

JavaScript

Lab 4: Asynchronous web model

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

1 INTRODUCTION 3

2 SETTING UP THE BACK-END 4

2.1 STARTING THE WEBSERVER 4

3 FETCHING DATA 6

3.1 SYNTAX 6

3.2 FETCHING GAMES FROM THE BACK-END 7

4 SERVER-SIDE FILTERING 9

5 ADDING NEW DATA 11

5.1 SENDING A POST REQUEST 11

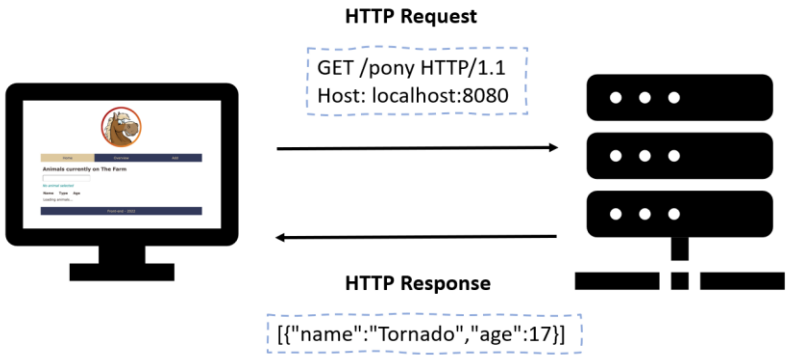
5.2 COMBINING EVENTS AND POST REQUESTS 13

6 DELETING DATA 15

7 SUBMITTING YOUR RESULT 16

1 Introduction

Up to now, our games were hard-coded in a JavaScript file. In a real-world application, data is stored on a database server and accessed via a back-end. The process works as follows: the front-end sends an HTTP request to the back-end, which performs the requested operation and then sends back an HTTP response.



In this course, we will focus exclusively on the front-end. We will use JavaScript to send the necessary requests to a given back-end. Our front-end application will then process the responses it receives and render the data in the browser. You will learn how to code the back-end application in the “Back-end Development” course, but that's for later.

We will start this lab by replacing our hard-coded games array with data we fetch from the back-end via an HTTP request. The rest of the existing functionality should not be affected by this change. Then, we will add new functionality that requires requests to the back-end, such as search, add, and delete operations.

This process will involve both refactoring existing code and adding new code. Always try to follow the DRY (Don't Repeat Yourself) principle. If you find two functions that do roughly the same thing, you can usually reduce them to one function that handles both tasks using parameters.

2 Setting up the back-end

We provide a back-end application called "The Games Library". This is a server that manages a collection of games. It handles requests to add, view, search, and delete games. In fact, it provides operations that you have previously coded yourself on the front-end.

The games themselves are not stored in an external database but are hard-coded into the application. While we aren't working with a "real" database, this is not a problem for the front-end. The front-end never connects directly to a database; it only sends requests to the back-end server. It doesn't matter where the server gets its data. The only disadvantage is that all data will be lost when the back-end is restarted.

"The Games Library" is an application that runs in the NodeJS runtime, which is an environment that allows JavaScript to run outside the browser. We will run this application locally, meaning your own computer will act as a back-end server. A browser can send HTTP requests to this local server and receive HTTP responses. Applications on your computer can access this local server via http://localhost:3000. Other people will not have access to it.

2.1 Starting the webserver

You do not need to understand the back-end's code. Below, we explain how to start the application and what requests it can handle.

1. Download The Games Library back-end (ZIP-file from Toledo) and extract it in a folder *outside* of your front-end project's working directory.
2. Open a new window in VS Code and open the folder you just extracted.
3. Open the `readme.md` file. VS Code provides a nice preview for markdown files. To use it, open the `.md` file and look for the "Open Preview to the Side" icon in the top-right corner of your window.
4. Install NodeJS on your system as explained in the `readme.md` file (if you haven't already).

The downloaded application cannot be run like a simple `.exe` file. For efficiency, not all required files (known as dependencies) are included in the zip. This is standard practice for

NodeJS applications. You can use `npm` (Node Package Manager) to automatically download and add these missing files.

Follow these steps:

1. Open a terminal (like Git Bash, Cmd, or your built-in VS Code terminal).
2. Navigate to the project root folder of the back-end application you downloaded. For example:

```
$ cd C:/Projects/TheGamesLibrary-Backend
```

3. Run the following commands one by one. The first command installs all dependencies (this may take a minute), and the second one starts the server:

```
$ npm install
$ npm start
```

This will start the back-end application on http://localhost:3000. The server will keep running in your terminal. To stop the application, you can press `CTRL+C` in that terminal window.

You can access the API documentation and test all available endpoints via Swagger, which is running at http://localhost:3000/. Swagger is a tool that creates an interactive user manual for your software, helping developers understand and test how different programs can talk to each other.

3 Fetching data

We will now replace our hard-coded games array in `overview-table.js` with live data fetched from the back-end server. Our first step is to use JavaScript to send an HTTP request to the back-end. JavaScript will then convert the server's HTTP response into an array of games. The good part is that this new array can be used by your existing code, like the `renderGames` function, without requiring any other changes.

To do this, make sure you have two VS Code windows open: one running the back-end server and a second window for your front-end project. You will continue working in your `table-overview.js` and `table-overview.html` files.

3.1 Syntax

As in the other chapters, we start by showing the demo code of the "Farm" project. The following code snippet queries the back-end for all the animals and stores them in the local `animals` array.

```
const animals = [];

const fetchAnimals = async () => {
    const response =  await fetch("http://localhost:3000/animals");
    const result = await response.json();
    animals.push(...result);
};
```

We first initialize an empty animals array. The `fetch()` function sends an HTTP GET request to the back-end. We use `await` to pause the function's execution until `fetch` completes and returns a response. You can only use the `await` keyword inside a function that is explicitly marked as `async`.

The HTTP response body must be converted from JSON into a JavaScript array. This is also an asynchronous operation, so we must `await` the `response.json()` method. Finally, we use the spread operator (`...`) to push all elements from the `result` into our local `animals` array.

3.2 Fetching games from the back-end

The next step is to create an overview of the animals. We previously defined a `renderAnimals()` function that creates a table row for each animal and assigns events (hover, click, etc.). We can reuse this function because its logic hasn't changed; only the source of the data is different. To ensure that rendering only starts *after* all animals are loaded, we combine these two steps in a new `async` "wrapper" function:

```
const fetchAndRenderAnimals = async () => {
    await fetchAnimals();
    renderAnimals();
};
fetchAndRenderAnimals();
```



Replace the hard-coded games array in `table-overview.js` with data fetched from the back-end.

Step 1: Fetch the data

Start by initializing an empty `games` array at the top of your `table-overview.js` file.

Next, create an `async` function named `fetchGames`. This function is responsible for:

1. Fetching the game data from the back-end.
2. Converting the response with `.json()`.
3. Pushing the results into your empty `games` array.

Step 2: Render the Data

Create a new async "wrapper" function named `fetchAndRenderGames`. This function must orchestrate the two steps:

1. First, it should `await` the `fetchGames` function (to wait until the data has arrived).
2. Then, it should call your existing `renderGames` function to display the (now populated) array

Step 3: Execute and Clean Up

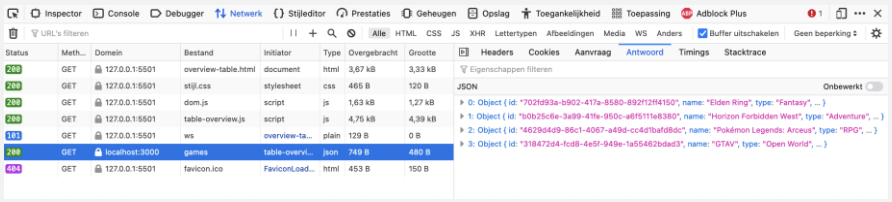
At the very bottom of your file, call your new `fetchAndRenderGames` function to start the entire process.

Remember to remove any old, direct calls to `renderGames()` that are now redundant.

Step 4: Verify

Once you're done, test the page. All existing functionality (filters, row click events) should still work. The only change is *how* the `games` array is initially populated.

- **Tip 1**: Open the "Network" tab in your developer tools. You should see a successful GET request to localhost:3000.



- **Tip 2**: If your "Show favorites" filter suddenly returns nothing, this is likely correct. The data from the server doesn't have any games marked as a favorite yet.

4 Server-side filtering

In the previous lab, you created a "rating higher than" filter. This filtering was implemented client-side. This means your JavaScript fetched all the games first and then filtered the local JavaScript array in the browser before re-rendering the table. This approach is inefficient if the database contains thousands of games. It is not recommended to fetch such a large amount of data just to filter it afterward. In such cases, we should ask the back-end to do the filtering and send only the appropriate data.

In this section, you will create a function that searches for data on the back-end. The back-end already has an implementation for this. You can test it by trying the following URL in your browser: http://localhost:3000/games?query=e. The response will show all games in the back-end with a name containing the letter “e”. Try other values. The keyword `query` in the URL tells the back-end what to search for, and the value after the `=` is the search term it will use as a parameter.



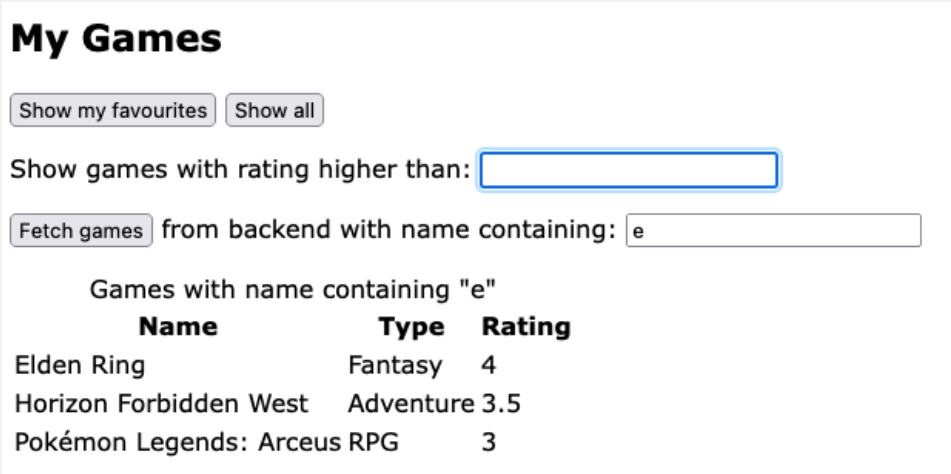
Add server-side search functionality.

1. In `table-overview.html`, add an `input` field for the search term, a `button` with the text "Fetch games", and a `<caption>` element inside your `<table>`.
2. In `table-overview.js`:
    a. Write a function `searchByFetch(chars)` that fetches only games containing "chars" in their name from the back-end, using the specified URL http://localhost:3000/games?query=chars.
    b. Write a function `searchByFetchAndRender()` that reads the search value from the input field and renders the selected games in the table.
    c. Add a `click` event listener to the "Get games" button. When clicked, this function should trigger the search and render, and you should see a GET request to localhost:3000.

    d. Add JavaScript so that the table caption contains the correct information ("All games" by default, and "Games with name containing 'chars'" after a search).
3. Your other functionality ("Show favorites", "Rating higher than") should still work. They will now operate only on the fetched (filtered) data. When this runs, only the table data should change; other elements (like the contents of `div#status`) should remain the same.

After implementing the search, you should refactor your existing code.

- Notice that your original `fetchAndRender()` and your new `searchByFetchAndRender()` functions do almost the same thing. Refactor both of these into one single, flexible function that can handle both fetching all games and fetching filtered games.
- Refactor the "Show all" button. It should now use your new, refactored function to reset all filtering and display the complete list of games.



5 Adding new data

Until now, we could only see games that were already in the back-end. We are now going to make it possible to add new games via a form in the front-end.

5.1 Sending a POST Request

There are several types of HTTP requests. In the previous exercises, we fetched animals from the back-end using a GET request. This is the standard way of fetching data. If we expect the status on the back-end to change (for example, because we add or change an animal), we need to send a POST request. An extra `method` property is added to the `fetch()` function.

```
const adoptAnimal = async () => {
    const response = await fetch(
        "http://localhost:3000/animals/adopt",
        { method: "POST" }
    );
};
```

The browser can send data with the request to the back-end. This can be done via the URL (as a query string). More complex data, such as objects, are sent in the `body` of the HTTP request. Additional properties are therefore required in the `fetch()` function.

```
const addAnimal = async () => {
    const animal = { name: "Woef", type: "Dog", age: 12 };

    const response = await fetch("http://localhost:3000/animals",
        {
            method: "POST",
            headers: {
                "Accept": "application/json",
                "Content-Type": "application/json",
            },
            body: JSON.stringify(animal),
        }
    );
};
```

The `headers` field tells the back-end that the data we are sending in the body is in JSON format. The `body` property contains the data, but it must be a string. The `JSON.stringify()` function converts our JavaScript object into a JSON-formatted string, which can be sent in an HTTP request. Note that the URL http://localhost:3000/animals is identical to the URL we used to GET the animals. The back-end recognizes the request method (GET vs. POST) and knows that it needs to handle them differently.



Create a new page that allows users to add games to the back-end via a form.

First, create two new files: `add-game.html` and `add-games.js`.

- In `add-game.html`, you must create a valid HTML form.
  - This form needs fields for "name", "type", and "rating".
  - Use the correct `input` type for each field (e.g. `number` for rating).
  - Make sure to use `label` elements and correctly link them to their corresponding `input` fields using `id` and `for` attributes.
  - Below the form, add a `div` with the `id` "status" to display status messages.

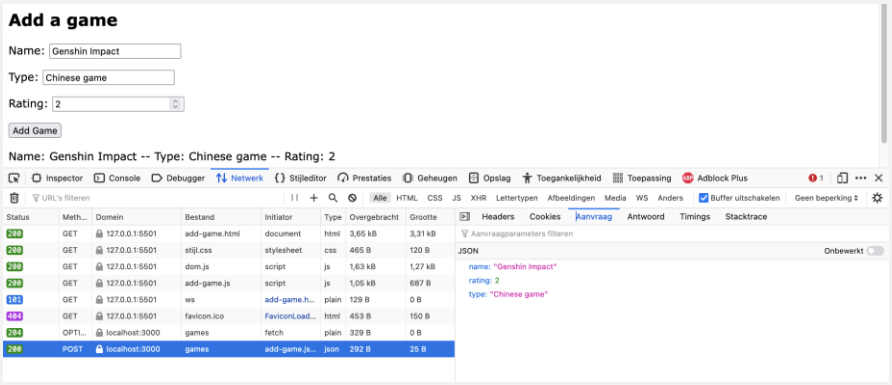


- Remember to link your `add-games.js` file to your new HTML file.

In your `add-games.js` file, you will write the logic to handle the form.

- Add a `submit` event listener to the form.
- The first line of your event handler must be `event.preventDefault()` to stop the browser's default form submission.
- This event handler should call a new `async` function, for example `addGame()`.
- This `addGame()` function must do the following:
  - Read the current values for name, type, and rating from the form's input fields.

- Create a new game object. This object should *only* contain these three properties (do not include a `isFavorite` field).
- Send an HTTP POST request to the back-end at http://localhost:3000/games. This request must include the new game object as a JSON string in the request `body`.
- After the request is complete, the back-end will send a response (the name of the added game). You must display this response as feedback in the `div#status` element.



To verify your work, after submitting a new game, navigate back to your `table-overview.html` page. When the overview page reloads, your newly added game should appear in the table.

5.2 Combining events and POST requests

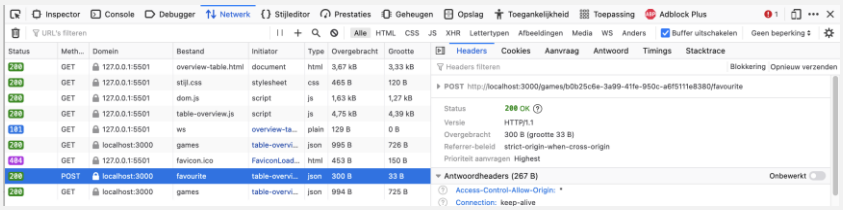
We will now combine event handling with data retrieval. This exercise requires you to use the `id` that the back-end automatically assigns to each game object. This `id` is a unique key (e.g., `game.id`) that allows us to modify a specific entry on the server.

Add a `dblclick` (double-click) event listener to each table row. When the user double-clicks a game, its `isFavorite` flag should be toggled on the back-end.

You must implement this in `table-overview.js` as follows:

- Update your `renderGames` function to add a `dblclick` event listener to each `<tr>` element.

- This listener should call a new async function, for example `toggleFavorite(game)`.
- This `toggleFavorite(game)` function must:
    - Send an HTTP POST request to http://localhost:3000/games/:id/favorite.
    - You must replace `:id` in the URL with the specific `id` of the game (e.g., using `${game.id}`).



- After the POST request is complete, the function must immediately call `fetchAndRenderGames()` to refresh the table with the updated data.
- When you test this, your browser's network tab should show two requests for each double-click: a POST request (to modify the game) and a GET request (to refresh the table).
- (Extra) The response from the POST request is the name of the game. You can use this to display a confirmation message in the `div#status`, such as "The game with name … is now (not) my favorite".

6 Deleting data

Your application currently sends POST and GET requests to the back-end. You will now expand on this by using a DELETE request to enable users to delete a game. This will require you to refactor the code in both `table-overview.html` and `table-overview.js`.

Implement the delete functionality.

- You must update your `renderGames` function in `table-overview.js`. Inside the loop, add a new table cell (`<td>`) to each row and place a "Delete" button inside it.
- Add a `click` event listener to this new button. This listener must call a new `async` function, for example `deleteGame(game)`.
- This `deleteGame(game)` function must:
    - Send a `fetch` request using `method: 'DELETE'` to the URL http://localhost:3000/games/:id, replacing `:id` with the `game.id`.

Name	Type	Rating	Delete
Elden Ring	Fantasy	4	Delete
Horizon Forbidden West	Adventure	3.5	Delete
Pokémon Legends: Arceus	RPG	3	Delete
GTAV	Open World	5	Delete

    - After the request is complete, it must call `fetchAndRenderGames()` to refresh the table with the new data.
- Extra: You can add a message to the `div#status` to confirm the deletion, e.g., "The game with name … is now deleted."

Extra Challenge: Refactor for reusability

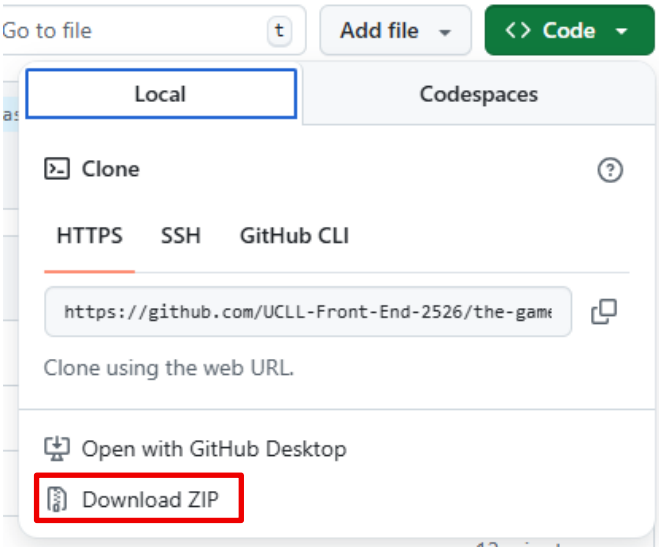
- You will notice that your new `deleteGame` function and the existing `toggleFavorite` function are very similar.
- Try writing a single, generic `async` function that can handle both operations. This new function could accept parameters like the `game` object, the specific `URL`, and the `request method` ('POST' or 'DELETE').

7 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.