Fetch

Realizar una petición simple

La API Fetch proporciona una interfaz JavaScript para gestionar peticiones y respuestas HTTP. La API Fetch es una interfaz moderna que le permite realizar solicitudes HTTP a servidores desde navegadores web.
Provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma **asíncrona** por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest (XHR). La API Fetch puede realizar todas las tareas como lo hace el objeto XHR.

Una petición básica de fetch es realmente simple de realizar. El método fetch() requiere como mínimo un parámetro con la URL del recurso que se solicita:
let response = fetch(url);
El método fetch() devuelve una Promesa por lo que que podremos utilizar then() y catch() para manejarla:

```
fetch(url)
  .then(response => {
    // handle the response
  })
  .catch(error => {
    // handle the error
  });
```

Cuando la petición se completa, el recurso está disponible. En ese momento, la Promesa resolverá el objeto Response.
El objeto Response es el “wrapper” de la API Fetch para el recurso obtenido. El objeto Response tiene un número de propiedades y métodos útiles para inspeccionar la respuesta.

Por ejemplo:

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo en la consola. El uso de fetch() más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto Promise conteniendo la respuesta, un objeto Response.

Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método json()

El ejemplo anterior

```
fetch('http://example.com/movies.json')
  .then(response => response.json())
  .then(data => console.log(data));
```

se podría haber escrito también de esta manera:

```
fetch('http://example.com/movies.json').then(response => response.json()).then(data => console.log(data));
```

También sería equivalente:

```
let funcion1 = response => response.json();
let funcion2 = data => console.log(data);

fetch('http://example.com/movies.json')
  .then(funcion1)
  .then(funcion2)
```

Y también:

```
function funcion1(response){
    return response.json();
}

function funcion2(data){
    console.log(data);
}

fetch('http://example.com/movies.json')
  .then(funcion1)
  .then(funcion2 )
```

Para solicitudes HTTP básicas, usar fetch() es un proceso de tres pasos:

1. Llamar a fetch(), pasando la URL cuyo contenido desea recuperar.
2. Obtenga el objeto de respuesta (Response), que devuelve de forma asíncrona el paso 1, cuando llegue la respuesta HTTP llamando a un método (then) de este Response para pedir el cuerpo de la respuesta.
3. Obtenga el objeto del cuerpo que se devuelve de forma asíncrona en el paso 2 y procésalo como quieras.

La API fetch() está completamente basada en Promise y hay dos pasos asincrónicos aquí, por lo que normalmente espera dos llamadas then() o dos expresiones await cuando se usa fetch().

Así es como se ve una solicitud fetch() si está usando then() y espera que la respuesta del servidor a su solicitud tenga formato JSON:

```
fetch("/api/users/current") // Make an HTTP (or HTTPS) GET request
.then(response => response.json()) // Parse its body as a JSON object
.then(currentUser => { // Then process that parsed object
        displayUserInfo(currentUser);
});
```

Aquí hay una solicitud similar realizada con las palabras clave async y await a una API que devuelve una cadena simple en lugar de un objeto JSON:

```
async function isServiceReady() {
    let response = await fetch("/api/service/status");
    let body = await response.text();
    return body === "ready";
}
```

Leyendo la Respuesta

Si el contenido de la respuesta está en formato de texto sin procesar, puede usar el método text(). El método text() devuelve una Promesa que se resuelve con el contenido completo del recurso obtenido:

```
fetch('/readme.txt')
  .then(response => response.text())
  .then(data => console.log(data));
```

En la práctica, a menudo se usa async/await con el método fetch() de esta manera:

```
async function fetchText() {
  let response = await fetch('/readme.txt');
  let data = await response.text();
  console.log(data);
}
```

Además del método text(), el objeto Response tiene otros métodos como json(), blob(), formData() y arrayBuffer() para manejar el tipo de datos respectivo.

Manejando los códigos de estados de la Respuesta

El objeto Response proporciona el código de estado y el texto de estado a través de las propiedades status y statusText. Cuando una solicitud es exitosa, el código de estado es 200 y el texto de estado es correcto:

```
async function fetchText() {
  let response = await fetch('/readme.txt');

  console.log(response.status); // 200
  console.log(response.statusText); // OK

  if (response.status === 200) {
    let data = await response.text();
    // handle data
  }
}

fetchText();
```

Output:

```
200
OK
```

Si el recurso solicitado no existe, el código de respuesta es 404:

```
let response = await fetch('/non-existence.txt');

console.log(response.status); // 400
console.log(response.statusText); // Not Found
```

Output:

```
400
Not Found
```

Si la URL solicitada arroja un error del servidor, el código de respuesta será 500.

Si la URL solicitada se redirige a la nueva con la respuesta 300-309, el estado del objeto Respuesta se establece en 200. Además, la propiedad redirigida se establece en verdadero.

La promesa devuelta por fetch() se resuelve en un objeto de respuesta (Response).
La propiedad **status** de este objeto es el código de estado HTTP, como 200 para solicitudes exitosas o 404 para respuestas "No encontrado" (statusText proporciona el texto estándar en inglés que acompaña al código de estado numérico.)

La propiedad **ok** de una Respuesta es verdadero si el estado es 200 o cualquier código entre 200 y 299 y es falso para cualquier otro código.

```
fetch("/api/users/current")      // Make an HTTP (or HTTPS) GET request.
.then(response => {              // When we get a response, first check it
if (response.ok &&               // for a success code and the expected type.
response.headers.get("Content-Type") ==="application/json") {
return response.json();                  // Return a Promise for the body.
} else {
throw new Error(                         // Or throw an error.
`Unexpected response status
${response.status} or content type`
);
}
})
.then(currentUser => {                   // When the response.json() Promise resolves
displayUserInfo(currentUser);            // do something withthe parsed body.
})
.catch(error => {                        // Or if anything went wrong,  just log the error.
                                 // If the user's browser is offline, fetch() itselfwill reject.
                                 // If the server returns a bad response then we throwan error
above.
console.log("Error while fetching current user:",
error);
});
```

Si un servidor web responde a su solicitud fetch(), entonces la Promesa que se devolvió se cumplirá con un objeto de respuesta, incluso si la respuesta del servidor fue un error “404 No encontrado” o “500 Internal Server Error”.
fetch() solo rechaza la Promesa que devuelve si no puede establecer conexión con el servidor web en absoluto.
Esto puede suceder si la computadora del usuario está fuera de línea,el servidor no responde, o la URL especifica un nombre de host que no no existe. Debido a que estas cosas pueden suceder en cualquier solicitud de red, es siempre es una buena idea incluir una cláusula .catch() cada vez que haga una llamada fetch().

Petición POST

Para realizar una solicitud POST, o una solicitud con otro método, debemos usar las opciones de fetch:

- method – HTTP-method, e.g. POST,
- body – the request body, one of:
  - a string (e.g. JSON-encoded),
  - FormData object, to submit the data as multipart/form-data,
  - Blob/BufferSource to send binary data,
  - URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

El formato JSON se usa la mayor parte del tiempo.

Por ejemplo, este código envía el objeto de usuario como JSON:

```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Tenga en cuenta que si el cuerpo de la solicitud es una cadena, el header Content-Type se establece en text/plain;charset=UTF-8 de forma predeterminada.

Pero, como vamos a enviar JSON, usamos la opción de cabeceras para enviar application/json en su lugar, el Content-Type correcto para los datos codificados en JSON.

Enviando datos JSON

Usa fetch() para enviar una petición POST con datos codificados en JSON .

```
var url = 'https://example.com/profile';
var data = {username: 'example'};

fetch(url, {
  method: 'POST', // or 'PUT'
  body: JSON.stringify(data), // data can be `string` or {object}!
  headers:{
    'Content-Type': 'application/json'
  }
}).then(res => res.json())
.catch(error => console.error('Error:', error))
.then(response => console.log('Success:', response));
```