Unidad de Trabajo 7. Parte III: Fetch

1. Introducción

Hasta ahora, has aprendido cómo manejar peticiones asíncronas por medio de dos métodos: el **objeto XMLHttpRequest** y por medio de **JQuery** (cuya finalidad es facilitar la programación pero sigue dependiendo de este mismo objeto). Sin embargo, con la introducción de **ECMAScript 6 (ES6)** para JavaScript, se incorporaron nuevas características que simplificaron considerablemente el uso del lenguaje en tareas que anteriormente impulsaron a muchos desarrolladores a depender de jQuery, especialmente en lo que respecta a las solicitudes HTTP. No obstante, a medida que el tiempo ha avanzado, las aplicaciones web modernas requieren herramientas más potentes. Es en este contexto que surge la relevancia de la **Fetch API**, una alternativa robusta que aborda las necesidades contemporáneas.

La Fetch API proporciona una interfaz moderna y flexible para realizar solicitudes HTTP, eliminando la necesidad de depender exclusivamente de jQuery para este propósito. Con su sintaxis más clara y la capacidad de devolver **Promesas**, la Fetch API simplifica la gestión de solicitudes asíncronas. En consecuencia, muchos desarrolladores encuentran que, al adoptar la Fetch API, es probable que prescindan de jQuery, ya que esta nueva herramienta proporciona una solución eficaz y nativa para las exigencias actuales de desarrollo web. En la siguiente página tienes la documentación de la API:

https://developer.mozilla.org/es/docs/Web/API/fetch

¿Qué son las promesas?

Las promesas, o **promise**, son un patrón de programación que se utiliza para manejar operaciones asíncronas de manera más efectiva. Antes de la introducción de promesas, el manejo de operaciones asíncronas se realizaba principalmente mediante **funciones callbacks**, lo que podía conducir a un código complicado y difícil de mantener. Las promesas proporcionan una sintaxis más clara y estructurada para gestionar **operaciones asíncronas**.

Una promesa representa un valor que puede estar disponible **ahora, en el futuro o nunca**. Esas promesas pueden estar en alguno de los estados siguientes:

- **pending (pendiente):** estado inicial, ni cumplido ni rechazado.
- **fulfilled (cumplida):** lo que significa que la operación se completó con éxito.
- **rejected (rechazada):** lo que significa que la operación falló.

Las promesas en JavaScript incluyen métodos como **then()** y **catch()**, que se encargan de manejar el resultado exitoso y los errores respectivamente. Además, las promesas pueden encadenarse utilizando **then()**, lo que facilita la creación de secuencias de operaciones

asíncronas más legibles, puesto que la respuesta dada por un then() se pasará al siguiente como parámetro. En resumen, las promesas son una herramienta esencial para trabajar con operaciones asíncronas de manera eficiente y estructurada en JavaScript. Puedes consultar su documentación aquí:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise

2. Fetch API. Sintaxis y manejo

El proceso de solicitud en Fetch resulta excepcionalmente claro:

a. Se inicia la solicitud.

b. Se obtiene una promesa que resuelve en un objeto **Response**.

c. El objeto Response se procesa a través de métodos específicos según el tipo de dato (json, blob, text, etc.).

¿Cuál es la sintaxis estándar de la solicitud?

La sintaxis básica para emplear Fetch es la siguiente:

```
fetch(url, [options])
  .then(function (response) {
    return response.text();
  })
  .then(function (data) {
    // Procesar los datos. Por ejemplo:
    console.log(data);
  })
  .catch(function (error) {
    // Manejar errores. Por ejemplo:
    console.error("Ocurrió un error:", error);
  });
```

El único parámetro obligatorio es la **url** de la petición. Además, se pueden pasar opciones en forma de objeto que veremos más adelante. A continuación, ,then() permite procesar la respuesta (en el ejemplo estamos asumiendo que la respuesta que vamos a recibir la procesaremos como texto, aunque existen otros formatos). A continuación, podemos encadenar tantos **then()** como deseemos para manipular esos datos. Fetch tiene la ventaja de **pasar como parámetro el resultado del primer then() al siguiente** siempre y cuando se añada la palabra reservada **return**. Finalmente, **catch()** permite manejar posibles errores en

el resultado de la petición. Sin embargo, esta petición se puede simplificar aún más teniendo en mente que las **arrow functions** tienen un **return implícito**. Quedaría entonces:

```
fetch(url, [options])
  .then(response => response.text())
  .then(data => console.log(data))
  .catch(error => console.error("Ocurrió un error:", error));
```

2.1. Opciones en la petición

Algunas de las opciones que se pueden configurar como parte de la petición son:

Opción	Descripción
method	Define el método HTTP de la solicitud (GET, POST, PUT, DELETE, etc.).
headers	Define los encabezados HTTP de la solicitud como un objeto clave-valor.
body	Contiene el cuerpo de la solicitud, generalmente para métodos POST.
mode	Define el modo de CORS (cors, no-cors, same-origin).
cache	Especifica cómo se deben manejar las cachés durante la solicitud.
credentials	Controla si se deben incluir credenciales en la solicitud (omit, include, same-origin).

Ten en cuenta que la lista completa puede variar según las especificaciones y versiones de ECMAScript. Puedes consultar la documentación oficial. Lo primero y más comúnmente realizado es especificar el **método HTTP** a utilizar en la solicitud. De manera predeterminada, se ejecutará un método GET, pero es posible cambiarlo a otros métodos según sea necesario, por ejemplo, POST para envío de datos. En segundo lugar, se pueden proporcionar **objetos para enviar en el cuerpo (body) de la solicitud**, y también es posible **modificar las cabeceras** mediante el campo headers. Por ejemplo:

```
const options = {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(jsonData)
};
```

En este ejemplo, estamos enviando una petición POST, indicando en la cabecera que se envía contenido JSON y en el cuerpo de los datos, enviando el objeto jsonData, codificado como texto mediante stringify().

Una forma fácil de generar las cabeceras en javaScript es mediante el **objeto Header**. En el ejemplo siguiente, creamos un objeto Headers, le añadimos un encabezado de autorización mediante el método append, y luego utilizamos dicho objeto al realizar una solicitud fetch:

```
// Crear un objeto Headers
const headers = new Headers();

// Agregar encabezado de autorización
headers.append('Authorization', 'Bearer tu-token-de-autorizacion');

// Realizar una solicitud fetch con las cabeceras personalizadas
fetch('https://api.ejemplo.com/data', {
 method: 'GET',
 headers: headers,
})
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error('Error:', error));
```

Este enfoque proporciona flexibilidad al personalizar las cabeceras de las solicitudes, siendo especialmente útil para incluir información como tokens de autorización, encabezados de contenido u otros datos específicos del cliente en las interacciones con el servidor.

Por otro lado, la opción **body** permite **enviar datos al servidor**. Este campo puede contener varios tipos de datos, como texto, JSON, FormData, entre otros. A continuación, se presenta un ejemplo de cómo utilizar la opción body para enviar datos en una solicitud fetch:

```
// Datos a enviar al servidor (puede ser un objeto, FormData, etc.)
const datosEnviar = {
 usuario: 'ejemplo',
 contraseña: 'secreta123'
};
```

```
// Realizar una solicitud fetch con la opción body
fetch('https://api.ejemplo.com/login', {
 method: 'POST',
 headers: {
  'Content-Type': 'application/json' // Especificar el tipo de contenido
 },
 body: JSON.stringify(datosEnviar) // Convertir los datos a formato JSON
})
 .then(response => response.json())
 .then(resultado => console.log(resultado))
 .catch(error => console.error('Error:', error));
```

2.2. Procesando la respuesta

La respuesta obtenida al realizar una solicitud fetch en JavaScript se encapsula en un objeto del tipo **Response**. Este objeto no solo contiene la información esencial de la respuesta del servidor, sino que también ofrece características valiosas. Además, la presencia de diversos métodos y propiedades en el objeto Response facilita el procesamiento y tratamiento eficiente de los diferentes tipos de respuestas, proporcionando así un conjunto versátil de herramientas para el manejo de datos en el entorno web.

Propiedades y Métodos del objeto Response

Propiedad/Método	Descripción
headers	Devuelve un objeto Headers que representa los encabezados de la respuesta.
ok	Indica si la solicitud fue exitosa (código de estado en el rango 200-299).
status	Devuelve el código de estado HTTP de la respuesta (por ejemplo, 200 para OK, 404 para No Encontrado, etc.).
statusText	Devuelve el texto del estado HTTP de la respuesta (por ejemplo, "OK" para 200, "Not Found" para 404, etc.).
url	Devuelve la URL del recurso solicitado.
clone()	Crea y devuelve un clon de la respuesta original.
json()	Devuelve una promesa que resuelve con los datos de la respuesta interpretados como JSON.

text()	Devuelve una promesa que resuelve con los datos de la respuesta como texto.
arrayBuffer()	Devuelve una promesa que resuelve con los datos de la respuesta como un ArrayBuffer.
blob()	Devuelve una promesa que resuelve con los datos de la respuesta como un objeto Blob.
formData()	Devuelve una promesa que resuelve con los datos de la respuesta como un objeto FormData.

Por ejemplo:

```
fetch(url)
 .then(response => {
  // Acceder a la propiedad status para obtener el código de estado
  console.log('Código de estado:', response.status);

  // Utilizar el método text() para obtener y mostrar el contenido como texto
  return response.text();
 })
 .then(textData => {
  // Trabajar con los datos obtenidos como texto
  console.log('Contenido de la respuesta (Texto):', textData);
 })
 .catch(error => {
  console.error('Ocurrió un error:', error);
 });
```

3. Bibliografía y Webgrafía

[1] “Fetch: Peticiones Asíncronas - Javascript en español”, Lenguajejs.com. [En línea]. Disponible en: https://lenguajejs.com/javascript/peticiones-http/fetch/ [Consultado: 19-ene-2024].

[2] “JSONPlaceholder - free fake REST API”, Typicode.com. [En línea]. Disponible en: https://jsonplaceholder.typicode.com/ [Consultado: 19-ene-2024].

[3] “fetch()”, MDN Web Docs. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/API/fetch [Consultado: 19-ene-2024].

[4] “Promise”, MDN Web Docs. [En línea]. Disponible en: https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise [Consultado: 19-ene-2024].

[5] “JavaScript Ya”, Tutorialesprogramacionya.com. [En línea]. Disponible en: https://www.tutorialesprogramacionya.com/javascriptya/index.php?inicio=100 [Consultado: 19-ene-2024].