

JavaScript

Lab 5: Error handling, polling

J. Pieck, L. Van Houdt, S. Van Peborgh, T. Van Nuland

1 INTRODUCTION 3

2 ERROR HANDLING 4

2.1 EMPTY DATA OR NO SEARCH RESULTS 4

2.2 INPUT VALIDATION 6

3 POLLING 7

3.1 INTRODUCTION 7

3.2 BUILDING A MINECRAFT SERVER MONITOR 7

3.3 PROJECT SETUP 9

3.4 UNDERSTANDING THE API RESPONSE 9

3.5 TRACK THE NUMBER OF ONLINE GAMERS 11

3.6 LIMITING THE NUMBER ROWS 12

3.7 SORT THE DATA ON NUMBER OF ONLINE PLAYERS 13

3.8 SORT THE DATA ON TIME THEY ARE FETCHED 14

4 SUBMITTING YOUR RESULT 15

1 Introduction

In this final lab, we will focus on making our application robust and “real-time”.

So far, we have coded for the “happy path”, assuming that the back-end always returns a list of games and that users always enter perfect data. However, in the real world, this is rarely the case. Databases can be empty, search filters might return no matches, and users often make mistakes when filling out forms. You will learn how to handle these scenarios gracefully. Instead of showing an empty screen or accepting bad data, we will ensure the application provides clear, helpful feedback to guide the user.

Additionally, our current application only updates the data when we reload the page or trigger a specific action. To make the application feel more dynamic, we will implement “polling”. This technique involves regularly asking the server for updates (e.g., every few seconds), ensuring that changes made by other users appear on your screen automatically without needing a manual refresh.

2 Error handling

In this section, we will implement *defensive programming* in two ways:

- Error handling: We will use try...catch blocks to gracefully catch network failures (like a server timeout) so the app doesn't crash.
- Input validation: We will verify user data before sending it, ensuring that invalid inputs (like empty fields) are caught early.

2.1 Empty data or no search results

When the back-end contains no data, or a search yields no results, simply displaying an empty table is not ideal. Users might find this confusing: did their search match nothing or is the application broken? To improve the user experience, you should hide the table (e.g., using the CSS property `display: none`) and show a clear, descriptive message instead. We will divide this task into two steps:

- First, we will handle the scenario where the database itself is completely empty.
- Then, we will modify the code to handle cases where a search or filter returns no results.

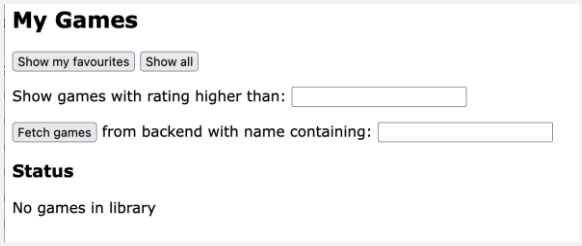


Exercise 1: Hide the table and show a message when there is no data

Update your code to handle the scenario where the game library contains no data at all.

1. Helper function: Write a function, for example `hideTable(tableId)`, that hides the table by setting its CSS style to display: none.
2. Refactor `renderGames()`: Update this function to check if the games array is empty.
    a. If it is empty: call hideTable and display a clear message in the `div#status` element.
    b. If games are present: ensure the table is explicitly set to be visible (e.g. `display: block` or remove the style) before rendering the rows.

3. Verification: Test the logic by deleting all games to ensure the error message appears, and then adding a new game to confirm the table reappears correctly.





Exercise 2: Show appropriate message when filtering gives empty result

Currently, if a search returns no results, your table correctly disappears. However, if a subsequent search does find games, the table likely remains hidden because the CSS `display:none` property is never reversed. Your task is to fix this "stuck" state by updating your `table-overview.js` file:

1. Helper Function: Write a function, `showTable(tableId)`, that sets the table's style back to display: table (which is the default display mode for table elements).
2. Refactor `renderGames()`: Update the logic to handle the scenario where games are found. In this case, the function must explicitly call `showTable` and clear any previous error messages from the `div#status`.
3. Customize the feedback message to distinguish between an empty library and a failed search (e.g., "There are no games with name containing 'abc'")

2.2 Input validation

In `add-game.js`, we will add validation to the form to ensure the user provides all necessary information. While HTML provides built-in validation (like the required attribute), we will build our own custom input validation using JavaScript for this exercise. This approach gives us full control over how errors are detected and displayed to the user.



Replace the standard HTML validation with your own custom JavaScript validation in add-game.js. First, remove any `required` or `min/max` attributes from your HTML form to ensure your JavaScript handles everything.

You must implement checks to ensure:

- ✓ No fields are empty.
- ✓ The name length is between 2 and 64 characters.
- ✓ The rating is a number between 0 and 10.
- ✓ The name is unique (does not already exist in the library).

To check if a name is unique, you must query the back-end. Sending a GET request to http://localhost:3000/games/name/:name (replacing `:name` with the input value) returns the game's ID if it exists, or `null` if it does not.

To complete this exercise, update `add-game.js` with the following logic:

- Create a helper function `addStatusError(message)` that clears any existing status and displays the new error message in red.
- Write an async function `nameIsUnique(gameName)`. This function should fetch the URL mentioned above and return true only if the response is null.
- Update your `addGame` function to read the input values and run through your validation checks using if/else statements.
- Important: Since `nameIsUnique` is asynchronous, you must use await when calling it inside your if condition.
- If a check fails, call `addStatusError` with an appropriate message and stop the function. If all checks pass, add the game and ensure the error message is cleared.

3 Polling

3.1 Introduction

To understand the need for polling, try this simple experiment:
Open your games application in two separate browser tabs. In the first tab, show the game overview. In the second tab, open the add-form and add a new game. You will notice that the new game appears immediately in the second tab, but the first tab remains unchanged unless you manually refresh the page.

We can solve this by implementing polling. This technique programs the browser to automatically fetch fresh data from the server at regular intervals, keeping the interface synchronized without user intervention. However, we will not be adding polling to the Games Library in this exercise. Doing so would require significant refactoring. Currently, your application only reads the state of filters (like the search bar or "Show favorites" button) when a user clicks something. If we introduced polling, the code would need to remember and re-apply these filters every time new data arrives from the server.

To keep things clear and manageable, we will instead practice polling on a completely new, standalone page using a public back-end API. This ensures we don't interfere with your existing code.

3.2 Building a Minecraft server monitor

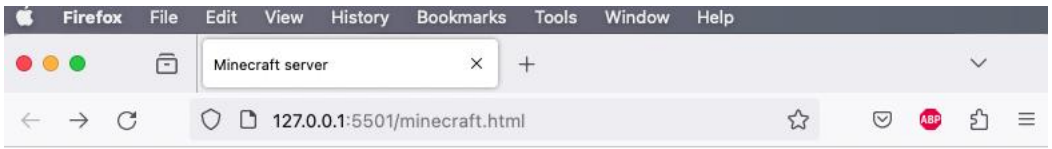
In this exercise, we will build a dashboard to monitor the real-time player count on the popular Minecraft server mc-central.net. To retrieve this data, we will use a public API: https://api.mcsrvstat.us/3/mc-central.net. This API provides a snapshot of the server's current status.

Our application will implement polling to fetch this data at short intervals (e.g., every 5 seconds). We will record these snapshots and display them in a dynamic overview table,

allowing the user to track how the number of online players evolves over time. Additionally, we will add functionality to sort and analyze this recorded data.



You can also use this API to request data from other Minecraft servers. On the API website https://mcsrvstat.us/ you can find a list of servers that are currently online.



Minecraft server

This table shows the number of players currently online on snapcraft.net.

Generated since 14:22:57

Limit lines of data to

○ 2 ◉ 5 ○ 20 (default)

Sort data on

When both time and number of players are checked: data is first ordered on number of players, next on date.

○ time (oldest values first) ◉ time (newest values first)

☑ number of players (highest numbers first)

Date	Time	Players	Max number of players
20/11/2023	09:16:30	359	400
20/11/2023	20:25:30	350	400
20/11/2023	11:25:30	350	400
20/11/2023	14:24:36	232	1750
20/11/2023	14:24:34	232	1750

3.3 Project setup

To prepare for the Minecraft polling exercise, you need to integrate the new files into your existing project structure and update your navigation.

- Add the provided `minecraft.html` file to the appropriate location within your project directory (typically the root).
- Copy the content of the provided `css/style.css` file and integrate it into your main project CSS file. Alternatively, you may link the new CSS file separately in your HTML header.
- Open your main navigation file (usually the main HTML file) and add a new menu item. This link should direct users to the `minecraft.html` page and be clearly labeled (e.g., “Minecraft status”)



You can write the CSS for the given HTML file yourself, so that it matches the rest of your games library. This will help you practice your HTML/CSS skills.

3.4 Understanding the API Response

Before writing any code, it is important to understand the data structure we are working with. Navigate to the following URL in your browser: https://api.mcsrvstat.us/3/mc-central.net.

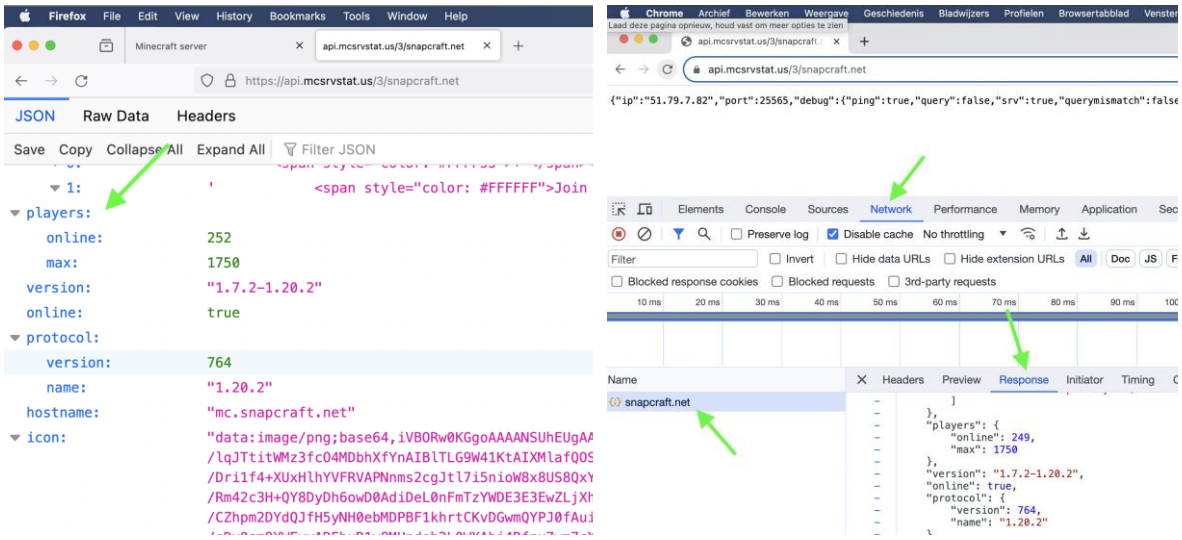
Just like our own games back-end, this server returns a JSON object. Inspect the response and locate the players object. Inside, you will find two crucial fields:

- `online`: The number of players currently connected.
- `max`: The maximum server capacity.

Since this is live data, these values change constantly.

How to inspect the JSON:

- Firefox: Automatically formats the JSON in the browser window for easy reading.
- Chrome: May display raw text. To view the structure clearly, open Developer Tools, go to the Network tab, refresh the page, select the request, and click the Preview tab.



Retrieving this data with JavaScript works exactly like interacting with our games back-end:

- Fetch: Use `fetch(url)` to send a GET request to the API.
- Parse: Use `response.json()` to convert the received JSON into a usable JavaScript object.
- Read: Use dot notation to read specific fields within that object. For example, if your object is named result, you read the player count via `result.players.online`.

Remember to use the `await` keyword to pause execution while waiting for the `fetch()` and `.json()` operations to complete. This requires your function to be marked as `async`.

```
const response = await fetch("https://api.mcsrvstat.us/3/...");
const result = await response.json();
console.log(result.players.online);
```

3.5 Track the number of online gamers

In this section, we're going to implement "polling" to track the number of players on a Minecraft server in real-time. You can use the `setInterval()` function for this.

The function `setInterval()` expects as first parameter a function that is called when a time interval is passed. That interval is the second parameter (in millisconds).
The code snippet above triggers the function `fetchAndRenderAnimals` to be called and executed every 1000ms.

```
setInterval(fetchAndRenderAnimals, 1000);
```



Create a new file named `js/minecraft.js`. Inside this file, you will implement the following logic:

- Initialize an empty array named `data`. This will store the history of your server checks.
- Write a function `renderData()`. This function acts just like your previous render functions: it clears the current table body and generates a new row for every item in the data array. The newest data should appear at the top of the table. You can verify this works by temporarily adding dummy data to your array.
- Write an async function `fetchData()`. This function must:
    - Send a `GET` request to the Minecraft server API.
    - Create a new JavaScript object with three specific properties: `time` (created using `new Date()`), `players` (the current number of online players from the response), and `maxplayers` (the server's maximum capacity).
    - Add this new object to the *beginning* of your global `data` array using the array method `.unshift()`, or use the array spreading syntax.

- Create a wrapper function `fetchAndRenderData()`. This function calls `fetchData()` (remember to `await` it) and immediately follows it with `renderData()` to update the screen.
- Finally, start the polling process. Use the JavaScript `setInterval` function to execute `fetchAndRenderData` every 5000 milliseconds (5 seconds).

3.6 Limiting the number rows

Currently, your table grows indefinitely. Your next task is to make the radio buttons on the form functional, allowing the user to choose whether to display the last 2, 5, or 20 lines of data. The default should be 20.

To achieve this, we first need to understand how to select radio buttons in JavaScript.

- **Grouping:** Notice in the HTML that the three input fields share the same name attribute (`numberOfLines`). This ensures that only one button in the group can be checked at a time.
- **Selecting:** In JavaScript, just like in CSS, you can use the `:checked` pseudo-class to find the active button.
- **Reading Value:** You can access the value of the selected button using the `.value` property.

The following code selector targets the currently selected radio button in the group:

```
document.querySelector('input[name="numberOfLines"]:checked')
```

Integrate the logic to control the amount of history shown in the player table.

- ✓ Create a function `readNumberOfLines()` that returns the number of lines that should be shown, depending on which radio button is checked.
- ✓ Refactored the function `renderData()` such that only the desired number of lines are shown.
- ✓ Use polling: it is not necessary to use `click` events since the browser window is refreshed every 5 seconds.

3.7 Sort the data on number of online players

Make the “sort on number of online players” checkbox functional.

- ✓ When the checkbox is checked, the table must display the data ordered by player count, with the highest numbers at the top. If unchecked, the data remains in its default chronological order.
- ✓ To check the box's status, use the `.checked` property of the DOM element (e.g., `document.getElementById("sort").checked`), which returns a `boolean`.
- ✓ Inside `renderData()`, use the `.toSorted()` method. Unlike the standard `.sort()`, this creates a new, sorted copy of the array, leaving your original history intact.
- ✓ You must pass a comparison function to `.toSorted()`. This function takes two parameters (e.g., a and b) and compares their player counts to determine the correct descending order. (see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/toSorted)

3.8 Sort the data on time they are fetched



Implement the logic for the “sort data on time” radio buttons.

- ✓ **Multi-level Sorting**: You must handle the scenario where multiple sorting criteria are active. If both the “number of players” checkbox and the “time” sorting are selected, the data must be ordered primarily by the number of players. The time should serve as a secondary sorting criterion, used only to determine the order of entries that have the exact same player count.
- ✓ **Date Comparison**: Remember that JavaScript Date objects can be compared directly using standard comparison operators (like < and >).
- ✓ **Checking State**: Continue to use the `.checked` property to determine if the relevant DOM elements (radio buttons or checkboxes) are active.



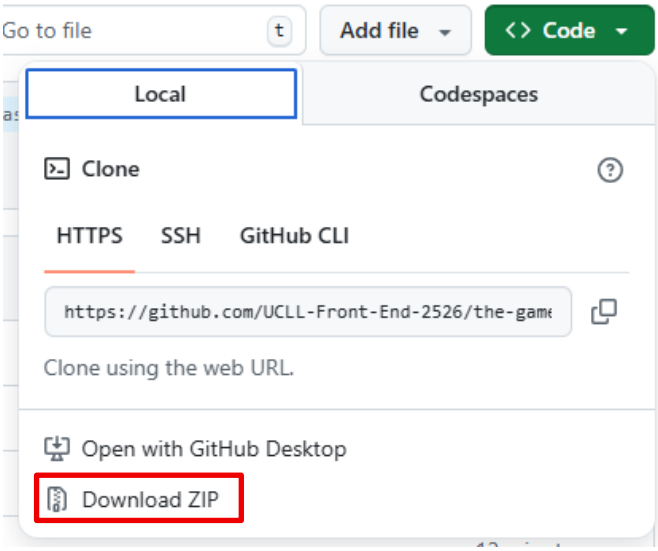
You can transform this task into an in-depth exercise on click events. To do this, disable the automatic polling (`setInterval`), create a static dummy array with test data, and refactor your code so that the table only updates when the user manually clicks the sorting or limit buttons.

4 Submitting your result

To complete the lab, you must submit a ZIP file of your project via the assignment on Toledo. Important: do not submit a link to your GitHub repository. Since you will continue to build on this same codebase in future labs, we need a "snapshot" of your code at this specific stage for grading.

The easiest way to create this snapshot is directly from GitHub:

- Navigate to your GitHub repository in your browser (e.g. https://github.com/UCLL-Front-End/the-games-library-jhnpck).
- Click the green "<> Code" button.
- Select Download ZIP from the dropdown menu.



Submit the downloaded ZIP file on Toledo to finalize your submission.