

JavaScript

Lab 3: DOM manipulation & Event-handling

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

1 INTRODUCTION 3

2 DOM MANIPULATION 4

2.1 CREATING AND ADDING HTML ELEMENTS 4

2.2 CREATING TABLES 6

3 EVENT HANDLING 8

3.1 RESPONDING TO USER ACTIONS 8

3.2 DATA FILTERING 10

4 SUBMITTING YOUR RESULT 13

1 Introduction

With the statistics page complete, we're ready to build a truly interactive web page. By the end of this chapter, you will have a complete game overview displayed in a table. You'll be able to click any game to see more details about it, and we will also create buttons that allow you to filter the games.

To get started, create a new HTML file named `table-overview.html` and add the standard page structure (like the navigation and footer) just as you did before. Next, create a new JavaScript file named `table-overview.js` and include it in your new HTML file. You must also include `dom.js`. It's not necessary to include `games.js`. From now on, all JavaScript code for this module should be written in `table-overview.js`, as it will control the content of `table-overview.html`. All screenshots in this lab will refer to this new file.

2 DOM manipulation

2.1 Creating and adding HTML elements

We can add, update, and remove HTML elements using JavaScript. This process is called DOM manipulation.

```
const footer = document.createElement("footer");
footer.innerHTML = "some text";
document.querySelector("body").appendChild(footer);
footer.className = "message";
```

Breakdown:

- First, `document.createElement("footer")` creates a `<footer>` element in memory.
- Next, `footer.innerHTML = "some text"` sets the HTML content inside the new element.
- Then, `document.querySelector("body").appendChild(footer)` selects the `<body>` element and adds the new footer as its last child, making it visible on the page.
- Finally, `footer.className = "message"` sets the element's class attribute. Let op: this will overwrite any existing classes on the element.

Note on `querySelector()`: We use `document.querySelector()` because it's a flexible selector. It returns the first element that matches any valid CSS selector (like a tag name `body`, an ID `#myId`, or a class `.myClass`).



Write the JavaScript code necessary to create a `div` element with its id set to "status". Inside this new `div`, you must also create and append an `h3` element containing the text "Status". The resulting DOM in table-overview.html should look like this:

```
<main id="main">
  <h2>My Games</h2>
  <div id="status">
    <h3>Status</h3>
  </div>
</main>
```



Not seeing the "Status" title appear on your page? Your first step is always to open the developer tools and check the Console tab. If there are red error messages, they will tell you what's wrong (e.g., a typo).

However, if the console is empty, but your elements are still missing, it usually means your script isn't being loaded or run correctly. Check these two things immediately:

1. Is your new table-overview.js file correctly linked in your table-overview.html file?
2. Did you remember to add the defer attribute to that <script> tag? (Without defer, your script will likely run before the <body> element exists, and it will fail silently.)

If these steps don’t resolve your issue, try debugging your code using developer tools.

2.2 Creating tables

Just as we dynamically added the footer and `h3` elements to the DOM, we can now create a table to display our game collection. This is a much cleaner and more structured approach than writing all the content in paragraphs, as we did in the previous lab. Using an HTML `<table>` is the semantically correct way to present this kind of data.



Exercise 1:

Add the main `<table>` structure to your `table-overview.html` file. This table requires a header section (`<thead>`) containing one row (`<tr>`) with three table headings (`<th>`): Name, Type, and Rating.

After the header, you must also add a table body (`<tbody>`). This element should be empty for now, but you must give it the specific ID `my-games-table-body`. This ID is crucial as it allows our JavaScript to select this element and populate it with game data later.

The result should look like this:

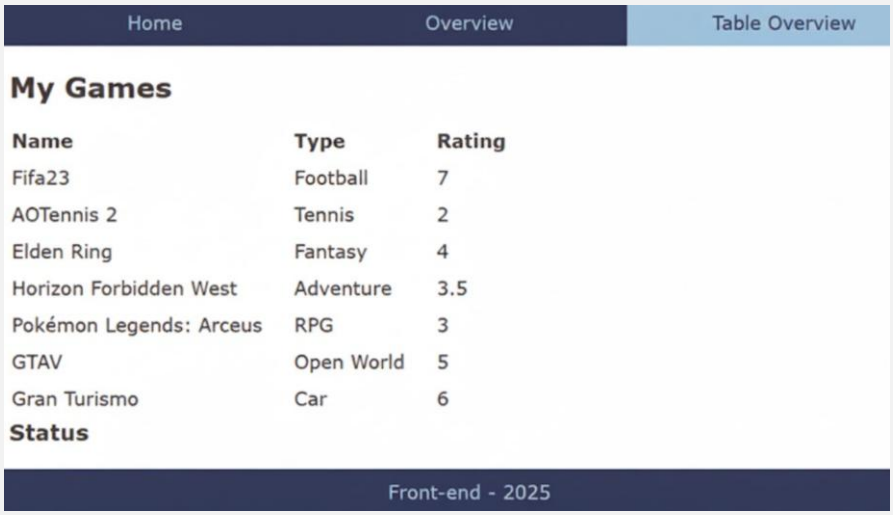
```
<main>
  <h2>My Games</h2> overflow
  <table> overflow
    <thead>
      <tr>
        <th>Name</th> overflow
        <th>Type</th> overflow
        <th>rating</th> overflow
      </tr>
    </thead>
    <tbody id="my-games-table-body"></tbody> overflow
  </table>
  <div id="status"> ··· </div>
</main>
```

Exercise 2:

Now that we have the table structure, we'll write a function to automatically populate it with our game data. We will loop through the games and dynamically create a new row in the `<tbody>` for each one.

1. First, copy your `games` array from games.js and paste it at the top of `table-overview.js`.
2. Next, write a function `renderGames(gamesArray)` in `table-overview.js`.
3. This function must iterate over the `gamesArray`. For each game object, it should:
    a. Create a new table row element `(<tr>)`.
    b. Create and append three table data cells (<td>) to this new row, one each for the game's Name, Type, and Rating.
    c. Append the completed row to the tbody#my-games-table-body element.
4. Finally, call your new renderGames() function, passing your games array as the argument.

The result should look like this:



3 Event handling

3.1 Responding to user actions

JavaScript can be used to make a web page more interactive. You can define effects that happen when the user performs an action, such as hovering with the mouse, clicking, etc.:

```
tableRow.addEventListener("click", () => selectAnimal(animal));
tableRow.addEventListener(
   "mouseover", () => tableRow.className = "select"
);
tableRow.addEventListener("mouseout", () => tableRow.className = "");
```

The `addEventListener` function attaches a "listener" to an HTML element, allowing it to respond to user actions. It requires two main arguments:

1. The `event type` (a string specifying the action to listen for, e.g., "`click`", "`mouseover`", "`mouseout`").
2. The `function` to execute when the event occurs. This is often called a "callback function" or "event handler."



Exercise 1:

Make the `h2` title interactive. You need to add a `click` event listener to the `h2` element.

Select the `h2` element using `document.querySelector("h2")`, you are not allowed to add an `id` or `class` to it.

Inside the event handler function, you must:

1. Generate a random whole number between 0 and 360 using `Math.floor(Math.random() * 361)`.
2. Create a color string using HSL format: `hsl(number, 100%, 50%)`, where number is your random number.
3. Set the `style.color` of the `h2` element to this new HSL color string.

You can use the “style” property to change the background colour:
https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/style

Exercise 2:

Make each game row in your table interactive.

You need to update your `renderGames` function. Inside the loop, immediately after creating the `<tr>` element for a game, you must attach a `click` event listener to that row.

When a user clicks that row, the event handler function should:

1. Call your `gameToString(game)` function, passing in the specific `game` object for that row.
2. Set the `innerHTML` of the `div#status` element to the string returned from `gameToString(game)`.



3.2 Data filtering

We can increase the page's interactivity by allowing users to filter the data, either with a text field or by clicking buttons. Take a look at the “Farm” exampled code in the demo clips. An input field with the `id="field"` was added. When a user types into that field, we re-render the list, but this time only showing animals whose names match the search.

```
const renderAllAnimals = () => {
  const chars = document.querySelector("#field").value;
  animals
    .filter((animal) => animal.name.toLowerCase().includes(chars.toLowerCase()))
    .forEach((animal) => {
      ...
    });
}
```

Breakdown:

- `const chars = ...`: This line reads the current text typed by the user in the #field input.
- `.filter(...)`: This method filters the `animals` array.
- `.toLowerCase().includes(...)`: This is the key to a good search. By converting both the animal's name and the search text to lowercase, we ensure the filter is `case-insensitive` (e.g., searching for "w" will find "Witcher").
- `.forEach(...)`: Finally, `forEach` is chained to the *new, filtered array* to display only the animals that passed the test.

Exercise 1:

Implement game filtering by adding two buttons and refactoring your `renderGames` function. First, add two `<button>` elements to `table-overview.html`: one named "Show my favorites" and another named "Show all".

Next, you must refactor your `renderGames()` function in `table-overview.js`.

- Modify the function so it accepts a new parameter (e.g. `filterFunction`).
- Inside the function, you must use this parameter with the `.filter()` method (e.g. `games.filter(filterFunction)`) before iterating over the array to render the rows.
- You must set up a default behavior. If no `filterFunction` is provided when `renderGames` is called, your function must show all games.

Finally, add click event listeners to your new buttons:

- The "Show all" button should call your refactored `renderGames()` function in a way that triggers the default (show all) behavior.
- The "Show my favorites" button should call `renderGames()` and pass it a function that correctly filters for only the favorite games.



Not working? Here are some troubleshooting tips.

- Check the browser console first. A syntax error is the most common problem.
- Inspect the DOM. Use the developer tools to see if the click event listeners are actually attached to your buttons.
- Check how you are passing the `filterFunction`. This is the most complex part. Are you passing a function definition (e.g. `() => game.isFavorite`)? Remember, you cannot pass a pre-calculated value (e.g. `game.isFavorite`), as `.filter()` needs a function to run.
- Finally, check your click event handler. Are you actually calling your `renderGames` function from *inside* the click listener? Nothing will appear on the screen if the function isn't called.
- If these steps don’t resolve your issue, try debugging your code using developer tools.

Exercise 2: Your final task is to add a real-time rating filter.

- Add an input element to your `table-overview.html` file. Give it an `id` (e.g. `rating-filter`) and set its type to number.
- In your `table-overview.js` file, add an `input` event listener to this new field. This event is required as it fires instantly while the user is typing.
- This event listener must call your `renderGames` function and pass it a `filterFunction`.
- This `filterFunction` must read the current value from the rating input field and filter the games array, returning only those games with a rating higher than or equal to the number the user entered.

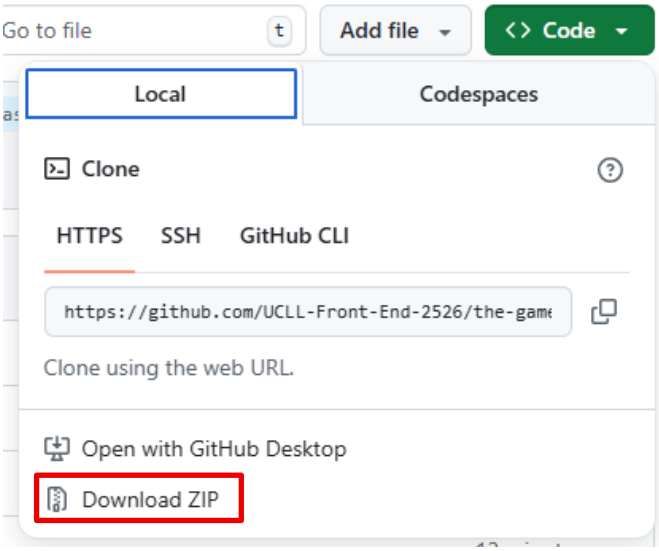


4 Submitting your result

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.