Material descargable

egg

Teoría **APIs**

En el apasionante mundo del desarrollo frontend, las APIs (Interfaz de Programación de Aplicaciones) se presentan como una pieza clave que desbloquea un sinfín de posibilidades para construir aplicaciones web modernas, interactivas y conectadas. **Las APIs actúan conectando aplicaciones y servicios, lo que permite el intercambio de datos y funcionalidades cruciales para mejorar la experiencia del usuario y crear soluciones innovadoras.**

La necesidad de comprender y utilizar APIs radica en la capacidad que brindan para acceder a datos y servicios externos, abriendo puertas hacia un vasto universo de información y recursos. Gracias a ellas, los desarrolladores frontend pueden acceder a bases de datos, sistemas de autenticación, servicios de pago, mapas y muchos otros servicios, sin tener que preocuparse por la complejidad de su implementación. Esto ahorra tiempo y esfuerzo, permitiendo enfocarse en la creación de interfaces de usuario intuitivas y atractivas.

Además de facilitar la comunicación entre diferentes aplicaciones y servicios, las APIs también **promueven la colaboración y la interoperabilidad entre desarrolladores y equipos.** Las empresas y organizaciones ofrecen APIs públicas para que terceros desarrolladores puedan construir aplicaciones que se integren con sus servicios. Esto fomenta la creación de ecosistemas y plataformas enriquecidos que pueden interactuar entre sí, brindando a los usuarios experiencias integradas y enriquecedoras.

Comprender cómo consumir y utilizar APIs desde el lado del cliente (frontend) abre un abanico de oportunidades para crear aplicaciones web más completas y poderosas. Los datos en tiempo real, las actualizaciones dinámicas y la interacción constante con servicios externos son ahora posibles gracias a la magia de las APIs. Por lo tanto, aprender sobre ellas es un paso imprescindible para cualquier

desarrollador frontend que desee crear aplicaciones web modernas y a la vanguardia de la tecnología.

En este emocionante viaje, exploraremos cómo las APIs son fundamentales en el desarrollo frontend, las mejores prácticas para consumirlas, cómo aprovechar su potencial en proyectos prácticos y cómo integrarlas en aplicaciones web que se conectan con el mundo. ¡Prepárate para sumergirte en el fascinante universo de las APIs y descubrir todo lo que pueden ofrecer en el desarrollo de aplicaciones web!

¿Qué es una API y por qué son importantes para el desarrollo frontend?

En el mundo del desarrollo frontend, una API (Interfaz de Programación de Aplicaciones) es una puerta de enlace que permite que distintas aplicaciones y servicios se comuniquen y compartan datos entre sí. Imagina una API como un conjunto de reglas y protocolos que facilitan la interacción entre aplicaciones, permitiendo que nuestros proyectos frontend accedan a información y funcionalidades de fuentes externas.

Importancia de las APIs en el desarrollo frontend

Las APIs son fundamentales para el desarrollo frontend por varias razones clave:

- **Acceso a datos y servicios externos:** Con las APIs, podemos acceder a datos, recursos y funcionalidades que residen en servidores externos. Esto es invaluable para enriquecer nuestras aplicaciones web con información actualizada y funciones especializadas sin tener que implementar todo desde cero.
- **Reutilización de servicios:** Las APIs nos permiten utilizar servicios que otras empresas u organizaciones han construido y puesto a disposición para uso público. Esto ahorra tiempo y esfuerzo, ya que no es necesario reinventar la rueda y podemos concentrarnos en la creación de experiencias de usuario excepcionales.

- **Construcción de aplicaciones más completas:** Al utilizar APIs, podemos combinar diferentes servicios y datos para crear aplicaciones más completas y potentes. Podemos integrar mapas, sistemas de autenticación, métodos de pago, análisis y mucho más para brindar experiencias enriquecidas y versátiles a los usuarios.
- **Fomento de la colaboración:** Las APIs abren la puerta a la colaboración entre diferentes equipos y desarrolladores. Las empresas pueden ofrecer APIs públicas para que terceros puedan construir aplicaciones que se integren con sus servicios, creando ecosistemas y plataformas enriquecidos.
- **Escalabilidad y modularidad:** Al construir aplicaciones frontend que interactúan con APIs, podemos lograr una arquitectura más modular y escalable. Las actualizaciones y mejoras pueden realizarse en los servidores externos sin afectar significativamente el código de nuestra aplicación frontend.

El protocolo HTTP (Hypertext Transfer Protocol)

El Protocolo de Transferencia de Hipertexto (HTTP) es el lenguaje común que utilizan los navegadores web y los servidores para comunicarse y transferir datos en la World Wide Web. Es la base fundamental de cómo se realiza la comunicación entre clientes (como navegadores web) y servidores, y es esencial en el consumo de APIs.

Por qué se usa el protocolo HTTP en el consumo de APIs

- **Comunicación cliente-servidor:** HTTP proporciona un conjunto de reglas que permiten que el cliente (generalmente un navegador web) envíe solicitudes a un servidor y reciba respuestas del mismo. Esto es fundamental para la interacción entre una aplicación frontend y una API que reside en un servidor externo.
- **Simplicidad y ligereza:** HTTP es un protocolo simple y ligero, lo que lo hace ideal para el consumo de APIs. Su facilidad de uso permite que los desarrolladores envíen solicitudes y reciban respuestas sin complicaciones excesivas.

- **Flexibilidad de métodos:** HTTP ofrece una variedad de métodos o verbos para las solicitudes, como GET, POST, PUT y DELETE. Cada método tiene una función específica, lo que permite que las aplicaciones frontend puedan realizar diferentes tipos de interacciones con la API, como obtener datos, enviar datos, actualizar información o eliminar recursos.

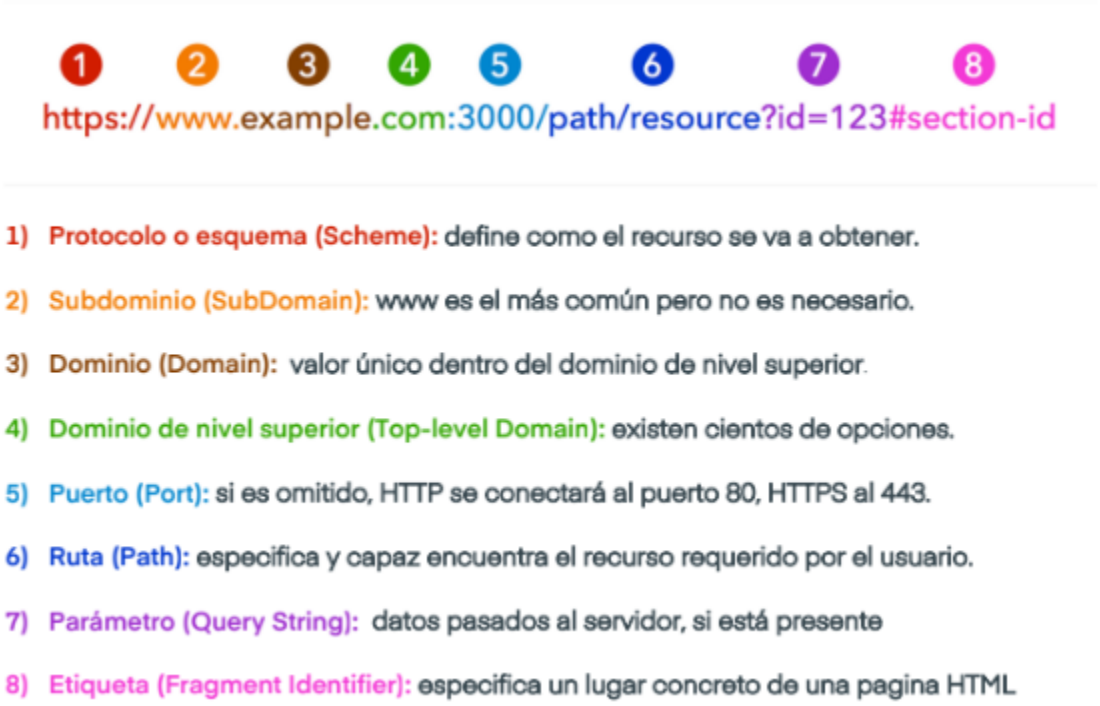
- **Respuestas estructuradas:** Las respuestas que se obtienen mediante HTTP siguen una estructura bien definida. Las API generalmente responden en formatos como JSON o XML, lo que facilita el procesamiento de los datos recibidos por parte de las aplicaciones frontend.
- **Acceso a datos y recursos:** Al utilizar HTTP para interactuar con APIs, las aplicaciones frontend pueden acceder a datos y recursos alojados en servidores remotos. Esto permite que nuestras aplicaciones se conecten a bases de datos, servicios de autenticación, sistemas de pago y otros recursos externos.

Estructura de una solicitud HTTP:

Una solicitud HTTP consta de varias partes que permiten al cliente comunicar su intención al servidor. La estructura básica de una solicitud HTTP es la siguiente:

➔ **Método:** Indica el tipo de acción que se quiere realizar en el servidor. Los métodos más comunes son GET (obtener datos), POST (enviar datos), PUT (actualizar datos) y DELETE (eliminar recursos).

➔ **URL (Uniform Resource Locator):** Es la dirección del recurso al que se quiere acceder en el servidor.



1. Protocolo

El esquema suele ser el nombre de un protocolo, que define cómo se obtendrá el recurso. Sin embargo, esquemas como "archivo" y "mailto" no especifican un protocolo. Los clientes, como los navegadores web, se conectan a sitios web a través del Protocolo de transferencia de Hipertexto, también conocido como HTTP.

2. Subdominio

Aunque www es el más común, los subdominios se pueden configurar con cualquier valor que consista en caracteres ASCII alfanuméricos que no distinguen entre mayúsculas y minúsculas. Se permiten guiones si están rodeados por otros caracteres ASCII u otros guiones. Los subdominios también se pueden eliminar, acortando y simplificando toda la URL.

3-4. Dominio + nivel superior

En octubre de 1984, el conjunto original de dominios de nivel superior se definió como:

- .com

• .edu

• .gov

• .mil

• .org

• .neto

Desde 1984 se crearon un puñado de nuevos dominios de nivel superior, pero el cambio más significativo se produjo en 2012 cuando ICANN (Corporación de Internet para la Asignación de Nombres y Números) autorizó la creación de casi dos mil nuevos dominios de nivel superior.

5. Puerto

El propósito de un puerto es identificar de forma única diferentes procesos o aplicaciones que se ejecutan en un solo servidor. Los números de puerto permiten que cada proceso o aplicación comparta una conexión de red. Las URL públicas a menudo no incluyen un número de puerto. En esos casos, se utiliza el puerto predeterminado del esquema. Los puertos predeterminados para HTTP y HTTPS son 80 y 443, respectivamente.

6. Ruta

Una ruta describe una ubicación específica dentro de un sistema de archivos. En el contexto de una URL HTTP, una ruta describe un recurso específico, como un documento HTML.

7. Parámetro

Los parámetros contienen datos que se enviarán al servidor para procesamiento adicional. Puede contener pares de nombre / valor separados por un ampersand (&), así:

?first_name=Alfred&last_name=Pennyworth

En el ejemplo anterior, los datos serían:

"Nombre: Alfred"

"Apellido: Pennyworth"

Luego, estos datos se pasan a los scripts del lado del servidor para su procesamiento. Si esta es una consulta válida, el servidor devolverá información pertinente a Alfred Pennyworth.

8. Etiqueta

Las etiquetas en una URL aparecen después del hashtag #.

Su función, entre otras cosas, consiste en permitir hacer scroll hasta un elemento en concreto. Por ejemplo, si mandamos a alguien una URL que contenga una etiqueta, ésta le dirigirá a la parte exacta de la página en cuestión.

- ➔ **Cabeceras (Headers):** Proporcionan información adicional sobre la solicitud, como el tipo de contenido que se espera recibir o la autenticación del cliente.
- ➔ **Cuerpo (Body):** Solo se incluye en las solicitudes POST y PUT, y contiene los datos que se desean enviar al servidor.

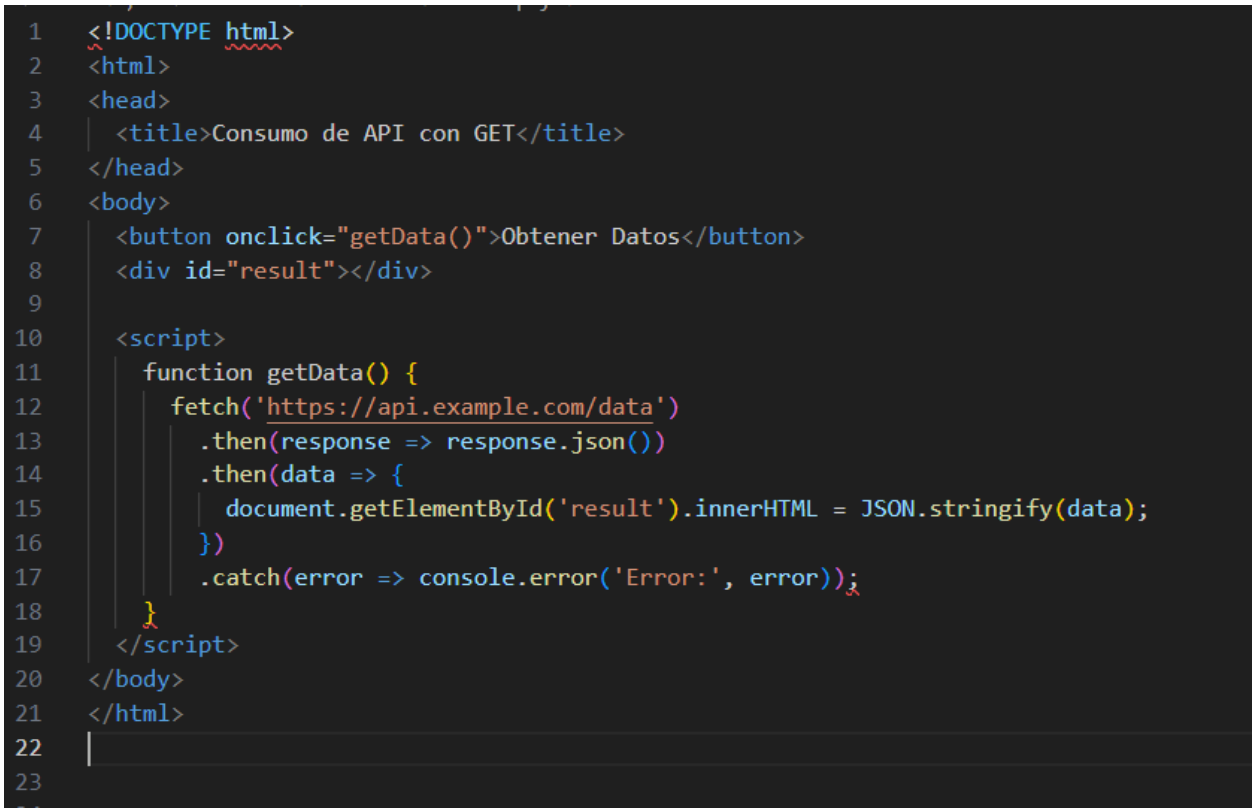
Métodos HTTP utilizados en el consumo de APIs (GET, POST, PUT, DELETE)

En el consumo de APIs, los métodos HTTP juegan un papel fundamental para realizar diferentes tipos de acciones en los recursos de la API. Cada método tiene una función específica y nos permite interactuar con los datos y recursos del servidor de diferentes maneras. A continuación, exploraremos los cuatro métodos más comunes utilizados en el consumo de APIs y veremos ejemplos de cómo se aplican en nuestras solicitudes.

1. Método GET:

El método GET se utiliza para obtener datos y recursos del servidor. Es la solicitud más simple y común en el consumo de APIs, y se utiliza para recuperar información sin realizar cambios en el servidor.

Ejemplo de solicitud GET utilizando JavaScript y Fetch API:



```html
<!DOCTYPE html>
<html>
<head>
  <title>Consumo de API con GET</title>
</head>
<body>
  <button onclick="getData()">Obtener Datos</button>
  <div id="result"></div>

  <script>
    function getData() {
      fetch('https://api.example.com/data')
        .then(response => response.json())
        .then(data => {
          document.getElementById('result').innerHTML = JSON.stringify(data);
        })
        .catch(error => console.error('Error:', error));
    }
  </script>
</body>
</html>
```

2. Método POST:

El método POST se utiliza para enviar datos al servidor y crear nuevos recursos. Es comúnmente utilizado en formularios y cuando necesitamos enviar información para su procesamiento o almacenamiento.

Ejemplo de solicitud POST utilizando JavaScript y Fetch API:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Consumo de API con POST</title>
</head>
<body>
  <form onsubmit="postData(event)">
    <label for="name">Nombre:</label>
    <input type="text" id="name" required>
    <button type="submit">Enviar</button>
  </form>
  <div id="response"></div>

  <script>
    function postData(event) {
      event.preventDefault();
      const name = document.getElementById('name').value;
      fetch('https://api.example.com/data', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ name: name })
      })
        .then(response => response.json())
        .then(data => {
          document.getElementById('response').innerHTML = `Respuesta: ${JSON.stringify(data)}`;
        })
        .catch(error => console.error('Error:', error));
    }
  </script>
</body>
```

3. Método PUT:

El método PUT se utiliza para actualizar datos y recursos existentes en el servidor. Es útil cuando necesitamos modificar información existente.

Ejemplo de solicitud PUT utilizando JavaScript y Fetch API:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Consumo de API con PUT</title>
</head>
<body>
  <button onclick="updateData()">Actualizar Datos</button>
  <div id="result"></div>

  <script>
    function updateData() {
      const newData = { name: 'Nuevo nombre' };
      fetch('https://api.example.com/data', {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(newData)
      })
        .then(response => response.json())
        .then(data => {
          document.getElementById('result').innerHTML = JSON.stringify(data);
        })
        .catch(error => console.error('Error:', error));
    }
  </script>
</body>
</html>
```

4. Método DELETE:

El método DELETE se utiliza para eliminar recursos y datos del servidor.

Ejemplo de solicitud DELETE utilizando JavaScript y Fetch API:

```html
<!DOCTYPE html>
<html>
<head>
  <title>Consumo de API con DELETE</title>
</head>
<body>
  <button onclick="deleteData()">Eliminar Datos</button>
  <div id="result"></div>

  <script>
    function deleteData() {
      fetch('https://api.example.com/data', {
        method: 'DELETE',
      })
        .then(response => {
          if (response.ok) {
            document.getElementById('result').innerHTML = 'Datos eliminados con éxito';
          } else {
            document.getElementById('result').innerHTML = 'Error al eliminar datos';
          }
        })
        .catch(error => console.error('Error:', error));
    }
  </script>
</body>
</html>

```

Tipos de Respuesta: HTML y JSON

Cuando consumimos APIs, las respuestas que recibimos del servidor pueden venir en diferentes formatos. Dos de los formatos más comunes son HTML y JSON.

- ➔ **Respuesta HTML:**

El formato HTML (Hypertext Markup Language) es el lenguaje estándar para crear páginas web. Si hacemos una solicitud a una API y obtenemos una respuesta en formato HTML, esto significa que el servidor nos ha devuelto una página web completa que puede ser renderizada directamente en el navegador. Este formato es útil cuando necesitamos mostrar información estructurada con estilos y diseño, como páginas web completas o contenido formateado.

Ejemplo de solicitud y respuesta HTML utilizando JavaScript:

```
fetch('https://api.example.com/page')
  .then(response => response.text())
  // Obtener el contenido HTML como texto
  .then(html => {
    document.getElementById('content').innerHTML = html;
    // Mostrar el contenido en un elemento HTML del documento
  })
  .catch(error => console.error('Error:', error));

```

➔ **Respuesta JSON:**

JSON (JavaScript Object Notation) es un formato ligero y fácil de leer utilizado para el intercambio de datos. Es ampliamente utilizado en el consumo de APIs, ya que permite transmitir y recibir datos estructurados de manera eficiente. Cuando hacemos una solicitud a una API y recibimos una respuesta en formato JSON, esto significa que el servidor nos ha devuelto datos en un formato que puede ser fácilmente manipulado y utilizado en nuestra aplicación.

Ejemplo de solicitud y respuesta JSON utilizando JavaScript:

```
fetch('https://api.example.com/data')
  .then(response => response.json()) // Parsear la respuesta como JSON
  .then(data => {
    console.log(data); // Utilizar los datos recibidos en la respuesta
  })
  .catch(error => console.error('Error:', error));
```

Arquitectura REST (Representational State Transfer):

REST es una arquitectura de estilo para el diseño de sistemas de comunicación en red. Se basa en el protocolo HTTP y establece una serie de reglas y principios que definen cómo deben ser diseñadas y estructuradas las APIs.

Principios de la arquitectura REST

Recursos identificables: Cada recurso de la API debe tener una identificación única y estar representado por una URL específica.

- **Verbos HTTP:** Utiliza los métodos HTTP (GET, POST, PUT, DELETE) para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en los recursos.
- **Uso de URI (Identificadores de Recursos Uniformes):** Las URLs de las APIs deben ser significativas y fáciles de entender, representando de manera semántica los recursos y sus relaciones.
- **Estado de la aplicación:** REST es "stateless", lo que significa que cada solicitud del cliente al servidor debe contener toda la información necesaria para comprenderla, sin depender de estados previos.
- **Respuestas con formato estándar**: Las respuestas de la API deben utilizar formatos estándar como JSON o XML para facilitar su procesamiento y comprensión.

Ventajas de la arquitectura REST

- **Simplicidad y facilidad de uso:** REST es fácil de entender y utilizar, lo que facilita el desarrollo y la integración de APIs.
- **Escalabilidad:** La arquitectura REST permite diseñar APIs escalables que pueden manejar grandes volúmenes de peticiones.
- **Interoperabilidad:** REST permite que las APIs sean independientes del lenguaje de programación y la plataforma, lo que facilita la interoperabilidad entre diferentes sistemas.
- **Facilita el cacheo:** RESTful APIs pueden aprovechar el cacheo para mejorar la eficiencia y reducir la carga del servidor.
- **Ampliamente adoptado:** REST es ampliamente utilizado en el desarrollo web y es un estándar de facto para la creación de APIs.

Repaso de conceptos básicos de JavaScript

En este repaso, reforzaremos los conceptos esenciales de JavaScript necesarios para el consumo efectivo de APIs. Aquí tienes una guía ejemplificada para tener a mano:

Variables y tipos de datos

Declarar una variable:

```
let nombre = "Juan";
const edad = 30;
```

Tipos de datos en JavaScript:

```
let numero = 10; // número
let texto = "Hola"; // cadena de texto
let esVerdadero = true; // booleano (true o false)
let miArray = [1, 2, 3]; // array
let miObjeto = { nombre: "Juan", edad: 30 }; // objeto
```

Funciones

Definir una función:

```
function saludar(nombre) {
  console.log(`¡Hola, ${nombre}!`);
}

- Llamar una función:
```javascript
saludar("Ana"); // Salida: ¡Hola, Ana!
```

Función con valor de retorno:
```

```
function suma(a, b) {
  return a + b;
}

let resultado = suma(5, 3); // resultado = 8
```

Objetos y propiedades

Crear un objeto:

```
let persona = {
  nombre: "Carlos",
  edad: 25,
  ciudad: "Madrid"
};
```

Acceder a las propiedades del objeto:

```
console.log(persona.nombre); // Salida: Carlos
console.log(persona.edad); // Salida: 25
```

Arrays y manipulación de datos

Crear un array:

```
let frutas = ["manzana", "pera", "plátano"];
```

Agregar elementos al array:

```
frutas.push("naranja");
```

Eliminar elementos del array:

```
frutas.pop(); // Elimina el último elemento (naranja)
```

Modificar elementos del array:

```
frutas[1] = "uva"; // Cambia "pera" por "uva"
```

Uso de la Fetch API para realizar solicitudes HTTP desde el navegador

La Fetch API es una poderosa herramienta que nos permite realizar solicitudes HTTP desde el navegador de manera sencilla y eficiente. Es compatible con los navegadores modernos y proporciona una interfaz más amigable para trabajar con peticiones y respuestas en comparación con las antiguas técnicas basadas en XMLHttpRequest. En este tema, exploraremos cómo utilizar la Fetch API para interactuar con APIs externas y obtener datos dinámicos en nuestras aplicaciones web.

Realizando una solicitud GET

La solicitud GET es una de las operaciones HTTP más comunes y se utiliza para obtener datos de un servidor. Utilizamos la Fetch API para realizar esta solicitud de la siguiente manera:

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => {
    console.log(data); // Procesamos los datos recibidos
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Realizando una solicitud POST

La solicitud POST se utiliza para enviar datos al servidor para su procesamiento o almacenamiento. Podemos enviar datos en el cuerpo de la solicitud utilizando el método fetch y especificando el método POST en las opciones:

```
const newData = { nombre: 'Ejemplo', edad: 25 };

fetch('https://api.example.com/data', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(newData)
})
  .then(response => response.json())
  .then(data => {
    console.log(data); // Procesamos los datos recibidos después del envío
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Realizando una solicitud PUT

La solicitud PUT se utiliza para actualizar datos existentes en el servidor. Funciona de manera similar a la solicitud POST, pero se aplica a un recurso específico que ya existe:

```
const updatedData = { nombre: 'Nuevo Ejemplo', edad: 30 };

fetch('https://api.example.com/data/1', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(updatedData)
})
  .then(response => response.json())
  .then(data => {
    console.log(data); // Procesamos los datos recibidos después de la actualización
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Realizando una solicitud DELETE

La solicitud DELETE se utiliza para eliminar datos del servidor. Simplemente especificamos el método DELETE en las opciones:

```
fetch('https://api.example.com/data/1', {
  method: 'DELETE'
})
  .then(response => {
    if (response.ok) {
      console.log('Recurso eliminado con éxito');
    } else {
      console.log('Error al eliminar recurso');
    }
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Ejemplo real paso a paso: Consumir una API de clima

A continuación, desarrollaremos un ejemplo real paso a paso para consumir una API de clima y mostrar la temperatura actual en una página web.

Primero, realizaremos una solicitud GET a la API del clima:

```
fetch('https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
  })
  .then(data => {
    console.log(data); // Procesamos los datos recibidos en formato JSON
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

A continuación, accederemos a los datos necesarios (en este caso, la temperatura actual) del objeto de respuesta:

```
fetch('https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
  })
  .then(data => {
    const temperaturaActual = data.main.temp;
    console.log(`La temperatura actual en Londres es de ${temperaturaActual} grados Celsius.`);
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Finalmente, mostraremos la temperatura en el HTML de nuestra página web:

```
fetch('https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_API_KEY')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la solicitud');
    }
    return response.json();
  })
  .then(data => {
    const temperaturaActual = data.main.temp;
    const temperaturaElement = document.getElementById('temperatura');
    temperaturaElement.textContent = `La temperatura actual en Londres es de ${temperaturaActual} grados Celsius.`;
  })
  .catch(error => {
    console.error('Error:', error); // Manejamos errores si los hay
  });
```

Con este ejemplo, hemos utilizado la Fetch API para realizar una solicitud a una API de clima, procesar la respuesta en formato JSON y mostrar la temperatura actual en una página web. Este es solo un ejemplo básico, pero con este conocimiento, los estudiantes podrán crear aplicaciones más complejas y dinámicas utilizando APIs externas en sus proyectos frontend.

Uso del DOM para mostrar datos en el HTML

El DOM (Document Object Model) es una representación estructurada de un documento HTML que nos permite interactuar y manipular los elementos de la

página web mediante JavaScript. Una vez que hemos obtenido los datos de una API mediante la Fetch API, podemos utilizar el DOM para mostrar esos datos de manera dinámica y atractiva en nuestro sitio web. A continuación, se presenta una explicación detallada junto con un ejemplo práctico:

- **Paso 1:** Obtener los datos de la API

Primero, utilizamos la Fetch API para realizar una solicitud a la API y obtener los datos que necesitamos. En este ejemplo, utilizaremos una API que proporciona información sobre libros.

```
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    // (este paso ya ha sido explicado en temas anteriores)
    console.log(data);
    // Continuar con el paso 2
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

- **Paso 2:** Crear una plantilla para mostrar los datos en el HTML

Antes de mostrar los datos en el HTML, es útil crear una plantilla que nos permita organizar y presentar los datos de manera coherente. En este ejemplo, utilizaremos una plantilla HTML para mostrar información sobre cada libro recibido desde la API.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Librería</title>
</head>
<body>
  <h1>Biblioteca de libros</h1>
  <div id="libros-container">
    <!-- Aquí se mostrarán los datos de los libros -->
  </div>
</body>
</html>
```

- **Paso 3:** Manipulación del DOM para mostrar los datos

Ahora, vamos a utilizar el DOM para agregar los datos recibidos de la API a la plantilla HTML que hemos creado. Utilizaremos el método createElement para crear elementos HTML y appendChild para agregarlos al contenedor de libros.

```
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    console.log(data);

    // Seleccionar el contenedor de libros en el HTML
    const librosContainer = document.getElementById('libros-container');

    // Iterar sobre cada libro en los datos recibidos
    data.forEach(libro => {
      // Crear un nuevo elemento div para representar cada libro
      const libroDiv = document.createElement('div');

      // Agregar contenido al div del libro utilizando los datos de la API
      libroDiv.innerHTML = `
        <h2>${libro.titulo}</h2>
        <p>Autor: ${libro.autor}</p>
        <p>Precio: $${libro.precio}</p>
      `;

      // Agregar el div del libro al contenedor de libros en el HTML
      librosContainer.appendChild(libroDiv);
    });
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Utilizando el DOM, hemos creado una página web dinámica que muestra información sobre libros obtenida desde una API. Cada libro se representa en un div individual que contiene su título, autor y precio. A medida que obtenemos nuevos datos de la API, el contenido del DOM se actualiza automáticamente, lo que nos permite crear aplicaciones interactivas y atractivas para nuestros usuarios. El DOM nos brinda la capacidad de manipular el contenido de nuestras páginas web en tiempo real, lo que es esencial para el desarrollo frontend y el consumo efectivo de APIs en JavaScript.

Crear vistas con tarjetas o tablas y realizar paginación

Una parte esencial del desarrollo frontend con JavaScript y APIs es la capacidad de mostrar datos de manera organizada y amigable para el usuario. En este tema, aprenderemos cómo crear vistas con tarjetas o tablas para presentar la información obtenida desde una API. Además, implementaremos la paginación para dividir los datos en páginas más pequeñas y facilitar la navegación del usuario.

- **Paso 1:** Obtener los datos de la API

Como en temas anteriores, utilizaremos la Fetch API para obtener los datos desde una API. En este ejemplo, continuaremos utilizando la API de libros para mostrar una lista de libros.

```
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    // (este paso ya ha sido explicado en temas anteriores)
    console.log(data);
    // Continuar con el paso 2
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

- **Paso 2:** Crear una vista con tarjetas

Para mostrar los datos en una vista con tarjetas, utilizaremos el DOM para crear elementos HTML y representar cada libro como una tarjeta.

```html
<!DOCTYPE html>
<html>
<head>
  <title>Librería</title>
</head>
<body>
  <h1>Biblioteca de libros</h1>
  <div id="libros-container">
    <!-- Aquí se mostrarán las tarjetas de los libros -->
  </div>
</body>
</html>
```

```javascript
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    console.log(data);

    // Seleccionar el contenedor de libros en el HTML
    const librosContainer = document.getElementById('libros-container');

    // Iterar sobre cada libro en los datos recibidos
    data.forEach(libro => {
      // Crear un nuevo elemento div para representar la tarjeta del libro
      const libroCard = document.createElement('div');
      libroCard.classList.add('card'); // Agregamos una clase para dar estilo a la tarjeta

      // Agregar contenido a la tarjeta utilizando los datos de la API
      libroCard.innerHTML = `
        <h2>${libro.titulo}</h2>
        <p>Autor: ${libro.autor}</p>
        <p>Precio: $${libro.precio}</p>
      `;

      // Agregar la tarjeta del libro al contenedor de libros en el HTML
      librosContainer.appendChild(libroCard);
    });
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

- **Paso 3:** Implementar la paginación

Ahora, vamos a implementar la paginación para dividir los datos en páginas más pequeñas y permitir al usuario navegar de manera más sencilla. En este ejemplo, mostraremos 5 libros por página.

```
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    console.log(data);

    // Seleccionar el contenedor de libros en el HTML
    const librosContainer = document.getElementById('libros-container');

    // Definir el número de libros a mostrar por página
    const librosPorPagina = 5;

    // Calcular el número total de páginas
    const totalPaginas = Math.ceil(data.length / librosPorPagina);

    // Función para mostrar libros en una página específica
    function mostrarLibros(pagina) {
      // Limpiar el contenedor antes de mostrar nuevos libros
      librosContainer.innerHTML = '';

      // Calcular el índice inicial y final de los libros a mostrar
      const indiceInicial = (pagina - 1) * librosPorPagina;
      const indiceFinal = indiceInicial + librosPorPagina;

      // Iterar sobre los libros en la página actual
      for (let i = indiceInicial; i < indiceFinal; i++) {
        if (data[i]) {
          const libro = data[i];
```

```
            // Crear un nuevo elemento div para representar la tarjeta del libro
            const libroCard = document.createElement('div');
            libroCard.classList.add('card'); // Agregamos una clase para dar estilo a la tarjeta

            // Agregar contenido a la tarjeta utilizando los datos de la API
            libroCard.innerHTML = `
              <h2>${libro.titulo}</h2>
              <p>Autor: ${libro.autor}</p>
              <p>Precio: $${libro.precio}</p>
            `;

            // Agregar la tarjeta del libro al contenedor de libros en el HTML
            librosContainer.appendChild(libroCard);
          }
        }
      }

      // Mostrar la primera página al cargar la página
      mostrarLibros(1);

      // Implementar la paginación con botones para navegar entre las páginas
      const paginacionContainer = document.createElement('div');
      for (let i = 1; i <= totalPaginas; i++) {
        const botonPagina = document.createElement('button');
        botonPagina.textContent = i;
        botonPagina.addEventListener('click', () => mostrarLibros(i));
        paginacionContainer.appendChild(botonPagina);
      }

      // Agregar la paginación al contenedor de libros en el HTML
      librosContainer.appendChild(paginacionContainer);
    })
```

```
    .catch(error => {
      console.error('Error:', error);
    });
```

Hemos creado una vista con tarjetas para mostrar información sobre libros obtenida desde una API. Además, hemos implementado la paginación para dividir los datos en páginas más pequeñas, lo que facilita la navegación del usuario. Con esta funcionalidad, nuestras aplicaciones frontend pueden mostrar grandes cantidades de datos de manera organizada y accesible, brindando una experiencia de usuario más amigable y satisfactoria. El uso combinado del DOM y las APIs nos permite crear aplicaciones web interactivas y dinámicas, enriqueciendo la experiencia del usuario y mejorando la funcionalidad de nuestras aplicaciones.

¿Qué es el localStorage y para qué sirve?

El localStorage es un objeto global en JavaScript que proporciona una forma simple de almacenar datos en el navegador web de manera persistente. A diferencia de las variables locales que se pierden cuando se cierra la página, el localStorage permite almacenar datos incluso después de que el navegador se haya cerrado y vuelto a abrir.

El localStorage es útil para almacenar datos que deben mantenerse entre sesiones, como configuraciones de usuario, datos de preferencias, carritos de compra, tokens de acceso y otra información que debe persistir en el navegador del usuario.

Beneficios del localStorage

- ➔ Persistencia: Los datos almacenados en el localStorage no se pierden cuando se cierra la página o se reinicia el navegador, lo que permite mantener información relevante a lo largo del tiempo.
- ➔ Fácil de usar: El localStorage proporciona una interfaz sencilla para almacenar y recuperar datos sin la necesidad de utilizar técnicas complejas.
- ➔ Capacidad de almacenamiento: El localStorage ofrece una mayor capacidad de almacenamiento en comparación con las cookies, lo que lo convierte en una opción adecuada para datos más grandes.
- ➔ Seguridad: Los datos almacenados en el localStorage son accesibles solo desde el dominio que los ha almacenado, lo que brinda cierta seguridad en la protección de la información del usuario.

Usos del localStorage

Vamos a utilizar el localStorage para crear un ejemplo práctico donde un usuario pueda almacenar sus preferencias de tema en una página web. El usuario podrá seleccionar un tema claro o oscuro y la preferencia se mantendrá incluso si cierra y vuelve a abrir el navegador.

```
fetch('https://api.example.com/books')
  .then(response => response.json())
  .then(data => {
    // Procesar los datos recibidos de la API
    console.log(data);
    // Seleccionar el contenedor de libros en el HTML
    const librosContainer = document.getElementById('libros-container');
    // Definir el número de libros a mostrar por página
    const librosPorPagina = 5;
    // Calcular el número total de páginas
    const totalPaginas = Math.ceil(data.length / librosPorPagina);
    // Función para mostrar libros en una página específica
    function mostrarLibros(pagina) {
      // Limpiar el contenedor antes de mostrar nuevos libros
      librosContainer.innerHTML = '';
      // Calcular el índice inicial y final de los libros a mostrar
      const indiceInicial = (pagina - 1) * librosPorPagina;
      const indiceFinal = indiceInicial + librosPorPagina;
      // Iterar sobre los libros en la página actual
      for (let i = indiceInicial; i < indiceFinal; i++) {
        if (data[i]) {
          const libro = data[i];

          // Crear un nuevo elemento div para representar la tarjeta del libro
          const libroCard = document.createElement('div');
          libroCard.classList.add('card'); // Agregamos una clase para dar estilo a la tarjeta

          // Agregar contenido a la tarjeta utilizando los datos de la API
          libroCard.innerHTML = `
            <h2>${libro.titulo}</h2>
            <p>Autor: ${libro.autor}</p>
            <p>Precio: $${libro.precio}</p>
          `;

          // Agregar la tarjeta del libro al contenedor de libros en el HTML
          librosContainer.appendChild(libroCard);
        }
      }
    }
```

```html
<!DOCTYPE html>
<html>
<head>
  <title>Preferencias de Tema</title>
</head>
<body>
  <h1>Preferencias de Tema</h1>
  <label>
    <input type="radio" name="tema" value="claro"> Tema Claro
  </label>
  <label>
    <input type="radio" name="tema" value="oscuro"> Tema Oscuro
  </label>
  <button onclick="guardarPreferencia()">Guardar Preferencia</button>

  <script>
    // Función para guardar la preferencia de tema en el localStorage
    function guardarPreferencia() {
      const temasRadio = document.getElementsByName('tema');
      let temaSeleccionado = '';
      temasRadio.forEach(radio => {
        if (radio.checked) {
          temaSeleccionado = radio.value;
        }
      });
      if (temaSeleccionado !== '') {
        localStorage.setItem('preferenciaTema', temaSeleccionado);
        alert('Preferencia de tema guardada exitosamente.');
      } else {
        alert('Por favor, selecciona un tema antes de guardar la preferencia.');
      }
    }
    // Función para cargar la preferencia de tema desde el localStorage
    function cargarPreferencia() {
      const preferenciaGuardada = localStorage.getItem('preferenciaTema');
      if (preferenciaGuardada) {
        const temasRadio = document.getElementsByName('tema');
        temasRadio.forEach(radio => {
          if (radio.value === preferenciaGuardada) {
            radio.checked = true;
          }
        });
      }
    }
    // Al cargar la página, cargamos la preferencia de tema desde el localStorage
    cargarPreferencia();
  </script>
</body>
</html>
```

El localStorage es una herramienta poderosa para el **almacenamiento persistente de datos en el navegador web.** Su uso puede mejorar significativamente la experiencia del usuario al mantener preferencias y configuraciones personalizadas. En este ejemplo, hemos utilizado el localStorage para permitir que el usuario seleccione su tema preferido y guardarlo para que se mantenga incluso después de cerrar el navegador. Esto es solo uno de los muchos usos posibles del localStorage en el desarrollo frontend.

¿Qué es Axios y para qué sirve?

Axios es una librería de JavaScript basada en promesas que facilita el manejo de solicitudes HTTP en el navegador y en Node.js. Es una herramienta ampliamente utilizada para realizar peticiones a APIs y servidores, ya que ofrece una sintaxis sencilla y concisa para trabajar con solicitudes y respuestas.

Axios proporciona métodos para enviar solicitudes HTTP, como GET, POST, PUT, DELETE, entre otros, y es especialmente útil para manejar respuestas en diferentes formatos, como JSON o XML. Además, es compatible con navegadores modernos y Node.js, lo que lo hace muy versátil para el desarrollo frontend y backend.

Beneficios de Axios

- ➔ Sintaxis clara y sencilla: Axios utiliza una interfaz basada en promesas que facilita la realización de solicitudes HTTP y el manejo de las respuestas.
- ➔ Soporte para navegadores y Node.js: Axios puede utilizarse tanto en el navegador como en el entorno de Node.js, lo que lo convierte en una opción versátil para proyectos que requieren trabajar en ambos lados.
- ➔ Manejo de errores mejorado: Axios proporciona un mecanismo eficiente para el manejo de errores en las solicitudes, lo que facilita la identificación y el tratamiento adecuado de problemas en la comunicación con la API o servidor.
- ➔ Intercepción de solicitudes y respuestas: Axios permite interceptar tanto las solicitudes como las respuestas antes de que sean enviadas o procesadas, lo que brinda flexibilidad para agregar lógica adicional o modificaciones a las peticiones.

Usos de Axios

A continuación, presentaremos un ejemplo práctico utilizando Axios para realizar una solicitud a una API pública y mostrar los resultados en el navegador.

Primero, asegúrate de incluir Axios en tu proyecto. Puedes hacerlo mediante un CDN o instalándolo con npm:

```
<!-- Usando CDN -->
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

Realizaremos una solicitud GET a una API que proporciona información sobre usuarios ficticios.

```
// Ejemplo de uso de Axios
axios.get('https://api.example.com/users')
  .then(response => {
    // Continuar con el paso 3
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Procesaremos los datos recibidos en la respuesta y los mostraremos en el HTML utilizando el DOM.

```
axios.get('https://api.example.com/users')
  .then(response => {
    // Procesar los datos recibidos de la API
    const usuarios = response.data;
    console.log(usuarios);

    // Seleccionar el contenedor de usuarios en el HTML
    const usuariosContainer = document.getElementById('usuarios-container');

    // Iterar sobre cada usuario en los datos recibidos
    usuarios.forEach(usuario => {
      // Crear un nuevo elemento div para representar la tarjeta del usuario
      const usuarioCard = document.createElement('div');
      usuarioCard.classList.add('card'); // Agregamos una clase para dar estilo a la tarjeta

      // Agregar contenido a la tarjeta utilizando los datos de la API
      usuarioCard.innerHTML = `
        <h2>${usuario.nombre}</h2>
        <p>Email: ${usuario.email}</p>
        <p>Edad: ${usuario.edad}</p>
      `;

      // Agregar la tarjeta del usuario al contenedor en el HTML
      usuariosContainer.appendChild(usuarioCard);
    });
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Axios es una herramienta poderosa para realizar solicitudes HTTP en JavaScript, ya que simplifica el proceso y proporciona una experiencia más fluida en el manejo de solicitudes y respuestas. En este ejemplo, hemos utilizado Axios para obtener datos desde una API pública y mostrarlos en una vista con tarjetas en el navegador. Esto es solo uno de los muchos usos posibles de Axios en el desarrollo frontend y backend. A medida que continúes explorando y aprendiendo, descubrirás cómo Axios puede ser una excelente opción para optimizar y mejorar tus aplicaciones web y consumir APIs de manera eficiente.

Estrategias para optimizar el consumo de APIs desde el cliente

Optimizar el consumo de APIs desde el cliente es esencial para mejorar la eficiencia de nuestras aplicaciones web y proporcionar una experiencia más rápida y fluida al usuario. Aquí exploraremos algunas estrategias que podemos implementar para lograr esta optimización:

1. Minimizar solicitudes:

Reducir el número de solicitudes realizadas a la API es una de las formas más efectivas de optimizar el consumo de APIs. Esto se puede lograr combinando varios endpoints en una sola solicitud o usando paginación para obtener solo la cantidad necesaria de datos.

- ➔ Beneficios: Menor carga en el servidor y tiempos de respuesta más rápidos, lo que se traduce en una experiencia más ágil para el usuario.

2. Cacheo de respuestas:

Almacenar temporalmente las respuestas de la API en caché en el cliente puede evitar la necesidad de hacer solicitudes repetidas para la misma información.

- ➔ Beneficios: Reducción significativa de las solicitudes al servidor y una carga más rápida de los datos almacenados en caché.

3. Cacheo de recursos:

Además del cacheo de respuestas de la API, también podemos aplicar técnicas de cacheo para almacenar en caché recursos estáticos, como imágenes, hojas de estilo y scripts. Esto se puede hacer mediante el uso de cabeceras de caché o mediante el almacenamiento en caché en el navegador.

- ➔ Beneficios: Tiempos de carga más rápidos y menor consumo de ancho de banda al reutilizar recursos ya almacenados en caché.

4. Gestión de errores y fallbacks:

Es importante implementar mecanismos para manejar errores de conexión o fallos en las solicitudes a la API. Podemos establecer fallbacks para cargar datos locales o mensajes de error en caso de que la conexión con la API falle.

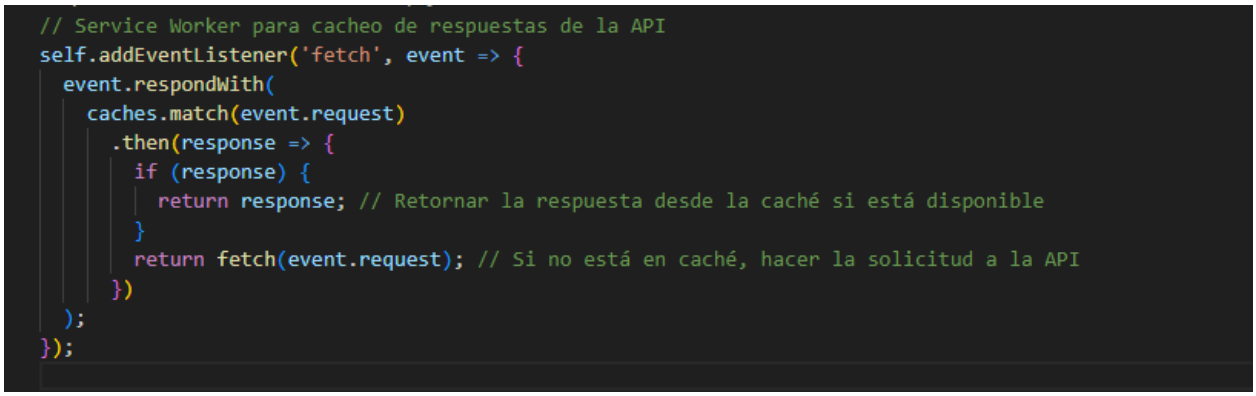
- ➔ Beneficios: Mejora de la confiabilidad y disponibilidad de la aplicación, incluso en situaciones adversas.

Uso de técnicas de caché

1. Cacheo de respuestas utilizando Service Workers:

Los Service Workers son scripts que se ejecutan en segundo plano y permiten interceptar y controlar las solicitudes realizadas desde el navegador. Podemos utilizar Service Workers para almacenar en caché respuestas de la API y recursos estáticos, incluso cuando la aplicación está fuera de línea.

Ejemplo:



```
// Service Worker para cacheo de respuestas de la API
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request)
      .then(response => {
        if (response) {
          return response; // Retornar la respuesta desde la caché si está disponible
        }
        return fetch(event.request); // Si no está en caché, hacer la solicitud a la API
      })
  );
});
```

- ➔ Beneficios: Mayor rapidez de carga de la aplicación y capacidad de funcionar sin conexión.

2. Almacenamiento en caché en el navegador:

Además del cacheo de respuestas con Service Workers, podemos utilizar la API de almacenamiento en caché del navegador para almacenar recursos estáticos, como imágenes y estilos.

Ejemplo:

```html
<!-- Establecer cabeceras de caché para almacenar en caché recursos estáticos en el navegador -->
<!DOCTYPE html>
<html>
<head>
  <title>Mi Aplicación</title>
  <link rel="stylesheet" href="styles.css">
  <script src="script.js"></script>
</head>
<body>
  <img src="logo.png" alt="Logo">
</body>
</html>
```

- ➔ Beneficios: Menor tiempo de carga de los recursos y reducción del consumo de ancho de banda.

Optimizar el consumo de APIs desde el cliente y aplicar técnicas de caché son prácticas esenciales para mejorar el rendimiento y la eficiencia de nuestras aplicaciones web. Al reducir las solicitudes, almacenar en caché respuestas y recursos, y gestionar adecuadamente los errores, podemos ofrecer una experiencia más rápida y fluida al usuario, reducir la carga en el servidor y mejorar la disponibilidad de la aplicación. El uso de Service Workers y el cacheo en el navegador son poderosas herramientas que nos permiten alcanzar estos objetivos y proporcionar una experiencia de usuario excepcional.