Fetching Data and Rendering SPAs

Course: Java

S1

Learning Outcomes

By the end of this lesson, you will be able to:

01 `fetch` JSON data.

02 Use `import` and `export`.

03 Use functions and template literals to generate markup.

04 Use a `render` function to add component markup to a page.

JavaScript and APIs

RESTful APIs

- As we previously learned, JSON is a universally embraced data format that virtually every language can consume.
- JSON can also be accessed/served by a JSON endpoint. For example, here is the `users` data that we saw in some previous activities.
- This an example of a RESTful API endpoint. It's a very simple one that only responds to `"GET"` requests.
- Most RESTful APIs provide endpoints that can handle `"GET"`, `"POST"`, `"PATCH"`, and/or `"DELETE"` requests. These can be served at the same endpoint or at different endpoints to manage different data.
- Most RESTful APIs provide endpoints that allow controlled access to a database. We can perform **Create, Read, Update, and Delete (CRUD)** operations solely through consuming given endpoints.
- Adequate documentation and examples are paramount to the effectiveness of any RESTful API.

Promises and fetch

- How do we consume endpoints and retrieve JSON data from our JS Code? One way that we might do so is via callbacks and the `XMLHttpRequest` object that is provided by the browser's Web API.

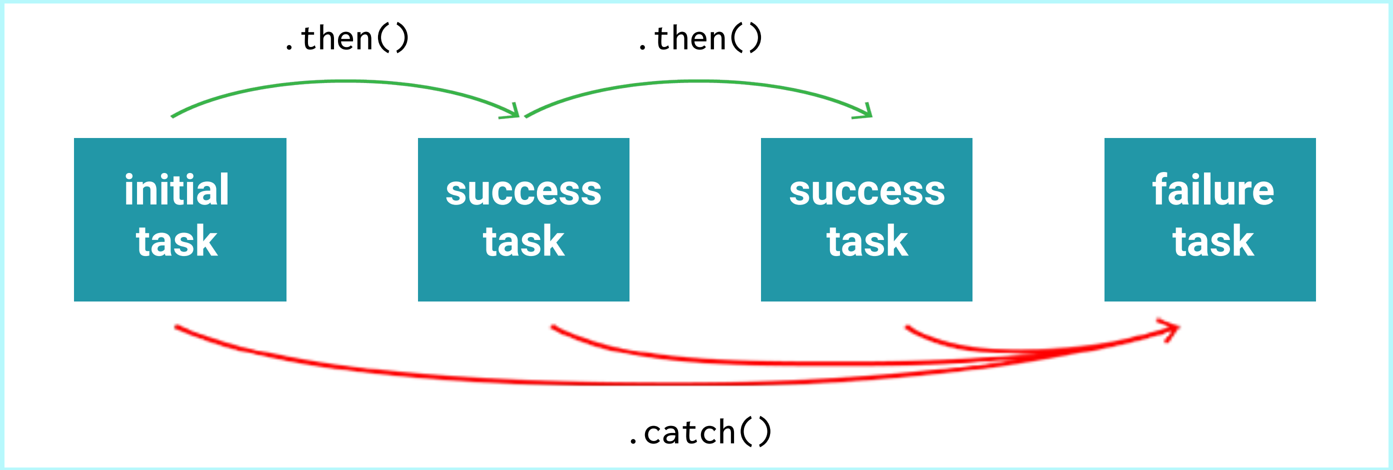
```
const req = new XMLHttpRequest();

// Listen for the data to be loaded with a callback
req.addEventListener("load", function() {
  console.log(this.responseText);
});
req.open("GET","https://jsonplaceholder.typicode.com/users");
 req.send();
```

- This approach might be fine for simple situations. However, over the years, this callback-based approach became a nested mess when dealing with complex situations and the loading of various assets. Each possible error situation required additional nested callbacks, resulting in Callback Hell.

Promises and fetch (continued)

- `fetch` is built around **promises**. Promises were introduced in ES6 as an alternative for situations such as data-fetching.
- **Promises don't replace callbacks** and are fundamentally different mechanisms from callbacks, down to how they are prioritized on the JS call stack. Callbacks are still used for event-driven asynchronicity, as we have seen. However, for I/O operations, promises are a much better fit.
- When an asynchronous function that returns a promise is invoked, we usually see the `then` keyword.
- The `then` keyword lets us know that this is indeed an asynchronous operation.
- Assuming that it resolves successfully, **then** call back another function.



Time to Code

fetch

Suggested Time:

10 Minutes

Time to Code

async - await

Suggested Time:

20 Minutes

fetch With Other HTTP Methods

```
const response = await fetch("https://jsonplaceholder.typicode.com/users", {
    // TODO: Change method as needed
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    // Convert JS into JSON to match `"Content-Type"`
    // TODO: Add/remove/update `body` with any necessary data
    body: JSON.stringify({
        name: "Mark West",
        email: "mark@west.com",
        phone: "800-993-8838",
      })
     ),
```

Note: The above endpoint is a mock. No data will actually be updated, written, or deleted.

Break

SPAs

Single Page Applications (SPAs)

- A SPA is a web application that uses JS to **render components** on a **single page**.
- These components can be written as **functional components**. That is, functions that essentially return some HTML code as a string:

```
const Header = () => `<h1>Hello!</h1>`
```

- Notice that these functional components are capitalized. This is because they represent nouns—the components—instead of verbs.
- These components will be modularized using EcmaScript modules with the keywords `import` and `export`. This is the modern version of CommonJS modules that use the `require` keyword.
- These components will change in response to changes to the **state** of our application.
- In this way, the SPA renders different **views**—that is, either different components, or the same components with different states.
- A simple example of some state in our app might be the current data that our app is using. For example, our app could start with no data; then, it might use `fetch` to asynchronously update `state` with newly retrieved data.
- Whenever `state` updates, we will trigger a **render function** that will update the components on the page in response.

Time to Code

Render a Functional Table from fetched Data

Suggested Time:

180 Minutes

RECAP

Questions?