

Desarrollo Web en Entorno Cliente

La API Fetch.

Contenidos

La API Fetch	3
¿Qué es la API Fetch?	3
Obtener una respuesta.	5
Propiedades de response.....	6
Métodos de response.	6
Manejo de errores.	7
Uso con async/await.	8
Métodos HTTP con fetch.....	9
Uso de headers con fetch.	10
Encabezados de respuesta.....	10
Encabezados de petición.....	11
Ejemplo petición XMLHttpRequest y petición fetch.	12
Ejemplos fetch con base de datos.	14
Ejemplo: Registro de un socio en la tabla socios.	14
Ejemplo: Consulta de un socio de la tabla socios.	15

La API Fetch

API Fetch es una interfaz moderna en JavaScript que permite realizar solicitudes HTTP de manera asíncrona, reemplazando el uso de *XMLHttpRequest*. Su sintaxis es simple y está basada en promesas, lo que la hace más legible y fácil de usar para manejar operaciones asíncronas en aplicaciones web. Fetch permite realizar peticiones *GET*, *POST*, *PUT*, *DELETE*, entre otras, y es útil para obtener o enviar datos a un servidor, sus opciones adicionales permiten manejar mejor las interacciones con servicios web y REST APIs.

¿Qué es la API Fetch?

La API Fetch es una herramienta que permite hacer peticiones HTTP de manera sencilla y eficiente. Está diseñada para manejar recursos de forma asíncrona, lo cual la convierte en una alternativa más flexible que *XMLHttpRequest*. En Fetch, las promesas reemplazan las devoluciones de llamada (*callbacks*), haciendo que el código sea más limpio y fácil de manejar, especialmente en casos complejos donde se encadenan varias peticiones o se usan *async/await*.

Sintaxis básica de Fetch

La estructura básica de una petición con Fetch es:

```
let promise = fetch(url, [options]);
```

Donde: La función *fetch()* acepta dos parámetros:

- **url:** Representa la dirección del recurso al que queremos acceder.
- **options:** Parámetros adicionales que configuran la solicitud, como el método HTTP, los encabezados, el cuerpo de la solicitud, entre otros.

Opciones de configuración

Algunas de las opciones más comunes que se pueden especificar en **{ options }** incluyen:

- **method:** Especifica el método *HTTP* de la solicitud. El valor predeterminado es *GET*, pero puede ser *POST*, *PUT*, *DELETE*, *HEAD*, etc.
- **headers:** Encabezados HTTP para la solicitud, proporcionados como un objeto {}.
- **body:** El contenido del cuerpo de la solicitud. Solo se usa con métodos que envían datos (como *POST* o *PUT*) y puede ser de varios tipos (*String*, *FormData*, *Blob*, etc.).
- **cache:** Controla el almacenamiento en caché, con opciones como *default*, *no-cache*, *reload*, etc.

- **mode**: Controla el modo de solicitud, como *cors*, *no-cors*, o *same-origin*.
- **redirect**: Define cómo manejar redireccionamientos, con opciones *follow*, *error* o *manual*.
- **referrer**: Permite establecer la referencia de la solicitud.
- **signal**: Asocia una señal **AbortSignal** que puede usarse para cancelar la solicitud.
- **credentials**: Controla el envío de cookies con la solicitud. Las opciones incluyen *omit* (no envía cookies), *same-origin* (solo envía cookies en el mismo origen), o *include* (envía cookies en todas las solicitudes).

Ejemplo básico de una solicitud GET:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json()) // Convierte la respuesta a JSON
  .then(data => {
    console.log("Datos de usuarios:", data);
    data.forEach(usuario => {
      console.log(`Nombre: ${usuario.name} - Email: ${usuario.email}`);
    });
  })
  .catch(error => console.error("Error en la solicitud:", error));
```

Obtener una respuesta.

Por lo general, obtener una respuesta es un proceso de dos pasos:

Primer paso

La promesa **promise**, devuelta por **fetch()**, resuelve la respuesta en un objeto de la clase **Response** en cuanto el servidor responde con los encabezados HTTP de la petición. En este paso, podemos acceder al código de estado HTTP (*status*) para ver si la petición ha sido exitosa o no y revisar los encabezados, pero el cuerpo de la respuesta aún no está disponible.

- La promesa será rechazada si **fetch()** no ha podido establecer la petición HTTP, por ejemplo, por problemas de red o si el sitio especificado en la petición no existe.
- Los estados HTTP anormales, como el 404 o 500 no generan errores.

Podemos visualizar los estados HTTP en las propiedades de la respuesta:

status → código de estado HTTP, por ejemplo: 200.

ok → booleana, true si el código de estado HTTP es 200 a 299.

Segundo paso:

Para obtener el cuerpo de la respuesta, debemos usar uno de los métodos del objeto Response. Estos métodos basados en promesas para acceder al cuerpo de la respuesta en distintos formatos:

- **response.text()** → lee y devuelve la respuesta en formato texto.
- **response.json()** → convierte la respuesta como un JSON.

Ejemplo de manejo de una respuesta JSON:

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => {
    if (!response.ok) {
      throw new Error(`Error: ${response.status}`);
    }
    return response.json(); // lee el cuerpo de la respuesta como JSON
  })
  .then(data => console.log("Usuarios:", data))
  .catch(error => console.error("Error en la solicitud:", error));
```

Propiedades de response.

Estas propiedades permiten manejar de manera efectiva las respuestas de las solicitudes realizadas con la API Fetch.

- **status:** Código de estado HTTP de la respuesta, por ejemplo, 200 para una solicitud exitosa.
- **statusText:** Texto asociado al código de estado, como "OK" para el código 200.
- **ok:** Valor booleano que indica si la respuesta fue exitosa (códigos de estado en el rango 200-299).
- **headers:** Objeto Headers que contiene los encabezados de la respuesta.
- **url:** URL de la solicitud que generó esta respuesta.
- **type:** Tipo de respuesta, que puede ser basic, cors, error, entre otros.
- **redirected:** Indica si la respuesta es el resultado de una redirección.

Métodos de response.

Además de las propiedades anteriores, el objeto Response implementa la interfaz Body, lo que proporciona métodos para acceder al contenido del cuerpo de la respuesta en diferentes formatos:

- **text():** Lee y devuelve el cuerpo de la respuesta como una cadena de texto.
- **json():** Interpreta el cuerpo de la respuesta como JSON y devuelve el objeto resultante.
- **formData():** Devuelve el cuerpo de la respuesta como un objeto FormData, útil para manejar datos de formularios.
- **blob():** Devuelve el cuerpo de la respuesta como un objeto Blob, útil para manejar datos binarios (imágenes, videos, etc.).
- **arrayBuffer():** Devuelve el cuerpo de la respuesta como un ArrayBuffer, adecuado para manejar datos binarios de bajo nivel.

Más información sobre Response:

<https://developer.mozilla.org/es/docs/Web/API/Response>

Manejo de errores.

La promesa que retorna `fetch()` **se rechaza** solo si ocurre un problema de red o si la URL solicitada tiene un formato incorrecto (no sigue las reglas estándar y el navegador no puede procesarla) o apunta a un dominio inexistente (el navegador no podrá establecer una conexión).

En estos casos, la promesa de `fetch()` **se rechaza** porque no se puede establecer una conexión HTTP. No llega a haber respuesta del servidor, porque la solicitud nunca llega a realizarse.

```
fetch('/localhost/php/coop..php') // URL malformada
  .catch(error => console.error('URL Inválida:', error));
```

Los errores HTTP (`404` o `500`) **no generan un rechazo** de la promesa, por lo que debemos verificar el estado de la respuesta usando `response.ok` o `response.status` para detectar errores HTTP.

Por ejemplo, el código de estado 404 (Página no encontrada), una URL válida que lleva a un servidor real, pero en la que el recurso específico solicitado no existe, la URL es bien entendida por el navegador y el servidor. Sin embargo, el servidor no encuentra el recurso (página, archivo, API, etc.) solicitado en esa ruta específica.

En estos casos, la promesa de `fetch()` **no se rechaza**, sino que se resuelve, porque la solicitud llegó al servidor. Sin embargo, `response.ok` será `false` y `response.status` será `404`.

```
fetch('//localhost/php/noexiste.php') // Recurso inexistente en servidor existente
  .then(response => {
    if (!response.ok) {
      console.error('Código 404: Página no encontrada');
    }
  })
  .catch(error => console.error('Otro error:', error));
```

```
fetch("https://jsonplaceholder.typicode.com/usuarios")
  .then(response => {
    if (!response.ok) {
      throw new Error(`Error HTTP: ${response.status}`);
    }
    return response.json();
  })
  .then(data => console.log("Datos obtenidos:", data))
  .catch(error => console.error("Error en la solicitud:", error));
```

Uso con async/await.

La API Fetch se integra muy bien con **async/await**, lo cual hace que el código asíncrono sea más fácil de leer y de estructurar.

Ejemplo con async/await:

```
async function obtenerUsuarios() {  
  try {  
    const response = await fetch("https://jsonplaceholder.typicode.com/users");  
    if (!response.ok) {  
      throw new Error(`Error HTTP: ${response.status}`);  
    }  
    const data = await response.json();  
    console.log("Usuarios:", data);  
  } catch (error) {  
    console.error("Error en la solicitud:", error);  
  }  
}  
  
obtenerUsuarios();
```


Métodos HTTP con fetch.

Fetch permite realizar cualquier método HTTP (GET, POST, PUT, DELETE, etc.) configurando el valor de `method` en las opciones de la solicitud.

- Solicitud **GET**: La solicitud **GET** es el método predeterminado en Fetch y se usa para obtener datos del servidor.

```
fetch('https://jsonplaceholder.typicode.com/posts')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

- Solicitud **POST**: La solicitud **POST** se utiliza para enviar datos al servidor. Es común en formularios de registro, inicio de sesión y envío de datos del usuario.

```
fetch('https://jsonplaceholder.typicode.com/posts', {  
  method: 'POST',  
  headers: { 'Content-Type': 'application/json' },  
  body: JSON.stringify({ title: 'Nuevo post', body: 'Contenido', userId: 1  
}))  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

- Solicitud **PUT**: La solicitud **PUT** se utiliza para actualizar recursos en el servidor.

```
fetch('https://jsonplaceholder.typicode.com/posts/1', {  
  method: 'PUT',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({ title: 'Nuevo post', body: 'Nuevo Contenido' })  
}))  
  .then(response => response.json())  
  .then(data => console.log('Actualización exitosa:', data))  
  .catch(error => console.error('Error:', error));
```

- Solicitud **DELETE**: La solicitud *DELETE* se utiliza para eliminar un recurso específico en el servidor.

```
fetch('https://jsonplaceholder.typicode.com/posts/1', {
  method: 'DELETE'
})
.then(response => {
  if (response.ok) console.log('Recurso eliminado');
  else console.log('Error al eliminar');
})
.catch(error => console.error('Error:', error));
```

Uso de headers con fetch.

A menudo, es necesario incluir encabezados (*headers*) para especificar el tipo de contenido, autorización, tokens, etc. Esto se hace en el objeto de opciones:

Encabezados de respuesta.

Los encabezados de respuesta están disponibles como un objeto de tipo *Map* dentro del *response.headers*. No es exactamente un Map, pero posee métodos similares para obtener de manera individual encabezados por nombre o bien recorrerlos como un objeto:

```
let response = async fetch('https://api.github.com/users/maria');
// obtenemos un encabezado
away console.log('Encabezados -->', response.headers.get('Content-Type'))
// application/json; charset=utf-8
// iteramos todos los encabezados
for (let [key, value] of
  response.headers) {
  console.log(`${key} = ${value}`);
}
```

Encabezados de petición.

Para especificar un encabezado en nuestra petición, podemos utilizar la opción **headers**. La misma posee un objeto con los encabezados salientes, como se muestra en el siguiente ejemplo:

```
// solicitud GET (request) pasando headers
fetch('https://api.github.com/users/maria',{
  method: 'GET',
  headers: {"Content-type": "application/json;charset=UTF-8"}
})
.then(response => response.json()) //convertir a json
.then(json => console.log('datos usuario maria ---> ', json)) //mostrar datos
.catch(err => console.error('Error: ', err)); //capturar errores
```

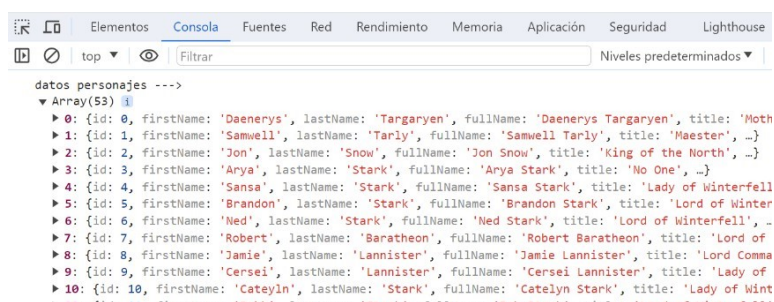
Ejemplo petición XMLHttpRequest y petición fetch.

Petición a la API publica de Juego de Tronos para obtener los personajes:

Petición con XMLHttpRequest:

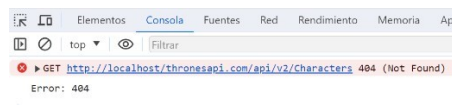
```
let peticion = new XMLHttpRequest();
peticion.open('GET', 'https://thronesapi.com/api/v2/Characters')
peticion.addEventListener('load', () => {
  if (peticion.status === 200) {
    let data = JSON.parse(peticion.responseText);
    console.log('datos personajes ---> ', data); // mostrar datos
  } else {
    console.log('Error: ' + peticion.status);
  }
})
peticion.send();
```

El código envía una solicitud *GET* a <https://thronesapi.com/api/v2/Characters> para obtener la información sobre los personajes de juego de tronos. Si la respuesta es exitosa, imprimirá el siguiente JSON en la consola:



```
datos personajes --->
▼ Array(10)
  ▶ 0: {id: 0, firstName: 'Daenerys', lastName: 'Targaryen', fullName: 'Daenerys Targaryen', title: 'Moth...}
  ▶ 1: {id: 1, firstName: 'Samwell', lastName: 'Tarly', fullName: 'Samwell Tarly', title: 'Maester', ...}
  ▶ 2: {id: 2, firstName: 'Jon', lastName: 'Snow', fullName: 'Jon Snow', title: 'King of the North', ...}
  ▶ 3: {id: 3, firstName: 'Arya', lastName: 'Stark', fullName: 'Arya Stark', title: 'No One', ...}
  ▶ 4: {id: 4, firstName: 'Sansa', lastName: 'Stark', fullName: 'Sansa Stark', title: 'Lady of Winterfell', ...}
  ▶ 5: {id: 5, firstName: 'Brandon', lastName: 'Stark', fullName: 'Brandon Stark', title: 'Lord of Winter...}
  ▶ 6: {id: 6, firstName: 'Ned', lastName: 'Stark', fullName: 'Ned Stark', title: 'Lord of Winterfell', ...}
  ▶ 7: {id: 7, firstName: 'Robert', lastName: 'Baratheon', fullName: 'Robert Baratheon', title: 'Lord of...}
  ▶ 8: {id: 8, firstName: 'Jamie', lastName: 'Lannister', fullName: 'Jamie Lannister', title: 'Lord Comma...}
  ▶ 9: {id: 9, firstName: 'Cersei', lastName: 'Lannister', fullName: 'Cersei Lannister', title: 'Lady of...}
  ▶ 10: {id: 10, firstName: 'Catelyn', lastName: 'Stark', fullName: 'Catelyn Stark', title: 'Lady of Wint...
```

Si la solicitud falla, imprimirá este mensaje de error en la consola:



```
Error: 404
```

Petición con fetch():

```
fetch('https://thronesapi.com/api/v2/Characters')
  .then(response => response.json()) //convertir a json
  .then(json => console.log('datos personajes ---> ', json)) //mostrar datos
  .catch(err => console.error('Error: ', err)); //capturar errores
```

Copiar el código y probar en la consola, comprobando que el resultado es el mismo que en el ejemplo anterior.

Diferencias entre XMLHttpRequest y fetch.

- Hay una diferencia importante entre el objeto de respuesta en *XMLHttpRequest* y *fetch*. El primero devuelve los datos como respuesta, mientras que el objeto de respuesta de *fetch* contiene información sobre el objeto de respuesta en sí mismo. Esto incluye *headers*, *status code*, etc.
- En *fetch*, llamamos a la función *response.json()* para obtener los datos que necesitamos del objeto de respuesta.
- Otra diferencia importante es que *fetch()* no generará un error si la solicitud devuelve un código de estado 400 o 500. Todavía se marcará como una respuesta exitosa y se pasará a la función *.then*.
- *Fetch* solo generará un error si la solicitud en sí se interrumpe. Para manejar respuestas 400 y 500, podemos escribir una lógica personalizada usando *response.status*. La propiedad *status* nos dará el código de estado de la respuesta devuelta.

```
fetch('https://thronesapi.com/api/v2/Characters')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json(); // convertir respuesta a JSON
  })
  .then(data => {
    console.log('datos personajes ---> ', data); // mostrar datos
  })
  .catch(error => {
    console.error('Error: ', error);
  });
```

Ejemplos fetch con base de datos.

Ejemplo: Registro de un socio en la tabla socios.

```
const paramSocio = new FormData();
paramSocio.append('opcion', 'RS');
paramSocio.append('nombre', 'Ejemplo');
paramSocio.append('apellidos', 'Registro con Fetch');
paramSocio.append('email', 'prueba.fetch@dwc.daw.es');
paramSocio.append('password', 'dwc111');
fetch('//localhost/php/coop.php', {
  method: 'POST',
  body: paramSocio,
})
.then(respuesta => respuesta.json())
.then(datos => {
  if (datos.ok) {
    console.log('Usuario registrado');
  } else {
    console.log('Error al registrar el socio ');
  }
})
.catch(error => console.error('Error en la petición:', error));
```

Ejemplo: Consulta de un socio de la tabla socios.

```
const params = new FormData();
params.append('opcion', 'SR');
params.append('email', 'prueba.fetch@dwc.daw.es');
params.append('password', 'dwc111');
fetch('//localhost/php/coop.php', {
  method: 'POST',
  body: params,
})
.then((response) => {
  if (!response.ok) {
    console.log(`Error en la respuesta ${response.status} -
${response.statusText}`);
  }
  return response.json();
})
.then((datos) => {
  if (datos.error) {
    console.log('Error usuario no registrado')
  } else {
    const datosUsuario = datos[0];
    console.log(`Id: ${datosUsuario.id}`);
    console.log(`Nombre: ${datosUsuario.nombre}`);
    console.log(`Correo: ${datosUsuario.email}`);
  }
})
.catch((error) => console.error('Error en la petición:', error));
```