Web Dev Basics 2

CS571: Building User Interfaces

Cole Nelson

Before Lecture

- Clone today's code to your machine.
- Download and install Postman!

Web Dev Basics 1

- The Web is made up of HTML, CSS, and JS!
  - **HTML:** structure
  - **CSS:** styling
  - **JS:** behavior
- CSS and JS can be applied to HTML inline, internal, or externally.

Web Dev Basics 1

Use `document` to reference the DOM.

```
let title = document.getElementById("articleTitle");
let loginBtn = document.getElementsByName("login")[0];
let callouts = document.getElementsByClassName("callout"); // *
```

*class refers to a **CSS** class

We can add *event listeners* or read/modify *properties*.

StackBlitz

Using these DOM elements, we can change the title of the article, add an action for when the button is clicked, and make all of the callouts red.

```
title.innerText = 'My Website!';
loginBtn.addEventListener("click", () => {
  alert("You are advancing to the next part of the site...");
});

for (let callout of callouts) {
  callout.style.color = "red";
}
```

StackBlitz

Finish ICE-1

Use *today's* starter code and (a) fix the broken reviews button and (b) implement the update yield function.

Learning Objectives

1. Manipulate the DOM via JavaScript.
2. Define a callback function.
3. Understand how asynchronous code executes.
4. Fetch, parse, and use JSON data from an API to populate webpage.

What is JSON?

Definition: JavaScript Object Notation (JSON) is a structured way to represent text-based data based on JS object syntax.

Refresher: JS Objects

Definition: Objects are unordered collection of related data of primitive or reference types defined using key-value pairs.

```
const instructor = {
  firstName: "Cole",
  lastName: "Nelson",
  roles: ["student", "faculty"]
}
```

JSON Equivalent

```
{
  "firstName": "Cole",
  "lastName": "Nelson",
  "roles": ["student", "faculty"]
}
```

What's the difference? A JS Object is executable code; JSON is a language-agnostic representation of an object. There are also slight differences in syntax.

You can write comments in JS Objects...

```
const drinks = [
  {
    name: "Mimosa",
    ingredients: [
      {name: "Orange Juice", hasAlcohol: false},
      {name: "Champagne", hasAlcohol: true}
    ]
  },
  {
    name: "Vesper Martini", // shaken, not stirred
    ingredients: [
      {name: "Gin", hasAlcohol: true},
      {name: "Vodka", hasAlcohol: true},
      {name: "Dry Vermouth", hasAlcohol: true},
    ]
  }
]
```

... but not in JSON!

```
[
  {
    "name": "Mimosa",
    "ingredients": [
      { "name": "Orange Juice", "hasAlcohol": false },
      { "name": "Champagne", "hasAlcohol": true }
    ]
  },
  {
    "name": "Vesper Martini",
    "ingredients": [
      { "name": "Gin", "hasAlcohol": true },
      { "name": "Vodka", "hasAlcohol": true },
      { "name": "Dry Vermouth", "hasAlcohol": true }
    ]
  }
]
```

Conversion

Because JS Objects and JSON are so similar, it is easy to convert between them.

- `JSON.parse` JSON String → JS Object
- `JSON.stringify` JS Object → JSON string

Conversion Examples

Using `JSON.parse` and `JSON.stringify`.

```
const myObj = JSON.parse('{"name": "Cole", "age": 26}');
const myStr = JSON.stringify(myObj);

console.log(typeof myObj); // object
console.log(typeof myStr); // string
```

❗ ❗ **Question:** Can I do `myObj.age = 27`? Yes!

Re-Visiting `const`

`const` means you cannot re-assign the variable. You can, however, re-assign one of its properties.

```
const cat = {name: "Pepper", age: 12}

cat.name = "Salt" // ok!
cat.age = 4; // ok

cat = {name: "Salt", age: 4} // not ok!
```

Yikes... this can be helpful, but also dangerous!

Deep Copying

We can make a deep copy using `JSON.parse` and `JSON.stringify` together*

```
let x = 1; // primitive! (stack)
const myObj = { "name": "Cole", "age": 26 }; // complex! (heap)
const myDeepCopy = JSON.parse(JSON.stringify(myObj));
myObj.name = 'Brad';
```

Interactive Example

*small caveat: does not copy functions of an object

Reference Copying

This is not a true copy! We call it a "reference" copy.

```
const myObj = { "name": "Cole", "age": 26 };
const myCopy = myObj;
myObj.name = 'Brad';
```

Why does this happen? *Objects* are stored on the heap; the variable's value is just a memory address!

Interactive Example

Why do I need to know this?

Web programming is **all about data**. Can I take this data an API and display it to a user?

- *Always* know the data that you are working with.
- *Be aware* of how assignments affect this data.

Let's start getting some data via an **API**!

What is an API?

Definition: An application programming interface (API) is a set of definitions and protocols for communication through the serialization and de-serialization of objects.

JSON is a language-agnostic medium that we can serialize to and de-serialize from!

How do we make an API request?

- Your browser!
- cURL
- Postman
- JavaScript

Try making an API request to...

- https://v2.jokeapi.dev/joke/Any?safe-mode
- https://cs571api.cs.wisc.edu/rest/su24/ice/chili

Your Turn!

Fetch from the Jokes and CS571 APIs using...

- Your browser!
- Postman

Request for JSON

- Requests can be `synchronous` or `asynchronous`.
- `asynchronous` requests are recommended as they are *non-blocking*. Typically, they use a *callback* when the data is received and lets the browser continue its work while the request is made.

More on synchronous/asynchronous requests

Making Asynchronous HTTP Requests

Two key methods: `XMLHttpRequest` (old) and `fetch` (new). `fetch` is a promise-based method.

- `Promise` objects represent the eventual completion/failure of an *asynchronous* operation and its resulting value.
- `async` / `await` — keywords to indicate that a function is *asynchronous* -- will learn later!

fetch()

```
fetch(url)
  .then((response) => response.json()) // ignore the headers, get the data
  .then((data) => {                    // implictly parses JSON to JS Object
    console.log("Data received!");
    console.log(data);
  })
  .catch(error => console.error(error)) // Print errors (if any)
```

Fetching Jokes

`fetch()`

Fetch happens *asynchronously*.

```
fetch(url)
  .then((response) => response.json())
  .then((data) => {
    console.log("I won't be printed 'til later!")
    console.log("Data takes time to fetch!")
  })
  .catch(error => console.error(error))

console.log("I will print first!")
```

StackBlitz

`fetch()` from a CS571 API

```
fetch(url, {
  method: "GET",
  headers: {
    "X-CS571-ID": "bid_xxxxxxxxxxx" // generally bad practice
  }
})
.then(response => response.json())
.then(data => {
  // Do something with the data
})
.catch(error => console.error(error)) // Print errors
```

There is a database that maps your BID to a WISC ID!

`fetch()` from a CS571 API

```
fetch(url, {
  method: "GET",
  headers: {
    "X-CS571-ID": CS571.getBadgerId() // better!
  }
})
.then(response => response.json())
.then(data => {
  // Do something with the data
})
.catch(error => console.error(error)) // Print errors
```

There is a database that maps your BID to a WISC ID!

Callback Functions

`then` and `catch` take a *callback function* as an argument.

Definition: A *callback function* (sometimes called a *function reference*) is passed into another function as an argument, which is then invoked inside the outer function to complete a routine or action.

More on callback functions

Callback Functions

Reminder: All of these define a function.

```
function fToC (temp) {
  return (temp - 32) * 5/9;
}
```

A *function definition*

```
const fToC = (temp) => {
  return (temp - 32) * 5/9;
}
```

An *arrow function*

```
const fToC = (temp) => (temp - 32) * 5/9
```

With an *implicit return*

Your Turn!

Let's fetch some recipes.

https://cs571api.cs.wisc.edu/rest/su24/ice/chili
https://cs571api.cs.wisc.edu/rest/su24/ice/pasta
https://cs571api.cs.wisc.edu/rest/su24/ice/pizza

Remember: You'll need a Badger ID to access these!

Badger IDs

Logging in to the CS571 APIs only allows you to see the data in your browser.

You will still need to send an `X-CS571-ID` header with each request. You can get your CS571 Badger ID with `CS571.getBadgerId()`, which grabs your Badger ID from `localStorage`, a concept we'll discuss later in the semester!

DOM Manipulation

Earlier, we learned how to get elements from the DOM and change their text.

```
let title = document.getElementById("articleTitle");
title.innerText = "My New Title!"
```

What if we want to *add* elements?

```
title.innerHTML = "<strong>My New Title!</strong>"
```

DOM Manipulation

We typically prefer to *not* use `innerHTML` when adding things to the DOM. Why?* Instead, we would...

```
const title = document.getElementById("articleTitle")
const newNode = document.createElement('strong')
newNode.innerText = 'My New Title!'
const newNode = title.appendChild(newNode);
```

* We could still safely clear the existing text with `title.innerHTML = ''`

Your Turn!

Let's display recipes to the page *dynamically*.

Questions?