

Fetch on Demand with Forms

- Due No Due Date
- Points 1
- Submitting a website url

 (<https://github.com/learn-co-curriculum/phase-1-js-fetch-on-demand-with-forms>)  (<https://github.com/learn-co-curriculum/phase-1-js-fetch-on-demand-with-forms/issues/new>)

Learning Goals

- Practice using forms to trigger `fetch` requests
- Add content to the DOM based on user input
- Override default form behavior using JavaScript

Introduction

In the previous lab, we accessed a remote API on page load and populated some content. This is a common practice in modern web development — when you visit a website, that site may fetch data from a variety of places right as the page loads. Sites often retrieve data from a backend (like your account info) while also retrieving data from other sources like APIs. From a user's perspective, it all just loads when they visit the site.

Now, we're going to look at a slightly different scenario — retrieving data based on user input.

The underlying code isn't much different than what we've seen. We'll pass a function into an event listener that, when called, sends a `fetch` request, then does something with the retrieved data.

For this code-along, we'll use JSON server as a mock API and build out a form to get specific data from our API.

Scenario

Imagine you've been hired to help build the frontend for a movie database company specializing in kids' movies. Some initial work has already been done for you — we have an API with some starter data and some initial HTML, but very little JavaScript.

Getting Started

First, let's get the JSON server up and running in the background. From inside this assignment's local directory, run `json-server --watch db.json` to start the JSON server. In your browser, you can verify the server is running by navigating to `http://localhost:3000/movies` to see the API data.

Leave the server running for now. Open a second terminal window and navigate to this assignment again. We'll use this second window to open files in your browser or text editor while the server is running.

Open `index.html` in your text editor and in the browser (`open index.html` for Mac, `explorer.exe index.html` for WSL). With everything set up, we can take a look at the HTML we currently have.

Existing HTML

In `index.html`, the movies from our database are currently hard-coded along with their IDs for our convenience. Below these is a form.

```
<h2>Movies Database</h2>

<ul>
  <li>
    <h3>The Brave Little Toaster</h3>
    <div>ID: 1</div>
  </li>
  <li>
    <h3>The Princess Bride</h3>
    <div>ID: 2</div>
  </li>
  <li>
    <h3>Spirited Away</h3>
    <div>ID: 3</div>
  </li>
</ul>
```

```
<section>
  <form>
    <label for="searchByID">Search By ID</label>
    <input id="searchByID" type="text" placeholder="Enter ID here" />
    <input type="submit" />
  </form>
</section>
<section id="movieDetails">
  <h4>Title</h4>
  <p>Summary</p>
</section>
```

This form doesn't do much at the moment. In the browser, if we type something in and try to submit, our input just disappears.

Your primary task will be to get this form working. When a user inputs a valid ID, the movie information should appear on the page.

Doing this will involve a few steps:

- Add event listeners to capture form data and override a form's default behavior
- Fetch data based on what the user types into that form
- Display that data on the page

Add Event Listeners to Capture Form Data and Override the Form's Behavior

By default, HTML form elements will refresh when a **Submit** input is clicked. Before we can run the code for fetching data, we need to override this behavior.

In `./src/index.js`, we can do this by adding an event listener. Note that it already contains one event listener and a callback function, `init`:

```
const init = () => {};
```

```
document.addEventListener("DOMContentLoaded", init);
```

We want to make sure the JavaScript we write executes when the DOM is fully loaded. Any code related to DOM manipulation should either go in `init` or in a function called within `init`.

In our case, we want to add an event listener to the `form` element. We would first target the DOM element we want:

```
const inputForm = document.querySelector("form");
```

Then, we'll need to add an event listener to the form, currently represented by `inputForm` in our code.

[Event listeners](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) (https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) require two arguments: the *type* of event, a string, and the *listener*, a callback function. In our case, we'll want to pass in `'submit'` as the type. For the listener, we need to provide a callback function that will be called to 'handle' the event.

When the event is triggered, the callback function we've provided will execute and an object representing the event will be passed in as an argument. We can expect this to happen and can write a parameter in our code to store the event object in a variable:

```
inputForm.addEventListener("submit", (event) => {});
```

At this point, the form will still refresh automatically, as we haven't done anything to override that yet. The `event` object that gets passed in to our callback contains a particular method we need in order to override our form's behavior — `preventDefault()`.

```
const init = () => {
  const inputForm = document.querySelector("form");

  inputForm.addEventListener("submit", (event) => {
    event.preventDefault();
  });
};

document.addEventListener("DOMContentLoaded", init);
```

Calling this inside our callback will stop the page from refreshing and allow us to do something else instead. We can confirm everything is working by adding a `console.log` in our callback:

```
inputForm.addEventListener("submit", (event) => {
  event.preventDefault();
  console.log(event);
});
```

With dev tools open in the browser, if you enter some text and submit the form, you should see the `event` logged.

There is quite a lot stored on this `event` object, but we only need one thing: if we're fetching data based off a user input, we need to get the value of whatever the user entered; whatever you just entered into the form.

There are two ways we can get this value:

- The `event` object actually contains the value we need
- We can select the specific DOM element and get its value

Access Input Value from an Event Object

To get the value from our `event` object, we first want to access `event.target` [`event.target`](https://developer.mozilla.org/en-US/docs/Web/API/Event/target) returns the DOM element targeted by our event, a `<form>` in our case.

```
event.target;
// => <form>..</form>
```

`event.target` has a property, `children`, that returns an `HTMLCollection` [`HTMLCollection`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection) containing all the nested elements of the `event.target` element.

```
event.target.children;
// => HTMLCollection(3) [label, input#searchByID, input, searchByID: input#searchByID]
```

Looking at the form, we can see we want to access the *second* element:

```
<form>
  <label for="searchByID">Search By ID</label>
  <input id="searchByID" type="text" placeholder="Enter ID here" />
  <input type="submit" />
</form>
```

So we access this element via its index:

```
event.target.children[1];
// => <input id="searchByID" type="text" placeholder="Enter ID here">
```

And to get the input value, we use the `value` attribute

```
event.target.children[1].value;
// => whatever you typed into the input
```

Access Input Value Directly

We will always need to use `event.preventDefault()` to stop the page from refreshing. However, we don't necessarily need to use the `event` to get the value we need. We can also choose to access the `input` element directly.

```
inputForm.addEventListener("submit", (event) => {
  event.preventDefault();
  const input = document.querySelector("input#searchByID");

  console.log(input.value);
});
```

Both options work for getting the value we need. For now, we'll use the code above.

With this data, and the default form behavior overridden, we can set up a `fetch` request.

Fetch Data Based on User Input

Let's first set up the basic shell of our `fetch` request. To make sure everything is working and we can connect to the JSON server, we'll send a basic request to '`http://localhost:3000/movies`' :

```
const init = () => {
  const inputForm = document.querySelector("form");

  inputForm.addEventListener("submit", (event) => {
    event.preventDefault();
    const input = document.querySelector("input#searchByID");

    console.log(input.value);

    fetch("http://localhost:3000/movies")
      .then((response) => response.json())
      .then((data) => {
        console.log(data);
        // LOG: (3) [...], [...], [...]
      });
  });
};

document.addEventListener("DOMContentLoaded", init);
```

If everything is working, you should see an array of three objects logged in the console using the code above.

Note: For users of the [Live Server VSCode extension](https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer) ↗(<https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer>), if the page is reloading when you initiate the fetch request above, you'll need to set up some additional configuration for Live Server to play nicely with `json-server`. Follow the steps in [this gist](https://gist.github.com/ihollander/cc5f36c6447d15dea6a16f68d82aacf7) ↗(<https://gist.github.com/ihollander/cc5f36c6447d15dea6a16f68d82aacf7>) (you'll only need to do this once), then come back to this lesson.

These three objects represent the three 'records' available from the movies API. In our example, this is enough for us to move on — we have our user input accessible in `input.value`, and each object in `data` has an `id` property. We could now iterate over `data` and find a match between `input.value` and `id`.

However, it isn't usually the case that we want to get *all* records from an API or server. It would be helpful if we could have *the 'server'* do that work for us.

JSON Server follows RESTful conventions. As a result of these conventions, we can expect to be able to access specific records directly by providing the appropriate parameter in our request URL.

If you open a new tab in your browser and visit `http://localhost:3000/movies/1`, instead of seeing all three movie objects, you'll be presented with the object with `1` as its `id`:

```
{  
  "id": 1,  
  "title": "The Brave Little Toaster",  
  "summary": "A group of appliances set off on a journey"  
}
```

Similarly, if we pass this URL into our `fetch` request, we'll get this single object in return.

We need to modify the URL we pass to our `fetch` function based on the input typed into the HTML form. Using interpolation, we can adapt our existing code to do this:

```
inputForm.addEventListener("submit", (event) => {  
  event.preventDefault();  
  const input = document.querySelector("input#searchByID");  
  
  fetch(`http://localhost:3000/movies/${input.value}`)  
    .then((response) => response.json())  
    .then((data) => {  
      console.log(data);  
    })  
});
```

```
});  
});
```

Now, if you type a valid ID into the form, a specific movie object will be logged!

Note: What happens when you enter an *invalid* ID? In the console, you should see a `404` error. Something to think about as you continue to learn — what are some ways you might *handle* an invalid request?

Display Fetched Data on the Page

We've captured some user input and used it to customize a fetch request to our JSON server. The final step in our code-along is to display some of the retrieved data on the page. In the HTML, we have a `section` element with an id, `"movieDetails"`, that contains some filler content.

```
<section id="movieDetails">  
  <h4>Title</h4>  
  <p>Summary</p>  
</section>
```

Let's replace `Title` and `Summary` with data we retrieved from our server. To do this, we'll work inside the second `then` of our `fetch` request. First, we'll access the DOM and store the two elements in JavaScript

```
fetch(`http://localhost:3000/movies/${input.value}`)  
.then((response) => response.json())  
.then((data) => {  
  const title = document.querySelector("section#movieDetails h4");  
  const summary = document.querySelector("section#movieDetails p");  
});
```

Here again, we could access these elements in many ways, this is just one way to approach it. We could add `id` attributes to the `h4` and `p` tags directly.

Next, we want to change the contents of our `title` and `summary` elements based on the retrieved data. We can do this by setting their `innerText` values to the appropriate values in our data:

```
fetch(`http://localhost:3000/movies/${input.value}`)
  .then((response) => response.json())
  .then((data) => {
    const title = document.querySelector("section#movieDetails h4");
    const summary = document.querySelector("section#movieDetails p");

    title.innerText = data.title;
    summary.innerText = data.summary;
  });
}
```

All together, our code looks like this:

```
const init = () => {
  const inputForm = document.querySelector("form");

  inputForm.addEventListener("submit", (event) => {
    event.preventDefault();
    const input = document.querySelector("input#searchByID");

    fetch(`http://localhost:3000/movies/${input.value}`)
      .then((response) => response.json())
      .then((data) => {
        const title = document.querySelector("section#movieDetails h4");
        const summary = document.querySelector("section#movieDetails p");

        title.innerText = data.title;
        summary.innerText = data.summary;
      });
  });
};

};
```

```
document.addEventListener("DOMContentLoaded", init);
```

In the browser, if we type `1` into the form, we should see info on the **Brave Little Toaster**. Type `2`, and we get **The Princess Bride**. We're successfully fetching data *on demand!*

Submit Using CodeGrade

Once you're done, be sure to commit and push your code up to GitHub, then submit the assignment using CodeGrade. Even though this code-along does not have tests, it must still be submitted through CodeGrade in order to be marked as complete in Canvas.

Conclusion

A core aspect of the modern JavaScript-based web is that web pages can dynamically update their content as a user interacts with it. When a user adds a comment, adds an emoji response, etc., refreshing the page isn't a great experience.

In this lesson, we've gone through the basic mechanisms for providing a better experience. By capturing user input via event listeners, using `fetch` requests, and DOM manipulation, we can update page content as a user requests it. Although this won't be the case for all events, we also overrode HTML's default behavior.

Resources

- [`addEventListener\(\)`](https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener) ↗ (<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>)
- [`event.target`](https://developer.mozilla.org/en-US/docs/Web/API/Event/target) ↗ (<https://developer.mozilla.org/en-US/docs/Web/API/Event/target>)
- [`HTMLCollection`](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection) ↗ (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection>)