Web development basics

Data fetching

Introduction to the fetch() API

fetch() is a **JavaScript API** that provides an **interface** for **fetching** resources (including across the network). It's purpose is to retrieve and handle data from an API or any other web resource. The fetch() API uses **Promises** and can be used in modern **browsers** and in **Node.js** environments.

With fetch(), you can make **HTTP requests**, such as **GET** and **POST**, to retrieve or send data to a remote resource. The API returns a Promise that resolves to the **Response object** representing the response to your request. This response object can then be used to access the data returned by the API, typically in **JSON** format.

Making a Basic fetch() Request (Promise chain)

In this example, the **fetch**() function is called with the API endpoint **URL** as its argument. The function returns a Promise that resolves to the Response object. The .**then**() method is used to extract the **JSON** data from the response and log it to the console. The .**catch**() method is used to handle any errors that may occur during the fetch operation.

Note: It's important to check the ok property of the Response object to ensure that the API returned a successful response before attempting to extract the data.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => {
    console.error('There was a problem with the fetch operation:', error);
  });
```

Handling Response Data

The Response object can contain different types of data, such as **JSON**, **text**, **blob**, and others. To handle different types of data, you can use methods like .**json**(), .**text**(), and .**blob**() to extract the data in the format you need.

In this example, the .**text**() method is called to extract the response **data as text**. The text data can then be used in your application.

It's important to understand the format of the data returned by the API and use the appropriate method to extract and handle the data.

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.text();
  })
  .then(text => {
    console.log(text);
  })
  .catch(error => {
    console.error('There was a problem with the fetch operation:', error);
  });
```

Using fetch() with Async/Await

In this example, the getData() function is declared as **async**, which allows the use of the **await** keyword inside the function. The fetch() function is called and the result is assigned to the response variable. **The await keyword is used to wait for the response to be received before proceeding with the rest of the code.**

Using **async/await** to handle the fetch() response is a cleaner and easier to understand approach compared to using **Promises**. It makes the code more readable and reduces the amount of code needed to handle the response. The code is also easier to understand, especially for developers who are new to JavaScript or Promises.

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('There was a problem with the fetch operation:', error);
  }
}

getData();
```

Using fetch() with Async/Await

In this example, the getData() function is declared as **async**, which allows the use of the **await** keyword inside the function. The fetch() function is called and the result is assigned to the response variable. **The await keyword is used to wait for the response to be received before proceeding with the rest of the code.**

Using **async/await** to handle the fetch() response is a cleaner and easier to understand approach compared to using **Promises**. It makes the code more readable and reduces the amount of code needed to handle the response. The code is also easier to understand, especially for developers who are new to JavaScript or Promises.

```
async function getData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('There was a problem with the fetch operation:', error);
  }
}

getData();
```

Fetch methods

Headers

In the context of HTTP requests and responses, headers define the operating parameters of the HTTP transaction. They carry information about the request or the response, or about the object sent in the message body. HTTP headers are made up of a case-insensitive header field followed by a colon (:), then by its value (without line breaks). Multiple extra spaces before or after the colon are ignored. Here are some common HTTP headers:

1) **Content-Type**: This defines the media type of the resource, for example: "text/html", "application/json", etc.
2) **Content-Length:** This indicates the size of the entity-body in bytes, sent to the recipient.
3) **Accept**: This indicates the media types which are acceptable for the response from the server.
4) **Authorization**: This carries credentials containing the authentication information of the client for the request.
5) **User-Agent**: This contains a characteristic string that allows the network protocol peers to identify the application type, operating system, software vendor, or software version of the requesting software user agent.
6) **Host**: This specifies the domain name of the server (for virtual hosting), and (optionally) the TCP port number on which the server is listening.

Classwork for async/await

Task: Display the titles of the latest 10 articles from a mock news API using async/await and fetch()

Instructions:

1) Use the following API endpoint to retrieve the latest 10 articles: https://jsonplaceholder.typicode.com/posts
2) Create a function called getArticles that will make a GET request to the API endpoint.
3) Inside the getArticles function, use fetch() to make a GET request to the API endpoint and store the response in a variable called response.
4) Use the await keyword to wait for the response to be received before proceeding with the rest of the code.
5) Check if the response was successful (status code 200) by calling response.ok. If it's not successful, throw an error with a message "Network response was not ok".
6) Extract the JSON data from the response by calling response.json() and store the result in a variable called data.
7) Use a for loop to iterate over the data array and log the title property of each article to the console.
8) Call the getArticles function to start the process.

Classwork for async/await - solution

```
async function getArticles() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    for (let i = 0; i < 10; i++) {
      console.log(data[i].title);
    }
  } catch (error) {
    console.error('There was a problem with the fetch operation:', error);
  }
}

getArticles();
```

Browser storage

Criteria	Local Storage	Session Storage	Cookies
Storage Capacity	5-10 mb	5-10 mb	4 kb
Auto Expiry	No	Yes	Yes
Server Side Accessibility	No	No	Yes
Data Transfer HTTP Request	No	No	Yes
Data Persistence	Till manually deleted	Till browser tab is closed	As per expiry TTL set

Homework

Task 1: Continue previous homework. Find some free API, that you can integrate into you application OR use local storage to store session between refreshes

Optional Find any free API, create get/post/put/patch/delete requests with it.