



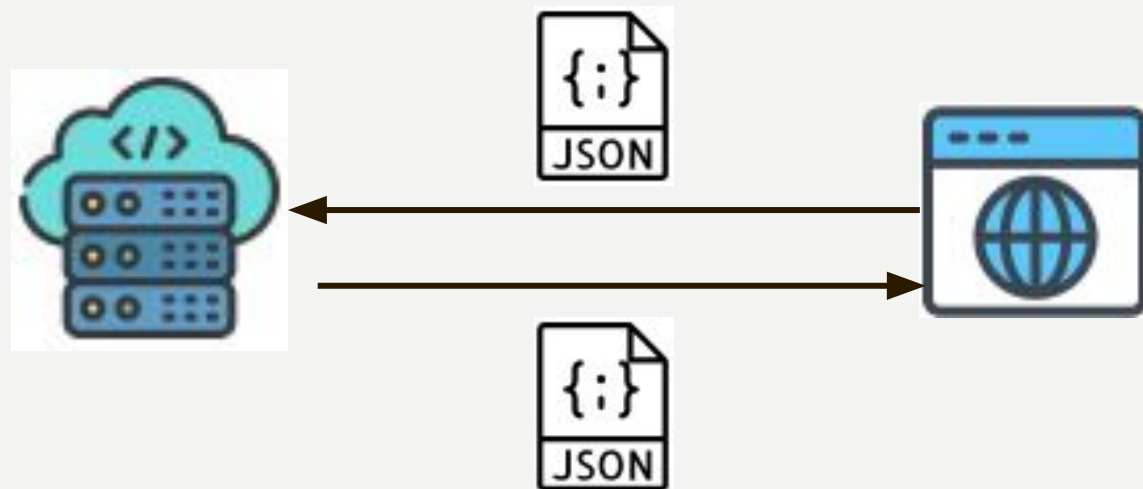
UNA HUR
UNIVERSIDAD NACIONAL
DE HURLINGHAM

Json

CÁTEDRA: CONSTRUCCIÓN DE INTERFACES DE USUARIO

¿Qué es JSON?

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos. Es fácil de leer y escribir para los humanos y fácil de analizar y generar para las máquinas. JSON se utiliza ampliamente para **enviar datos** entre un servidor y una aplicación web, así como para almacenar datos de configuración.



Características de JSON

Formato de Texto:

- JSON es una cadena de texto que representa datos estructurados.
- Los datos se organizan en pares de **clave-valor**.

Compatible con JavaScript:

- JSON es una subespecificación del objeto literal de JavaScript, lo que lo hace nativamente compatible con JavaScript.

Ligero y Fácil de Leer:

- La estructura de JSON es simple y legible tanto para humanos como para máquinas.

Sintaxis de JSON

La sintaxis de JSON se basa en dos estructuras principales:

- **Colección de pares clave-valor (Objeto):** Encerrada entre llaves {}.
 - clave: siempre entre comillas y es case Sensitive
 - valor: depende el tipo se escribe en el formato correspondiente
- **Lista ordenada de valores (Array):** Encerrada entre corchetes []

Ejemplo de Objeto JSON

```
{  
  "name": "Juan",  
  "age": 30,  
  "isStudent": false,  
  "address": {  
    "street": "Arias 576",  
    "city": "Castelar"  
  },  
  "courses": ["Matematica", "Naturales", "Historia"]  
}
```

Pares Clave-Valor: Cada par consiste en una clave (cadena) y un valor (puede ser una cadena, número, booleano, objeto, array o null).

Ejemplo de Array JSON

```
[  
  {  
    "name": "John",  
    "age": 30  
  },  
  {  
    "name": "Jane",  
    "age": 25  
  }  
]
```

Array de Objetos: Una lista de objetos JSON

Json y Objetos de JavaScript

Convertir un **Objeto** JavaScript a una **Cadena JSON**

Se utiliza **JSON.stringify()** para convertir un objeto JavaScript en una **cadena JSON**. Ejemplo:

```
const user = {  
  name: "Juan",  
  age: 30,  
  isStudent: false  
};
```

```
const jsonString = JSON.stringify(user); ->Devuelve Cadena Json  
console.log(jsonString);  
//{"name":"Juan","age":30,"isStudent":false}
```

Json y Objetos de JavaScript

```
const user = {  
  "name": "Juan",  
  "age": 30,  
  "isStudent": false,  
  saludar: function() {  
    console.log(`Hola soy ${name}`);  
  }  
};  
  
const jsonString = JSON.stringify(user);  
console.log(jsonString); -> ¿Que devuelve?
```

SOLO LAS PROPIEDADES, como dijimos Json es un formato para intercambio de **DATOS**

Json y Objetos de JavaScript

Convertir una **Cadena JSON** a un **Objeto** JavaScript

Se utiliza **JSON.parse()** para convertir una cadena JSON en un objeto JavaScript.

Ejemplo:

```
const jsonString = '{ "name": "Juan", "age": 30, "isStudent": false }';
```

```
const user = JSON.parse(jsonString); -> Devuelve OBJETO
```

Ahora podemos acceder a sus propiedades:

```
console.log(user.name);
```

Leer Json desde una Variable

```
JS app.js > ...  
let miPersonaJson = '{"name":"Juan","age":30,"isStudent":false}';
```

```
<> index.html > ...  
<!DOCTYPE html>  
<head>  
|   <meta charset="utf-8">  
|   <title>Practica Json</title>  
</head>  
<body>  
|   <div id="miDiv"></div>  
</body>  
<script src="app.js"></script>  
</html>
```

Quiero visualizar el nombre de
“miPersonaJson” en “miDiv”:

```
let miDiv=document.getElementById("miDiv");  
miDiv.textContent= xxxxxx
```

Leer Json desde una Variable

```
<> index.html > ...  
<!DOCTYPE html>  
<head>  
  <meta charset="utf-8">  
  <title>Practica Json</title>  
</head>  
<body>  
  <div id="miDiv"></div>  
</body>  
<script src="app.js"></script>  
</html>
```

```
JS app.js > ...  
let miPersonaJson = '{"name":"Juan","age":30,"isStudent":false}';  
let miDiv = document.getElementById("miDiv");  
  
//Convierto el json en un objeto js  
miPersonaObj = JSON.parse(miPersonaJson);  
//ahora puedo acceder a las propiedades del objeto  
miDiv.textContent= miPersonaObj.name;
```

Leer Json desde un Archivo

Ahora el json lo tenemos dentro del archivo “persona.json”, ya no está asignado a una variable:

```
{  
  "name": "Juan",  
  "age": 30,  
  "isStudent": false,  
  "address": {  
    "street": "Arias 576",  
    "city": "Castelar"  
  },  
  "courses": ["Matematica", "Naturales", "Historia"]  
}
```

Para accederlo debemos primero acceder al archivo, traernos los datos y guardarlos en una variable

Leer Json desde un Archivo

```
JS app.js > ...
let datosJson;
let xhr= new XMLHttpRequest();
xhr.open("GET","persona.json",true);
xhr.responseType="json";
xhr.onload = function (){
    if (xhr.status === 200){
        datosJson=xhr.response;
        let elementoTexto = document.getElementById("miDiv");
        elementoTexto.textContent = datosJson.name;
    } else{
        // manejo errores
    }
}
xhr.send();
```

Una forma es usando XMLHttpRequest, para lo cual se definen los siguientes métodos y propiedades:

- open()
- responseType
- onload
- send()

open(): lleva 3 parámetros:

- método: ponemos “GET” para indicar que queremos traernos un json
- url: la ruta del Archivo json
- async: se indica true si esa búsqueda se realizará de manera async, sino false

Leer Json desde un Archivo

```
JS app.js > ...
let datosJson;
let xhr= new XMLHttpRequest();
xhr.open("GET","persona.json",true);
xhr.responseType="json";
xhr.onload = function (){
    if (xhr.status === 200){
        datosJson=xhr.response;
        let elementoTexto = document.getElementById("miDiv");
        elementoTexto.textContent = datosJson.name;
    } else{
        // manejo errores
    }
}
xhr.send();
```

ResponseType: indica el tipo de dato que viene en la respuesta. Cuando le indicamos que la respuesta viene en formato “json”, ya nos realiza el parse para que podamos trabajarlo como objeto. Sino por default asume todo como “text”

onload: se asigna el evento que se queremos que se desencadene cuando la solicitud haya sido completada exitosamente y los datos están listos para ser usados. Le asignamos una función se ejecutará cada vez que se complete una solicitud. Dentro de la función primero validamos que el estado de la solicitud ha sido exitosa (status=200) y luego las instrucciones que queremos ejecutar.

Leer Json desde un Archivo

```
JS app.js > ...
let datosJson;
let xhr= new XMLHttpRequest();
xhr.open("GET","persona.json",true);
xhr.responseType="json";
xhr.onload = function (){
    if (xhr.status === 200){
        datosJson=xhr.response;
        let elementoTexto = document.getElementById("miDiv");
        elementoTexto.textContent = datosJson.name;
    } else{
        // manejo errores
    }
}
xhr.send();
```

send(): se indica que se desea enviar la solicitud

Cada vez que se invoque el send -> se realizará el envío de la solicitud(get)
Y cada vez que se complete la solicitud -> se ejecutará la función asignada al “onload”



UNA HUR
UNIVERSIDAD NACIONAL
DE HURLINGHAM

Fetch

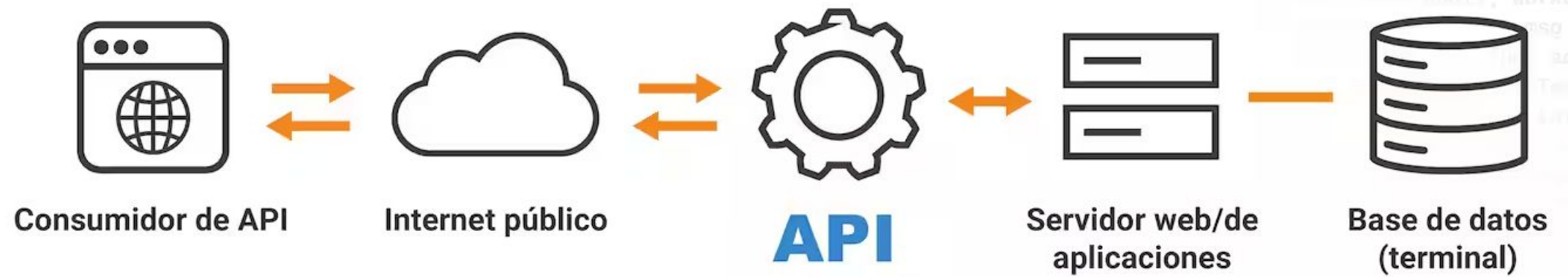
CÁTEDRA: CONSTRUCCIÓN DE INTERFACES DE USUARIO

API

Una API (Application Programming Interface) es un conjunto de reglas y definiciones que permite a las aplicaciones comunicarse entre sí.

Las APIs definen cómo se deben hacer las solicitudes y respuestas entre los distintos componentes de software, permitiendo que se intercambien datos y servicios de manera eficiente y segura.

API



¿Qué es fetch?

fetch es una función nativa de JavaScript para realizar solicitudes HTTP.

Permite hacer solicitudes GET, POST, PUT, DELETE, etc.

fetch se compone de dos partes principales:

- enviar la **solicitud** http
- devuelve una **Promesa** que se resuelve con la respuesta del servidor

Cuando se trabaja con fetch para realizar solicitudes HTTP, es común enviar y recibir datos en formato JSON.

Comparando código...

```
JS app.js > ...
let datosJson;
let xhr= new XMLHttpRequest();
xhr.open("GET","persona.json",true);
xhr.responseType="json";
xhr.onload = function (){
  if (xhr.status === 200){
    datosJson=xhr.response;
    let elementoTexto = document.getElementById("miDiv");
    elementoTexto.textContent = datosJson.name;
  } else{
    // manejo errores
  }
}
xhr.send();
```

con fetch

con XMLHttpRequest

```
JS app.js > ...
let datosJson;
fetch("persona.json")
  .then (res => res.json())
  .then((salida) => {
    datosJson=salida;
    let elementoTexto = document.getElementById("miDiv");
    elementoTexto.textContent = datosJson.name;
  })
```

Envío de solicitud

Antes teníamos que indicar el método Get y asincronía en true

```
JS app.js > ...  
let datosJson;  
let xhr= new XMLHttpRequest();  
xhr.open("GET","persona.json",true);  
xhr.responseType="json";  
xhr.onload = function (){
```

Pero ahora esos dos valores vienen configurados por default. Solo se debe indicar la url:

```
JS app.js > ...  
let datosJson;  
fetch("persona.json")  
  .then(res => res.json())  
  .then((salida) => {  
    datosJson=salida;  
    let elementoTexto = document.getElementById("miDiv");  
    elementoTexto.textContent = datosJson.name;  
  })
```

***Promesas

El fetch devuelve una promesa (acción asincrónica), y le debemos indicar qué hacer cuando esa promesa se cumpla o se rechace.

```
> console.log(fetch("persona.json"))
```

```
▼ Promise {<pending>} i
  ► [[Prototype]]: Promise
    [[PromiseState]]: "fulfilled"
    ► [[PromiseResult]]: Response
```

```
JS app.js > ...
let datosJson;
fetch("persona.json")
  .then (res => res.json())
  .then((salida) => {
    datosJson=salida;
    let elementoTexto = document.getElementById("miDiv");
    elementoTexto.textContent = datosJson.name;
  })
```

.then define la función que se va a ejecutar cuando la promesa **finalice exitosamente**.

El segundo **.then** que aparece es otra promesa que devuelve el método “json()”, también se ejecuta asíncrono porque puede demorar en convertir mucha información

Devolución de Promesa

```
JS app.js > ...  
let datosJson;  
fetch("persona.json")  
  .then (res => res.json())  
  .then((salida) => {  
    datosJson=salida;  
    let elementoTexto = document.getElementById("miDiv");  
    elementoTexto.textContent = datosJson.name;  
  })
```

El primer **.then** define la función que se va a ejecutar cuando la promesa **fetch** finalice correctamente.

Se define una función anónima que :

- Recibe como parámetro el **valor de la promesa resuelta**, el cual guardamos en “res”.
- Solicita convertir el valor recibido (que debemos saber que viene como JSON) a un objeto javascript para que podamos manipularlo.

Devolución de Promesa

```
JS app.js > ...
let datosJson;
fetch("persona.json")
  .then(res => res.json())
  .then((salida) => {
    datosJson=salida;
    let elementoTexto = document.getElementById("miDiv");
    elementoTexto.textContent = datosJson.name;
  })
```

El segundo **.then** define la función que se va a ejecutar cuando la promesa del **json()** finalice correctamente.

Se define una función anónima que:

- Recibe como parámetro el **json parseado (convertido a Objeto)**, el cual guardamos en “salida”.
- Realiza las acciones requeridas: en este caso mostrar el contenido de “name” obtenido del json en el div

Ejemplo de una Solicitud GET

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la respuesta del servidor. Ver detalle:' +
        response.statusText);
    }

    return response.json(); // Parsear la respuesta como JSON
  })
  .then(data => {
    console.log(data); // Manejar los datos recibidos
  })
  .catch(error => {
    console.error('Error Inesperado:', error);
  });
```

Ejemplo de una Solicitud GET

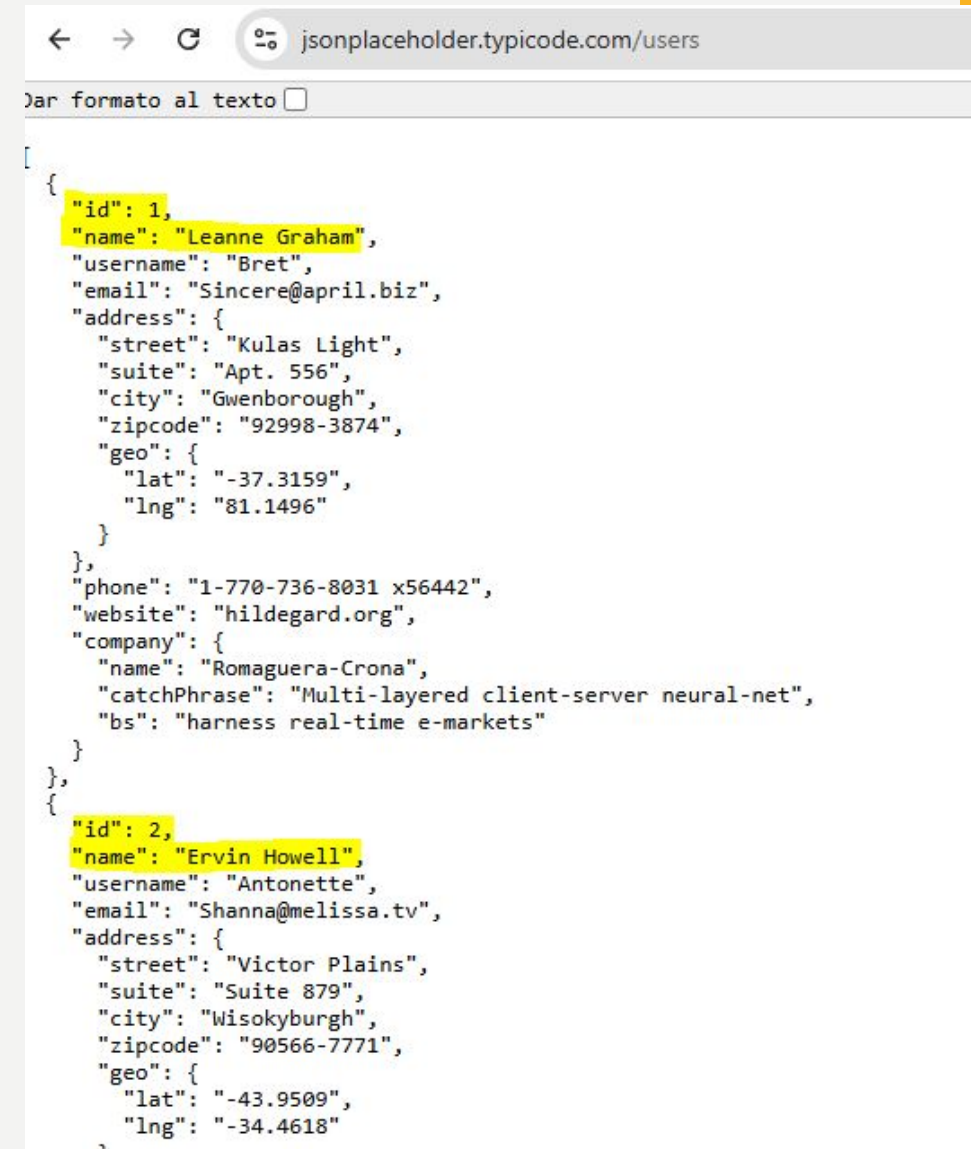
En este ejemplo:

- **fetch('https://api.example.com/data')** realiza una solicitud GET a la URL especificada.
- **response.ok** verifica si la respuesta fue exitosa (estado HTTP 200-299).
- **response.json()** convierte la respuesta en un objeto JavaScript.
- El primer **then** maneja la conversión de la respuesta.
- El segundo **then** maneja los datos recibidos.
- **catch** captura y maneja cualquier error que ocurra durante la solicitud.

Ejemplo (API pública)

Probemos el código anterior pero realizando el fetch a esta url que es una api pública que devuelve un json con datos de usuarios:

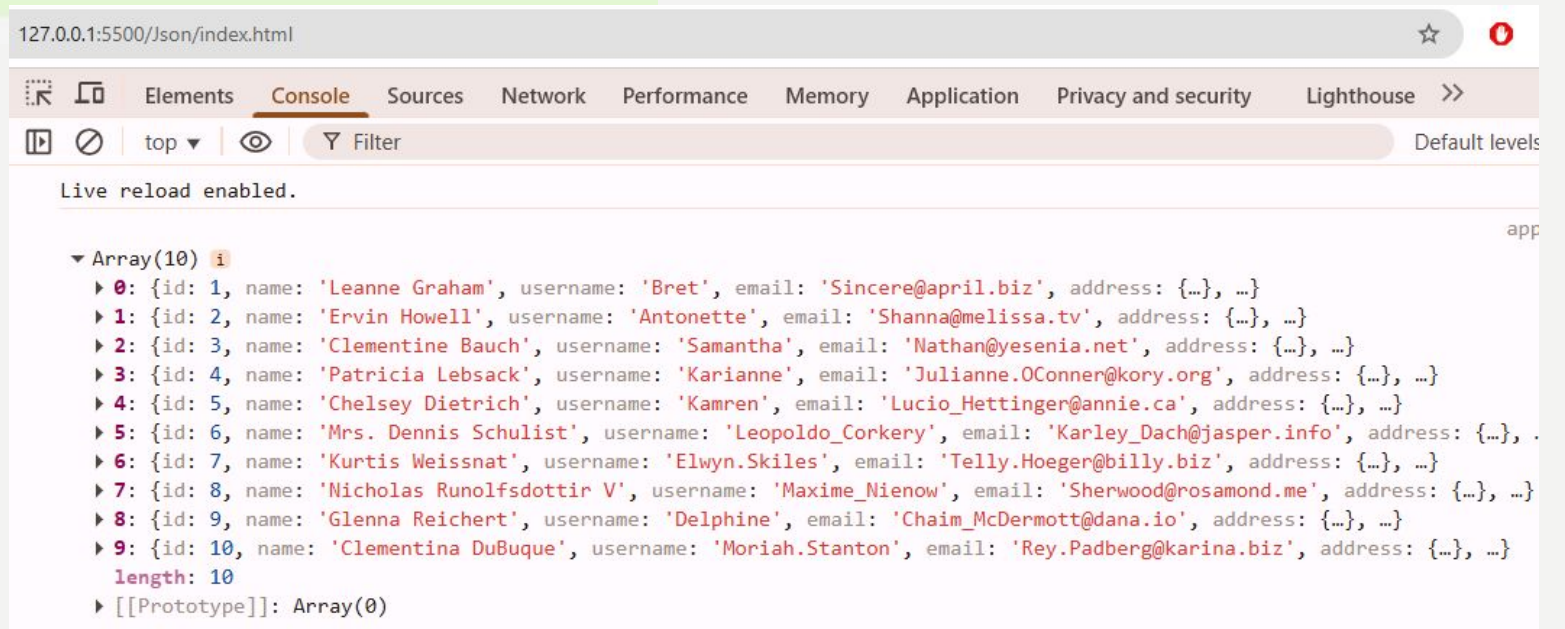
<https://jsonplaceholder.typicode.com/users>



```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "username": "Antonette",
    "email": "Shanna@melissa.tv",
    "address": {
      "street": "Victor Plains",
      "suite": "Suite 879",
      "city": "Wisokyburgh",
      "zipcode": "90566-7771",
      "geo": {
        "lat": "-43.9509",
        "lng": "-34.4618"
      }
    }
  }
]
```

Ejemplo (API pública)

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok ' + response.statusText);
    }
    return response.json(); // Parsear la respuesta como JSON
  })
  .then(data => {
    console.log(data); // Manejar los datos recibidos
  })
  .catch(error => {
    console.error('There was a problem with the fetch operation:', error);
  });
```



Ejemplo (API pública)

Url original: <https://jsonplaceholder.typicode.com/users>

Y Ahora probemos modificar user por usuarios, le estamos diciendo que acceda

<https://jsonplaceholder.typicode.com/usuarios>

Y Ahora probemos modificar a una URL inexistente:

<https://jsonholder.typicode.com/users>

Usando async y await

```
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error('Error en la respuesta del servidor. Ver
        detalle:' + response.statusText);
    }
    return response.json(); // Parsear la respuesta como JSON
  })
  .then(data => {
    console.log(data); // Manejar los datos recibidos
  })
  .catch(error => {
    console.error('Error Inesperado:', error);
  });
```

Usando async y await

```
async function obtenerUsuarios() {  
  try {  
    const response = await fetch('https://jsonplaceholder.typicode.com/users');  
    if (!response.ok) {  
      throw new Error('Error en la respuesta del servidor. Ver detalle: ' +  
        response.status);  
    }  
    const data = await response.json();  
    console.log(data); // Manejar los datos recibidos  
  } catch (error) {  
    console.error('Error inesperado:', error);  
  }  
}
```

Resumen

JSON: Formato de texto para representar datos estructurados, usado para intercambiar información entre aplicaciones.

fetch: Función moderna de JavaScript para hacer solicitudes HTTP y obtener datos de servidores. Puede manejar cualquier tipo de datos que pueda ser representado como una cadena de texto o binario, aunque el más usado es el json

Promesa: Objeto que representa una operación asíncronica que puede completarse con éxito o fallar en el futuro.

Resumen

Solicitud externa a un servidor:

`fetch("url externa")`

`.then` -> Si el servidor responde -> la promesa entra al `.then`

`.catch` -> Si el servidor NO responde -> la promesa entra al `.catch`

Respuestas del servidor:

`response.ok`:

- Devuelve **true** -> si el código de estado HTTP está entre 200 y 299
- Devuelve **false** -> si es 4xx o 5xx

En este caso el servidor es el que responde, pero puede respondernos que la solicitud que realizamos fue exitosa o que no pudo realizarse