Hyperiondev

TASK

JSON and AJAX With Fetch

Visit our website

Introduction

WELCOME TO THE JSON AND AJAX WITH FETCH TASK!

JavaScript objects and arrays are crucial data structures in web development and are frequently used when working with HTTP. This task introduces essential tools like the Web Storage API and JSON, which are important for handling data in web applications. You will also explore object-oriented programming, essential computer science data structures such as queues, trees, and stacks, and the concept of asynchronous programming. These foundations are key for mastering AJAX techniques with the Fetch API and understanding the deeper workings of JavaScript in web development.

RECAP OF OBJECT-ORIENTED PROGRAMMING: WHAT IS AN OBJECT?

Before we start working with JSON, you need to be comfortable with the concept of objects. So, what is an object? To answer this question, let's look at a typical object we might find on a web page: a button. By its very nature, a button should make something happen when you press it – i.e., it should perform an **action**. A button also needs to have **attributes**: it needs to be a certain size and shape, it needs a colour, and possibly a label or image on it to suggest what it might do when clicked. These characteristics are the foundation of any object you create in programming. An object needs to have **attributes** and it needs to have **methods** to enable it to perform a certain action. Have a look at the button below. What are its attributes? What action does it execute?

Simply put, object-oriented programming (OOP) is a way of creating neat packages of code (i.e., objects) that have certain attributes and can perform specific actions. This is a shift away from the procedural paradigm described in the next section.

PROCEDURAL PROGRAMMING VS OBJECT-ORIENTED PROGRAMMING

Procedural code follows a sequential flow, where each statement executes one after the other. The data is kept separate from the code that manipulates it, and functions are standalone modules that can access external variables. In OOP, data and the code that manipulates it are combined into objects. These objects communicate with each other through methods to perform tasks, integrating both data and functionality within the same module.

JAVASCRIPT OBJECTS

JavaScript is an object-based programming language, so an understanding of objects will enable you to understand JavaScript better. Almost everything in JavaScript is an object!

OOP makes your code more modular, leading to shorter development times and easier debugging by reusing proven, reliable code.

Take note:

JavaScript is not strictly an OOP language. It is an object-based scripting language. Like pure object-oriented languages, JavaScript works with objects, but with JavaScript, objects are created using prototypes instead of classes. A prototype is an object from which other objects can inherit properties. Every JavaScript object is linked to a prototype, which allows for property sharing and reuse.

ES6 update: ES6 allows us to create objects using keywords (like `class`, `super`, etc.) that are used in class-based languages, although, under the hood, ES6 still uses functions and prototypal inheritance to implement objects. See here for more information on **ES6 classes**.

We've covered the basic theory of what objects are and why we use them; now let's get coding and learn to create JavaScript objects.

CREATING DEFINED OBJECTS

There is more than one way of creating objects. We will consider two key ways in this task.

Method 1: Using an object literal (easiest)

Let's consider the following object declaration:

```
let car = {
  brand: "Porsche",
  model: "GT3",
  year: 2004,
  color: "White",
  howOld: function () {
    const currentYear = new Date().getFullYear();
    return currentYear - this.year;
  },
};
```

Here an object is declared with four different properties, which are then stored in a variable called `car`. This object also contains a method called `howOld`.

- **Object properties/attributes:** As you know, an object is comprised of a collection of named values called attributes. To demonstrate this concept, we'll consider the `car` object:

Property	Value
Brand	Porsche
Model	GT3
Year	2004
Colour	White

- **Object methods:** Methods (functions) are a series of actions that can be carried out on an object. An object can have a primitive value, a function, or even other objects as its properties. Therefore, an object method is simply a property that has a function as its value.

  Notice that the `car` object above contains a method called `howOld`. This object method calculates how old the car is by dynamically checking the current year using `getFullYear()` and subtracting the car's production year. This ensures that the age calculation is always up-to-date.

In the method, `this.year` is used instead of `car.year`. The `this` keyword refers to the current object within the method. It's used to access the `year` property of the object that owns the method, ensuring the calculation is based on the correct property.

This first method for creating objects is used to create a single object at a time. What if you want to create several objects that all have the same properties and methods but different values? The second method that you will consider provides an effective way of creating many objects using a single object constructor.

Method 2: Using an object constructor

The second way of creating an object is by using an object constructor. A **constructor** is a special type of function that is used to make or construct several different objects.

Consider the example below. The function `carDescription` is the constructor, and it's used to create three different car objects: `car1`, `car2`, and `car3`.

```
function carDescription(brand, model, year, color) {
   this.brand = brand;
   this.model = model;
   this.year = year;
   this.color = color;
};

let car1 = new carDescription("Porsche", "GT3", 2012, "White");
let car2 = new carDescription("Ford", "Fiesta", 2015, "Red");
let car3 = new carDescription("Opel", "Corsa", 2014, "White");
```

The dot or bracket notation can be used to modify any value in an object. For example, if you had your Porsche painted black, you could change the colour property of your car as shown below:

```
car1.color = "Black";
```

To delete properties of an object, you use the **`delete`** keyword:

```
delete car2.color;
```

Take note:

In JavaScript, **`this`** is used to refer to the object itself. So, in the example above, **`this`** is used to refer to **`carDescription`** to identify the brand, model, year, and colour of each new object. This is what enables us to say **`car1.brand = "Porsche"`** – because **`this.brand`** shows that we are referring to this specified object.

SOME OBJECTS THAT YOU HAVE ALREADY ENCOUNTERED

As noted previously, JavaScript is an object-based language. Most of the built-in code that you have worked with so far has, therefore, used objects. For example, whenever a web page is loaded, an object called **`document`** is created. This object contains methods and properties that describe and can be used to manipulate the web page's structure and content.

You have been using the **`document`** object when you use code such as the following:

```
let htmlSelect = document.getElementById('personList');
```

Similarly, every time you create an array, you are creating an array object that is defined by an array class. The whole JavaScript programming language is built using objects.

COMBINING FUNCTIONS AND OBJECTS

Now that you have a good understanding of functions and objects, let's consider the declaration of an object with implementation through a function (after all, you do want the object to serve a purpose):

```
// Creating an empty object called “loaded”
let loaded = {};

// Adding a method (a function stored inside an object) to the “loaded” object
// The method is called “testing” and it takes one parameter called “signal”
loaded.testing = function (signal) {
  // Display the output with "Hello World!" and the value of “signal”
  console.log("Hello World! " + signal);

  // Save the “signal” value in the object for later use
  loaded.signal = signal;
};

// Calling the “testing” method with the text "This page has loaded!"
// This will show the output and save the text inside the object
loaded.testing("This page has loaded!");

// Displaying the value of “signal” saved in the object
console.log(loaded.signal);
```

In the above example, an empty object is created with `let loaded = {};`. In JavaScript, an object can be thought of as a container that can be used to store information or functions that perform specific tasks. Currently, the object is empty, but within the example a function will be added to the object.

After the object is created, a method, which is a function stored within the object, is assigned to `loaded.testing`. In this case, the method takes a single input, called `signal`. When the method is called, it will handle two actions. Firstly, it displays a message to the user. The method starts with “Hello World!” and is followed by the value passed as the `signal`. When the method is called with `loaded.testing("This page has loaded!");`, it will display the following message, `"Hello World! This page has loaded!"`.

The second action, which the method handles, is to store the value of `signal` inside the object so that it can be used later. After running `loaded.testing("This page has loaded!");`, the object stores the text `"This page has loaded!"`. As a result, the

object not only displays a message but also keeps track of the text that was passed to the method.

This provides the object with the ability to both perform an action such as displaying a message and to store information such as text.

SENDING OBJECTS BETWEEN A WEB SERVER AND CLIENTS

HTTP (HyperText Transfer Protocol) is the protocol that allows for information to be transferred across the Internet. Everything that is transferred over the web is transferred using HTTP. There are two very important facts about HTTP that we need to keep in mind, though:

1. **HTTP is a stateless protocol.** This means HTTP doesn't see any link between two requests being successively carried out on the same connection. Cookies and the Web Storage API are used to store necessary state information.
2. **HTTP transfers text (not objects or other complex data structures).** To send objects or other structured data, we need to turn them into text so they can be transferred using HTTP. Later, we can convert that text back into the original data structure once it's received.

THE WEB STORAGE API: SESSION STORAGE

Thus far, we have used variables to store data used in our programs. When storing data used for web applications, it's important to keep in mind that HTTP is a **stateless protocol**, meaning that the web server doesn't store information about each user's interaction with the website. For example, if 100 people are shopping online, the web server that hosts the online shopping application doesn't necessarily store the state of each person's shopping experience (e.g., how many items each person has added or removed from their shopping cart).

Instead, to store this kind of data for each user, we rely on the browser. One way to store information in the browser is by using **cookies**, but a more modern and efficient solution is the **Web Storage API**. The Web Storage API allows us to save data in the browser using key-value pairs (essentially, pairs of labels and values). This method has largely replaced the use of cookies for many purposes.

The Web Storage API provides two ways of storing data:

1. `sessionStorage`: This only stores data for the current session. In other words, the data only remains available for as long as the browser is open. Once the browser has been closed, the data that was stored within the session storage is cleared.

2. **`localStorage`**: This stores data across sessions. Even when the browser has been closed and reopened, the data stored within the local storage remains available until it has been specifically removed.

Example of **`sessionStorage`**:

Imagine a scenario where you're creating an online store and want to track how many items the user has added to their cart while they're browsing. This can be done using **`sessionStorage`** to keep track of this information as long as the browser remains open.

Here is an example of how we can add the number of items in the user's cart using **`sessionStorage`**:

```
sessionStorage.setItem("cartItemCount", 4);
```

This adds a key-value pair to **`sessionStorage`**. The key is **`"cartItemCount"`**, and the value is **`4`**. This means that the browser will remember that the user currently has four items in their shopping cart during this browsing session.

To retrieve the value later, if we wished to display the number of items in the user's cart, we can do the following:

```
let itemCount = parseInt(sessionStorage.getItem("cartItemCount"));
```

This would retrieve the value associated with the key **`"cartItemCount"`** and convert it to a number so it can be used more easily. Now, **`itemCount`** will be **`4`**, and you can display this to the user. However, if the user closes the browser, the **`sessionStorage`** will be cleared and the cart item count will no longer be stored.

Example of **`localStorage`**:

Now, let's imagine that you would like to remember the items that the user has added to their cart, even after they leave the website and close their browser. This can be helpful for when they return to the website. For this, **`localStorage`** can be used to store this information across sessions.

Below is an example of how cart items can be stored within **`localStorage`**:

```
localStorage.setItem(
  "cartItems",
  JSON.stringify(["item1", "item2", "item3", "item4"])
);
```

This would add a key-value pair to `localStorage`. The key, in this case, is `"cartItems"`, and the value is a string representation of an array with the items `["item1", "item2", "item3", "item4"]`. Because `localStorage` can only store text, `JSON.stringify` is used to convert the array into a string.

When the user returns to the website at a later stage, the cart items can be retrieved if the following code is used:

```
let storedCart = JSON.parse(localStorage.getItem("cartItems"));
```

This retrieves the value stored under the key `"cartItems"` and converts it back into an array using `JSON.parse`. Now, the variable `storedCart` will be `["item1", "item2", "item3", "item4"]`, which allows you to show the user their saved cart.

Compared to `sessionStorage`, the data that has been stored within `localStorage` will persist even when the browser is closed. This way the user's cart will still be there when they return to the website.

For more information about how to use `sessionStorage`, see the files **personObjectEG2.js** and **personObjectEG.html**, or read up on it further on **this web page**.

JAVASCRIPT OBJECT NOTATION

As stated previously, everything that is transferred over the web is transferred using HTTP. As the name suggests, this protocol can transfer **text**. All data that is transferred across the web is, therefore, transferred as text. As such, we cannot transfer JavaScript objects between a web server and a client. XML and JSON are commonly used to convert JavaScript objects into a format that can be transferred with HTTP.

What is XML?

eXtensible Markup Language (XML) is a markup language used to annotate text or add additional information. Tags are used to annotate data, and these tags are not shown to the end-user but are needed by the "machine" to read and subsequently process the text correctly.

Below is an example of XML. Notice the tags: They are placed on the left and the right of the data you want to markup, and they wrap around the data.

```
<book id="bk101">
  <author>Gambardella, Matthew</author>
  <title>XML Developer's Guide</title>
  <genre>Computer</genre>
  <price>44.95</price>
  <publish_date>2000-10-01</publish_date>
  <description>An in-depth look at creating applications
  with XML.</description>
</book>
```

In the example `<book>` is an opening tag, and `</book>` is a closing tag. The closing tag has a forward slash (`/`) to indicate the end of that element. Tags can also contain attributes, such as `id="bk101"`, which add extra information to the element.

This is the general pattern that we have to follow for all tags in XML:

```
<opening tag>Some text here.</closing tag>
```

Looking at the example of XML above may remind you of HTML. They are both markup languages, but whereas HTML focuses on **displaying data**, XML just **carries data**. XML files don't do anything except carry information wrapped in tags. We need software to read and display this data in a meaningful way. XML is used to structure, store, and transport data independent of hardware and/or software.

What is JSON?

JSON (JavaScript Object Notation) is a lightweight, language-independent data format used to exchange data between a web server and a client. Although derived from JavaScript syntax, JSON is not limited to JavaScript and can be easily used with many programming languages. JSON is designed to be both human-readable and easy for machines to parse, making it ideal for data storage and transfer in web applications.

Key features of JSON:

- **Lightweight:** JSON is compact and easier to read and write compared to alternatives like XML.
- **Language independent:** Although derived from JavaScript, JSON can be used with many programming languages.
- **Easy to parse:** Most programming languages have built-in support or libraries for converting JSON into native data structures, such as objects or arrays.

JSON's syntax

JSON'S syntax is similar to JavaScript object notation, but it has its own rules and conventions:

- **Data in key-value pairs:** JSON organises data into key-value pairs, where keys are strings and values can be strings, numbers, objects, arrays, booleans, or null.
- **Property names and values:** Keys must be enclosed in double quotation marks (**"**). Values can be of various types like strings, numbers, objects, arrays, booleans, or null.
- **Objects and arrays:** JSON objects are enclosed in curly braces **{ }**, while arrays are enclosed in square brackets **[ ]**.

Example:

```
{
  "name": "Jason",
  "age": 30,
  "isStudent": false,
  "courses": ["JavaScript", "Python", "React"]
}
```

JSON uses a syntax that looks similar to JavaScript objects, but it's purely a data format. Unlike JavaScript objects, JSON cannot contain functions or undefined values, and its keys must be in double quotation marks. Here are a few key differences between JSON and JavaScript objects:

- **Syntax:** In JSON, keys must be enclosed in double quotes, whereas JavaScript objects allow unquoted keys if they're valid identifiers.
- **Data limitations:** JSON can only store data types like strings, numbers, arrays, and other objects, while JavaScript objects can include functions.
- **Primary purpose:** JSON is used most frequently for data storage and transfer between systems, while JavaScript objects can include functions.

These differences make JSON an ideal format for transmitting data across different systems and languages, where data needs to be clearly defined and easily parsed. JSON files typically use the **.json** file extension, and the **MIME type** for JSON text is `application/json`.

Converting between JSON and JavaScript objects

One key advantage of JSON is its ability to convert JavaScript data types, like objects and arrays, into a text format that can be transferred easily using HTTP. When data is exchanged between a client (e.g., a web browser) and a server, it's often in the form of JSON strings so both the client and server can interpret it consistently.

In JavaScript, there are two main methods to work with JSON data: **JSON.parse()** and **JSON.stringify()**.

`JSON.parse()`

The `JSON.parse()` method is used to convert a JSON string into a JavaScript object. This is especially useful when you receive data from a web server. It's important to know that the web server sends JSON information as a string, not as a JavaScript object. This string must be parsed to be converted into a format that JavaScript can use directly.

For example, imagine that you receive the following JSON string from a web server:

```
'{ "name": "Jason", "age": 30, "city": "New York" }'
```

Here, this text is a JSON-formatted string. To convert this string into a JavaScript object that can be directly used in code, we can use the `JSON.parse()` function:

```
let person = JSON.parse('{ "name": "Jason", "age": 30, "city":
"New York" }');
```

After parsing, `person` is now a JavaScript object, allowing you to access its properties directly:

```
console.log(person.name); // Output: Jason
console.log(person.age);  // Output: 30
console.log(person.city); // Output: New York
```

```
// You can also perform operations on the object:
person.age += 1; // Increment Jason's age by 1
console.log(person.age);  // Output: 31
```

Note that JSON strings can come from various sources, such as a server API response or a local file. As long as the data is in JSON format, `JSON.parse()` can convert it into a JavaScript object, even if it's not from a **.json** file.

`JSON.stringify()`

When sending data from a client to a server, it needs to be in a format the server can easily read, which is usually a JSON string. JavaScript objects, however, aren't automatically in JSON format, so you need to convert them into a JSON string before sending them. `JSON.stringify()` does this conversion for you.

For example, let's say you have the following JavaScript object that you want to send to a server:

```
let person = { "name": "Jason", "age": 30, "city": "New York" };
```

Using `JSON.stringify()` converts this object into a JSON-formatted string:

```
let jsonString = JSON.stringify(person);
console.log(jsonString); // Output:
'{"name":"Jason","age":30,"city":"New York"}'
```

Now, `jsonString` is a string representation of the `person` object, formatted as JSON. This string can easily be sent to a web server using methods like `fetch()`, making it suitable for transmission over HTTP.

SOME BASIC COMPUTER SCIENCE DATA STRUCTURES

A data structure is an object that organises and stores data. You have already used various kinds of data structures, such as arrays and lists, in your code. There are many other data structures in the field of computer science. These include stacks, trees, queues, heaps, etc. Questions about data structures are often used in software developer job interviews. Understanding the basics of how the most common data structures work helps prepare you for technical interviews and for starting a career as a developer.

An in-depth discussion of all of these data structures is beyond the scope of this bootcamp. However, we will briefly discuss the basics of three important data structures used in JavaScript: queues, stacks, and trees.

Queues

A queue is a concept every one of us is familiar with. We have all had to wait in a queue at a bank, bathroom, or checkout line at the grocery store. Queues are also often encountered in IT, e.g., print queues. In computer science, a queue is a simple data structure where data has to enter the queue at the back and leave the queue from the front. It is therefore known as a first-in, first-out (FIFO) data structure.

Queue terminology:

- **Enqueue:** An operation that adds an item to the back of a queue.
- **Dequeue:** An operation that removes an item from the front of a queue.

Stacks

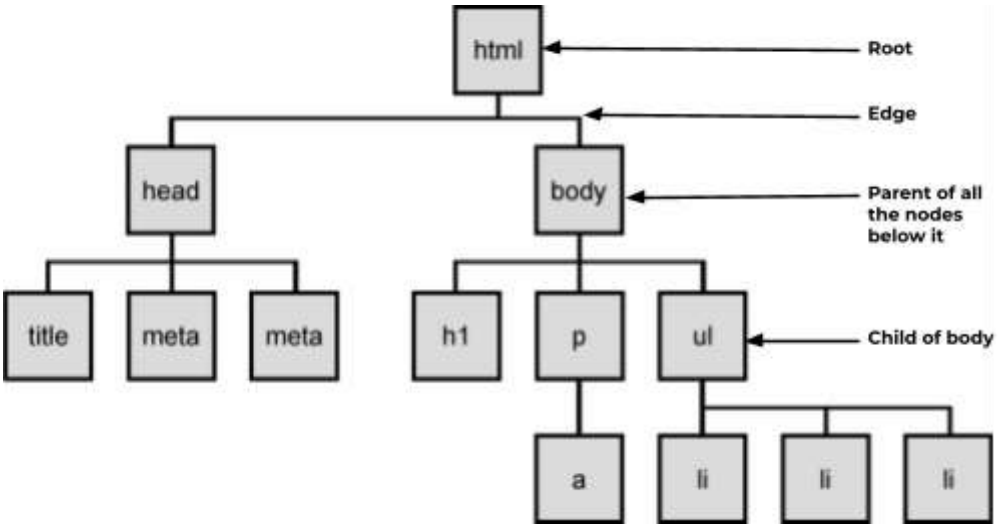
A stack is a data structure in which items are added to the top of the stack and removed from the top of the stack. It is therefore known as a last-in, first-out (LIFO) data structure.

Stack terminology:

- **Push:** An operation that adds an item to the top of a stack.
- **Pop:** An operation that removes an item from the top of a stack.

Trees

A tree is a data structure that stores data in a hierarchy made up of a collection of nodes connected by edges. Nodes are shown in the image below as squares (e.g., “html”, “head”, “body”, etc.) and edges are the lines that connect them. The HTML DOM (Document Object Model) is an example of an implementation of a tree data structure.



Tree data structures (DMS110 Programming for Digital Art, n.d.)

RECAP ON ASYNCHRONOUS PROGRAMMING

In coding, instructions usually run in sequence, one after the other, known as **synchronous execution**. Here, each line waits for the previous one to finish.

In **asynchronous execution**, however, code doesn’t wait, so tasks can complete out of order. This is useful in web development, allowing long tasks to run in the background without blocking others. For instance, while writing data to a database, the server can keep processing other tasks. Some tasks, though, depend on others to finish first, so JavaScript offers tools to manage asynchronous code in the correct order.

To get JavaScript to work asynchronously but also ensure that tasks are executed in the correct order, you can use some of the following tools:

1. **Callback functions:** A callback is a function that is passed as an argument to another function. The callback isn’t executed right away and is executed later somewhere within the calling function. The following code shows an example of the use of callback functions:

```
function a(s, callback) {
  callback(s);
}

function b(s) {
  console.log(s + "! It's me!");
}

// Call function “a” with a predefined callback “b”
a("Hello world", b); // Outputs: Hello world! It's me!

// Call function “a” with a custom callback
a("Hello world", function(s) {
  console.log(s + ", I can use callbacks");
  // Outputs: Hello world, I can use callbacks
});
```

An example of a very useful method that requires a callback function to work is the `map()` function. The `map()` function creates a new array with the results of calling a provided function on every element in this array. See an example of the `map()` function from **this web page** below:

```
// Create an array
let numbers = [1, 4, 9, 16];

// Call map(), passing a function
let mapped = numbers.map(function(x) { return x * 2 });

// Log the result
console.log(mapped);

// Output would be: 2,8,18,32
```

2. **Timing events:** With JavaScript, you can make some functions of code wait for a certain amount of time before being executed. You can do this by using timing events like:
    a. `setTimeout`:

    ```
    setTimeout(aFunction, 3000);
    ```

    This code would wait 3000 milliseconds before executing the function called `aFunction`. Notice that `aFunction` is an example of a callback function since it’s a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

b. `setInterval`:

```
setInterval(aFunction, 3000);
```

This code would repeatedly execute `aFunction` every 3000 milliseconds.

Take note:

Recap on arrow functions

Arrow functions offer an alternative way to declare functions in JavaScript. For example, an arrow function can be used as an anonymous callback in `setInterval()`. Anonymous functions, without a name, are typically inaccessible after creation. Arrow functions also have two key advantages over traditional functions:

- You use less code to do the same thing.
- There is no separate `this` property, and `this` has the same value as the enclosing environment. Arrow functions are essentially closures.

Without arrow functions:	**With** arrow functions:

`function Person() {`
`  let thisPerson = this;`
`  this.age = 0;`
 	
`  setInterval(function () {`
`  thisPerson.age++;`
`  console.log(thisPerson);`
`  }, 1000);`
`}`
`let p = new Person();`
	`function Person(){`
	`  this.age = 0;`
	
	`  setInterval(() => {`
	`  this.age++;`
	`  console.log(this);`
	`  }, 1000);`
	`}`
	`let p = new Person();`

The basic syntax for an arrow function is shown below. In the example above, there are no parameters for the arrow function, therefore, empty parentheses are used.

```
(param1, param2, ..., paramN) => { statements }
```

To revamp your code using arrow functions, see **this web page**.

3. **Promises:** A promise allows the interpreter to carry on with other tasks while a function is executed without waiting for that function to finish. This is because the promise basically "promises" to let you know the outcome of the function once it is finished. The basic structure for consuming a promise is:

```
doSomethingThatReturnsAPromise().then(successCallback,
failureCallback);
```

You consume the promise using `.then()` which accepts two optional arguments: a callback function for successful execution and a callback function for failure.

Fetch API

An example of an API that returns a promise is the **Fetch API**. As the name suggests, the Fetch API provides an interface for asynchronously fetching resources.

The Fetch API was introduced as a replacement for the **XMLHttpRequest API** when using the AJAX technique. AJAX in contemporary web development refers to the family of techniques employed to asynchronously fetch data and update the view that the users interact with based on the fetched data. It no longer refers to the specific use of **XML** or the XmlHttpRequest API. There are alternatives to the Fetch API such as **axios**, however, for this task, we will be working with the Fetch API because it's already **standardised** and built into your browser.

The `fetch()` method takes one mandatory argument: the path to the resource you want to fetch. It returns a **promise** that resolves to the **response** of that request, whether it is successful or not. An example of code using the `fetch()` method is shown below:

```
fetch("https://www.example_API.com/")
  .then((response) => response.json())
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error("There was an error with the request:", error);
  });
```

In the example above, the `fetch()` method makes a request to fetch data from the resource at the URL "https://www.example_API.com/". It returns a promise that resolves to the response of the request. The first `.then()` method takes this response and calls `response.json()`, which parses the response body as JSON. This parsing process also returns a promise. In the second `.then()`, the parsed data (now in a usable format) is logged to the console. If any errors occur during the `fetch` or parsing process, the `.catch()` method will handle them by logging the error message, ensuring smooth error management.

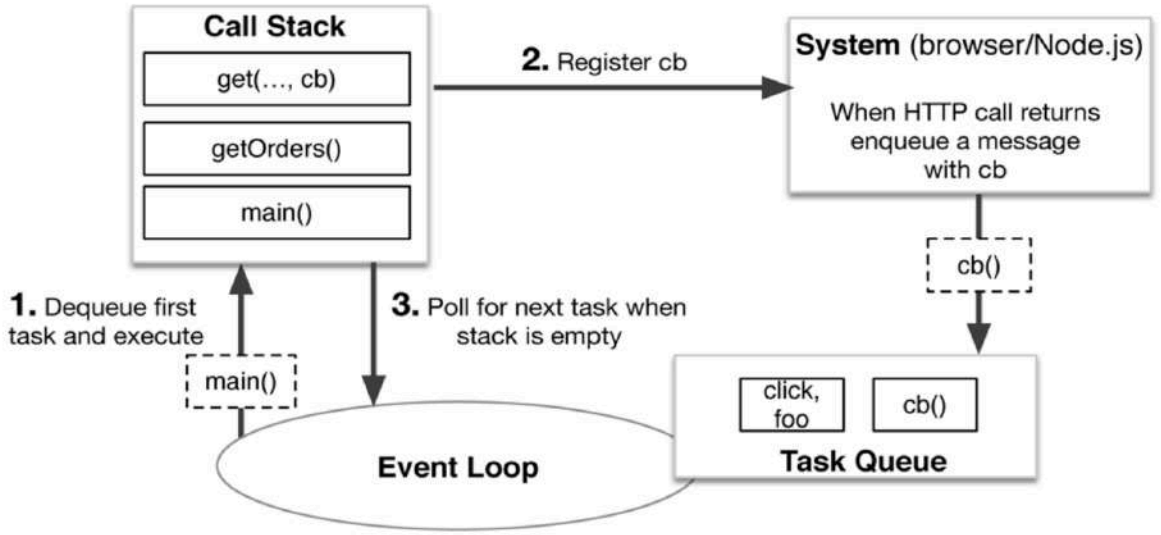
4. **Async functions:** More recent revisions of JavaScript have made it easier to consume promises using async functions (sometimes called async/await functions). An async function can contain an `await` expression that pauses the execution of the async function, waits for the passed promise's resolution, and then resumes the async function's execution, returning the resolved value. Notice that the code in the example below does exactly the same as the previous example that consumes the promise returned by the `fetch()` method but uses an async function instead:

```
const request = async () => {
    const response = await fetch("https://www.example_API.com/");
    const json = await response.json();
    console.log(json);
}
request();
```

HOW DOES JAVASCRIPT WORK?

JavaScript is a hosted language in that it never runs on its own; rather, it runs in a container. The container can be the browser (on the client side) or in Node.js (on the server side). As seen in the previous task, JavaScript is executed on the client side by a JavaScript engine. **Google's V8 engine** is a high-performance JavaScript engine that "compiles and executes JavaScript source code, handles memory allocation for objects, and **garbage collects** objects it no longer needs." On the server side, Node.js is the runtime environment that executes the JavaScript source code.

JavaScript uses an event-driven model with a single thread of execution. JavaScript's event loop is illustrated below. Notice how **stacks** (the call stack) and **queues** (the task queue) are involved with the event loop. Keep the theory you have read about these data structures in mind as we analyse the event loop.



JavaScript's event loop (Gallaba, Mesbah, & Beschastnikh, 2015)

1. When your code is compiled, functions are placed on the call stack and executed one at a time because JavaScript is single-threaded. If a function is slow, it can block others from running. To avoid this, callbacks are used as **non-blocking functions** to keep the system responsive.

2. When a function with a callback is encountered, the callback is sent to an API (web APIs on the client side and **C++** APIs with Node.js on the server side). Since the API is dealing with the callback function, the interpreter no longer has to worry about it and it's removed from the call stack. This means that the other functions on the call stack can continue being executed by the interpreter. Once the API is finished processing the callback function it pushes the callback onto the **task queue**.

3. The **event loop** sits between the call stack and the task queue. When the call stack is empty, the event loop will dequeue a function from the task queue onto the call stack where it will be executed. Therefore, callbacks on the task queue are put onto the call stack once all the other functions have been removed from the call stack.

The above should make it clear that, although JavaScript is single-threaded, it's fast because it allows functions to be executed concurrently (at the same time) by letting callbacks be handled by APIs, which act like multiple threads.

Use the **Loupe tool** and watch the accompanying video to make sure that you can visualise exactly how JavaScript's event loop works. The concepts discussed here are some of the most important but hardest to understand in this bootcamp. You

will be using these concepts more and more as you work with the MERN stack in the following tasks, and it's vital that you understand how JavaScript supports asynchronous, concurrent programming.

Instructions

Read through and run all the example files in the folder that accompanies this task. Getting to grips with JavaScript takes practice, and you will make mistakes in this task. This is to be expected as you learn the keywords and syntax rules of this programming language.

Practical task 1

Follow these steps:

- Create a basic HTML file. You are required to create a budgeting website with the following specifications.
- Make use of session storage to store the values.
- Create an `income` object where you can put in the following information as attributes:
    - Name, as a string (e.g., `Salary`).
    - Amount, as a number (e.g., 4000).
    - Whether or not it's recurring, as a boolean.
- Create five different objects to represent income from different places.
- Create an `expenses` object where you can put in the following information as attributes:
    - Name, as a string (e.g., `Groceries`).
    - Amount, as a number (e.g., 350).
    - Whether or not it's recurring, as a boolean.
- Create five different objects to represent different expenses.
- Using a prompt box, display the income items and let the user add another entry.
- Using a prompt box, display the expense items and let the user add another entry.

- Display the total amount of disposable income (income minus expenses).
- Using a prompt box, ask the user how much of their disposable income they would like to put into savings.
- Finally, create an alert to display the total amount of disposable income left.

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

Practical task 2

You will be working on three distinct programs, each placed in different folders. Follow these steps:

- **AJAX fetch:** Copy the Example 2 folder from the **AJAX With Fetch** directory to your local computer. Open the **fetch-example.js** file in your editor and complete all steps outlined in the comments. Then, copy the updated Example 2 folder back to your task folder.
- **Weather web page:** Create a web page that will display the current weather for Paris or a location of your choosing. Use the **Open-Meteo API**.
    - Note your app replaces the use of **cURL** shown in the example code block on Open-Meteo.
- **Real-time clock web page:** Create a web page that always displays the current time. The time should be updated every second. *Hint: use* `setInterval` *for this functionality.*

Be sure to place files for submission inside your task folder and click “Request review” on your dashboard.

Rate us

Share your thoughts

HyperionDev strives to provide internationally excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved, or think we've done a good job?

Click here to share your thoughts anonymously.

Reference list

DMS110 Programming for Digital Art. (n.d.) WordPress.
https://programmingfordigitalart.wordpress.com/2015/03/05/w6-html-and-js/

Gallaba, K., Mesbah, A, & Beschastnikh, I. (2015). Don't call us, we'll call you: Characterizing callbacks in JavaScript. *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 1–10. DOI: 10.1109/ESEM.2015.7321196