (POST), Read (GET), Update (PUT) Private & Public bins using the Create API.
  - GET & PUT actions do not require a key, safe to perform in public code
  - POST & DELETE actions require a key, unsafe to perform in public code
- **Collections API:** Using the COLLECTIONS CREATE API, you can CREATE Collections to group the records which later, can be fetched using the Query Builder.

Example from LAB: ***(Open the link in a Browser)***
https://api.jsonbin.io/b/605ed0fd9ab74a5f2bcc9c02/latest

Create your own JSON Bin

(for use in this Lab)

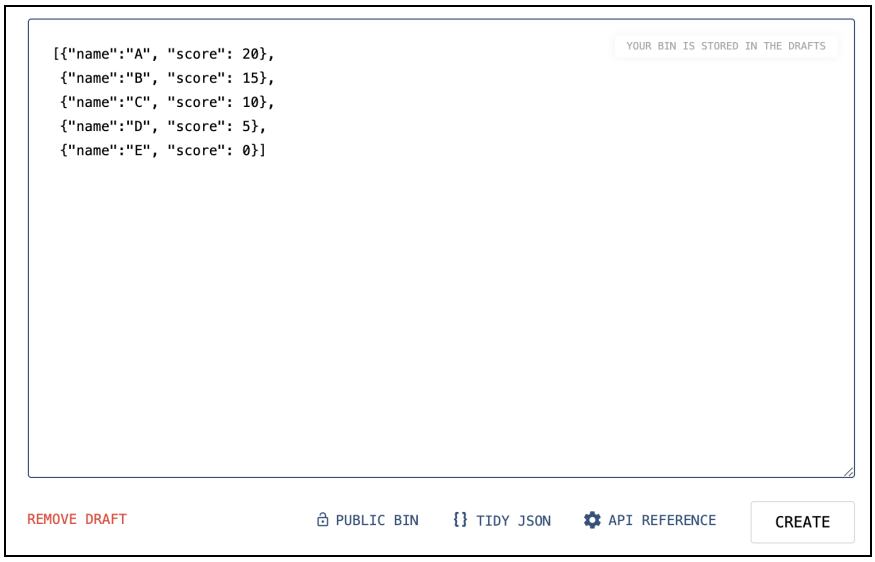
Part 1: Create a Free account: https://jsonbin.io/login



Part 2: Go to Dashboard: https://jsonbin.io/dashboard

Part 3: Select "Create New" button

Part 4: Enter valid JSON into the editor (use the example below)



Part 5: Toggle to PUBLIC BIN *(so the data is available without a key)*

Part 6: Select the CREATE button

Part 7: JSON Bin is POSTed. Use the ACCESS URL field to get its BIN ID.

Goal 2-1: Display Main Menu to DOM

'Approach' → Plan Phase

Define a Main Menu Scene in view & render to DOM.

'Apply' → Do Phase

Step 1: [view.js] An exported function that renders the Start Menu into the view

scripts/view.js

```
export const StartMenu = (props) => {
   const {timer, score, trivia} = props;
   renderDOM(
       `${HUD(timer,score)}
       <hr>
       <button onclick='createGame()'>Play</button>`
   )
}
```

Step 2: [app.js] Refactor START to set score to 0 & have view render the Start Menu.

scripts/app.js → start()

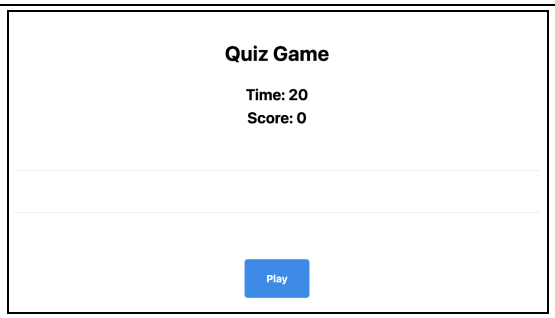
```
window.start = async () => {
   state.score = 0;                                          //reset score
   state.timer = 20;                                         //reset timer
   view.StartMenu(state);                                    //render Start Menu
}
```

Step 3: [app.js] Refactor createGame into global scope so Event Dispatcher may access it.

scripts/app.js → createGame

```
window.createGame = () => {                                  //CREATE function
  state.intervalId = setInterval(countdown, 1000);           //set interval id
  playGame();                                                //call PLAY function
}
```

'Approve' → Test Phase



Using an HTTP server, (i.e. Chrome Web Server)

Open the `index.html` file in the browser.

Game should launch into Start Menu
Gameover should go to Start Menu

Goal 2-2: GET Request to Web Server for Leaderboard

'Approach' → Plan Phase

Use an HTTP GET Request on the web server's REST endpoints to get Leaderboard into the Browser

'Apply' → Do Phase

Step 1: [app.js] Initialize constants for the jsonbin ID & the HTTP GET Endpoint for the JSON.

scripts/app.js → constants

```
const GET_TRIVIA = `https://opentdb.com/api.php?amount=1&difficulty=easy`;
const BIN_ID = '605ed0fd9ab74a5f2bcc9c02';                    //replace with your own
const GET_LEADERBOARD = `https://api.jsonbin.io/v3/b/${BIN_ID}/latest`;
```

***Note: Use your own BIN ID.*

Step 2: [app.js] Add a TopScores property into the Game State

scripts/app.js → state

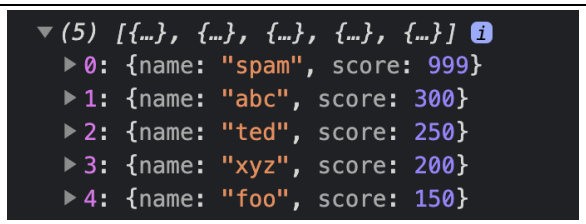
```
const state = {
  score: 0,
  timer: 20,
  intervalId: null,
  trivia: null,
  topScores: []
};
```

Step 3: [app.js] Refactor START to GET leaderboard data from server & display in console

scripts/app.js → start()

```
window.start = async () => {                                    //START function
   const leaderboardJSON = await http.sendGETRequest(GET_LEADERBOARD);   //Fetch LeaderBoard
   state.topScores = leaderboardJSON.record;                    //data in record prop
   console.log(state.topScores);                                //Print LeaderBoard
   state.score = 0;                                             //reset score
   state.timer = 20;                                            //reset timer
   view.StartMenu(state);                                       //render Start Menu
}
```

'Approve' → Test Phase



Using an HTTP server, (i.e. Chrome Web Server)

Open the `index.html` file in the browser.

In the console is the response json object.
The name & scores from the server.

Goal 2-3: Display Leaderboard to DOM

'Approach' → Plan Phase

Define a Leaderboard component that returns the HTML to be used by the Main Menu Scene.

'Apply' → Do Phase

Step 1: [Leaderboard.js] Define function to return HTML for Leaderboard, then export as default

scripts/components/Leaderboard.js → Leaderboard

```
const Leaderboard = (topScores) => (                              //Function for HTML component
   `<h2>Top Scores:</h2>
   <section>
       <ol>
           ${ ListItems(topScores) }
       </ol>
   </section>`
);
export default Leaderboard;                                        //Export Leaderboard function
```

Step 2: [Leaderboard.js] Define helper function to return HTML of list items from an array

scripts/components/Leaderboard.js → ListItems

```
const ListItems = (topScores) => {                                //Function for HTML component
   let li = ``;                                                   //empty string for HTML of list items
   const scores = topScores.sort( (a,b) => b.score - a.score );   //sort by scores
   for (let row of scores){                                       //for each row in scores
       li += `<li>${row.name}: ${row.score}</li>`                 //concat row to HTML string
   }
   return li;                                                     //return HTML-formatted text
}
```

Step 3: [view.js] Import the `Leaderboard` function from `Leaderboard.js`

scripts/view.js → imports

```
import Question from './components/Question.js';
import HUD from './components/HUD.js';
import Skip from './components/Skip.js';
import Leaderboard from './components/Leaderboard.js';
```

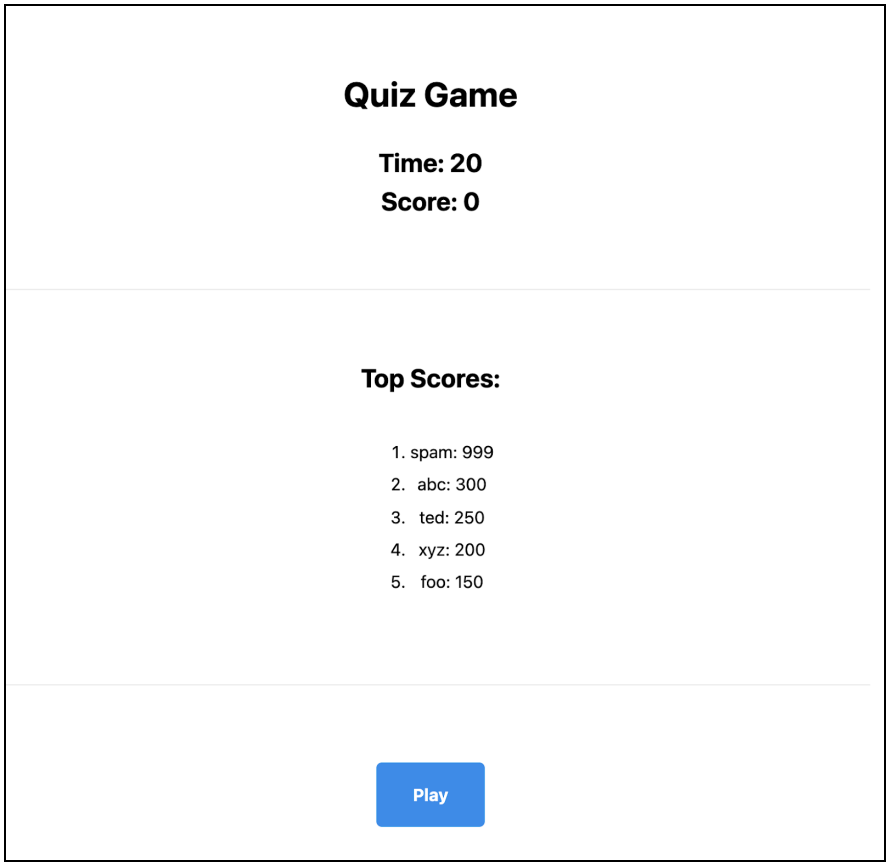
Step 4: [view.js] Refactor StartMenu to include the Leaderboard with the top scores

scripts/view.js → StartMenu

```
export const StartMenu = (props) => {
   const {timer, score, topScores} = props;
   renderDOM(
       `${ HUD(timer,score) }
       ${ Leaderboard(topScores) }
       <hr>
       <button onclick='createGame()'>Play</button>`
   )
}
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser.
The Leaderboard displays in the Main Menu.



Goal 2-4: PUT Request to update Leaderboard

'Approach' → Plan Phase

Use an HTTP PUT Request on the web server's REST endpoints to update Leaderboard data

'Apply' → Do Phase

Step 1: [http.js] Define exported function that sends an HTTP PUT Request to a REST Endpoint

scripts/http.js → sendPUTRequest

```
export const sendPUTRequest = async (url, data) => {
   const options = new Object();
   options.method = "PUT";
   options.headers = {"Content-type": "application/json"};
   options.body = JSON.stringify(data)
   const response = await fetch(url, options);
   return response
}
```

Step 2: [app.js] Initialize constants for the HTTP PUT Endpoint for the Leaderboard JSON

scripts/app.js → constants

```
const GET_TRIVIA = `https://opentdb.com/api.php?amount=1&difficulty=easy`;
const BIN_ID = '605ed0fd9ab74a5f2bcc9c02';                          //replace with your own
const GET_LEADERBOARD = `https://api.jsonbin.io/v3/b/${BIN_ID}/latest`;
const PUT_LEADERBOARD = `https://api.jsonbin.io/v3/b/${BIN_ID}`;
```

Step 3: [app.js] Global function to TEST the http PUT, *Note: delete this function after testing!*

scripts/app.js → constants

```
//Test: remove after this step
window.testPUT = async () => {
   const data = [
      {name:'A', score:30 },
      {name:'B', score:20 },
      {name:'C', score:10 },
      {name:'D', score:5 },
      {name:'E', score:0 },
   ];
   await http.sendPUTRequest(PUT_LEADERBOARD, data);
}
```

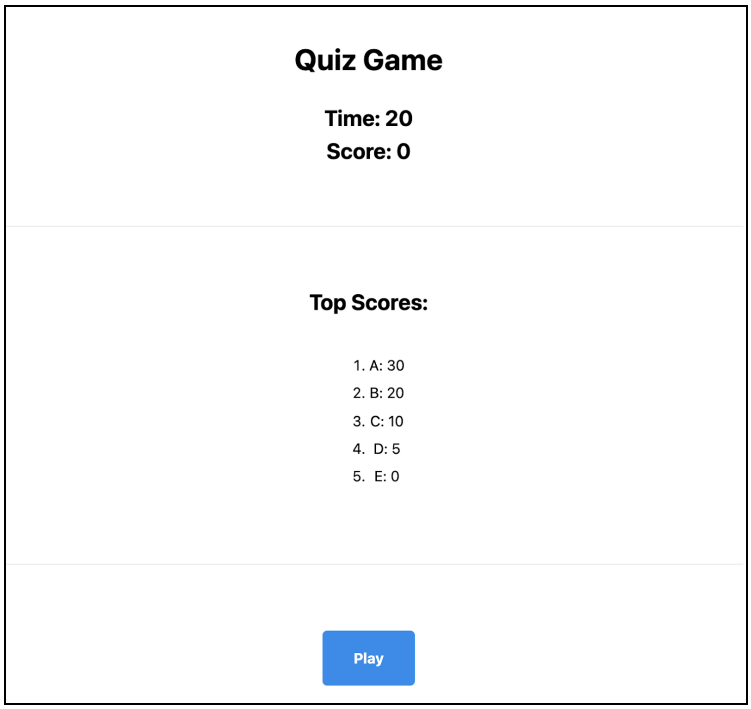
'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser.

Step 1: Dev Console
From the dev console, invoke the test function:

```
> await testPUT();
```

Step 2: Browser Viewport
Reload the browser and the top scores should be set to the test data values.



Step 3: Remove Test function
After verifying that PUT successfully worked, remove the test function from your code.

Goal 2-5: Get Name & Update Leaderboard

'Approach' → Plan Phase

When a player's score is top 5, then the game should prompt for a name & update the leaderboard.

'Apply' → Do Phase

Step 1: [app.js] Function that adds new score to the current tops 5, & removes the lowest

scripts/app.js

```
const getTop5 = async (newScore) => {
  const leaderboardJSON = await http.sendGETRequest(GET_LEADERBOARD);
  const top5 = leaderboardJSON.record;
  top5.push( newScore );
  top5.sort( (a,b) => b.score - a.score );
  top5.pop();
  return top5
}
```

Step 2: [app.js] Global function (for Event Dispatcher) to update the leaderboard on the web server.

scripts/app.js

```
window.updateLeaderboard = async () => {
  const name = document.getElementById('name').value;
  const currentScore = {name:name, score: state.score};
  const top5 = await getTop5(currentScore);
  await http.sendPUTRequest(PUT_LEADERBOARD, top5);
  start();
}
```

Step 3: [LeaderMenu.js] Function that returns the HTML for a Leader Menu component

scripts/components/LeaderMenu.js

```
const LeaderMenu = () => (                                          //Function for HTML component
   `<div>
       <h2>High Score!</h2>
       <section>
           <input id='name' type='text' placeholder='Your Name'>
           <input onclick='updateLeaderboard()' type='button' value='Submit'>
       </section>
       <hr>
   </div>`
);

export default LeaderMenu;                                          //Export LeaderMenu function
```

Step 4: [view.js] Import the `LeaderMenu` function from `LeaderMenu.js`

scripts/view.js → imports

```
import Question from './components/Question.js';
import HUD from './components/HUD.js';
import Skip from './components/Skip.js';
import Leaderboard from './components/Leaderboard.js';
import LeaderMenu from './components/LeaderMenu.js';
```

Step 5: [view.js] Helper function to determine if current score is a top 5 score.

scripts/view.js → isTop5()

```
const isTop5 = (score, top5) =>  top5.some( item => item.score < score );
```

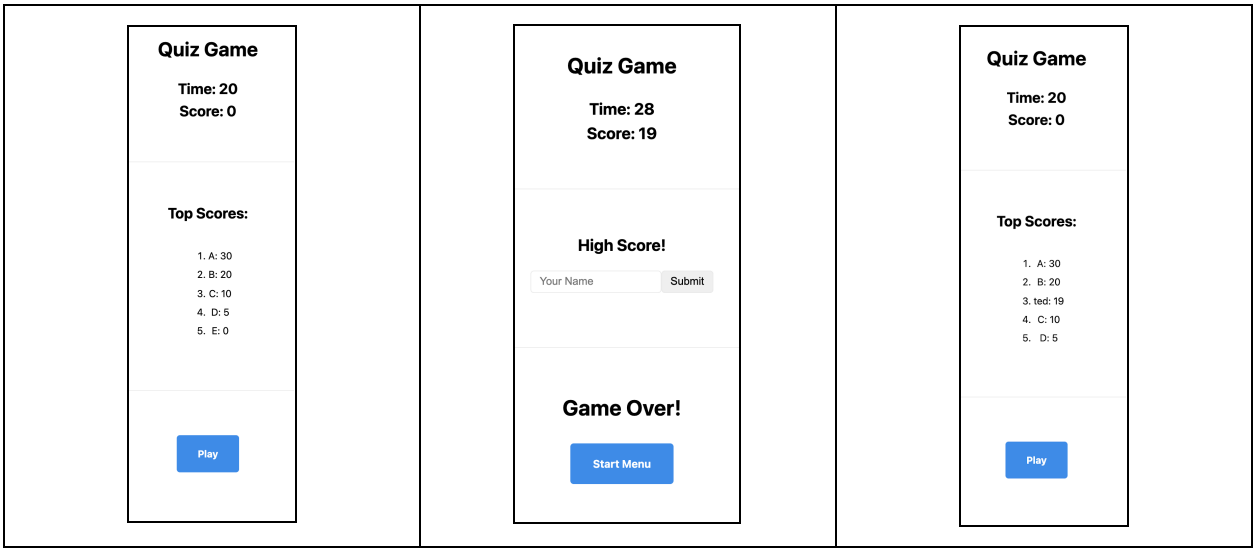
Step 6: [view.js] Refactor GameoverScene to include Leader Menu if player has a top 5 score.

scripts/view.js → GameoverScene

```
export const GameoverScene = (props) => {                          //Function for HTML view
   const {timer, score, topScores} = props;                        //Destructure properties
   renderDOM(                                                      //render the Gameover HTML to DOM
       `${HUD(timer, score)}
        ${ isTop5( score, topScores) ? LeaderMenu() : '' }
       <h1>Game Over!</h1>
       <button onclick='start()'>Start Menu</button>`
   )
}
```

'Approve' → Test Phase

Using an HTTP server, (such as the Chrome Web Server) open the `index.html` file in the browser.



Before *Leader Menu* *After*

Concluding Notes

REST Client & Modules

We designed a browser application that uses web servers to generate questions and store the top scores for the leaderboard. The browser code was organized into modules which improves readability, maintainability, and browsability. No more spaghetti code in the HTML imports!

Future Improvements

- Handling leaderboard PUT requests from a server is safer!
- Encoding issue causes incorrect evaluations if solution uses quotations

Lab Submission

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

Companion Homework

Homework:

Design your own REST Client App that uses some REST API. Your application must also use a module design pattern.

Resources: (Free REST APIs)

https://any-api.com/
https://github.com/public-apis/public-apis
https://apilist.fun/

Homework Bonus:

Showcase bonus. You can receive up to 20 bonus points if your project is outstanding and novel.
I'll publish all showcase projects on UNO's web page as a demo for future students. You should cite such projects on your resume. Please post into the SHOWCASE channel for eligibility.