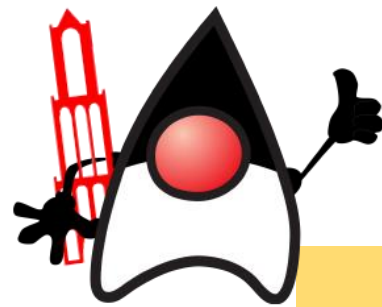
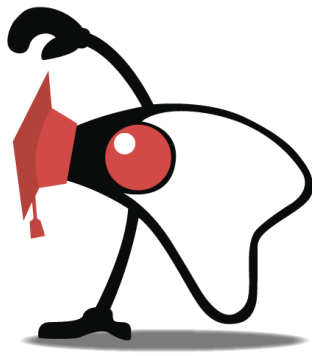




Curso FullStack Python

Codo a Codo 4.0



VUE.js

Parte 3

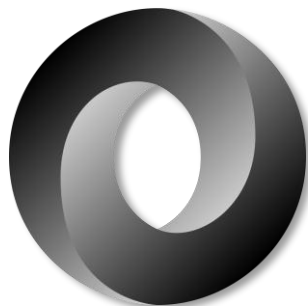


JSON: JavaScript Object Notation

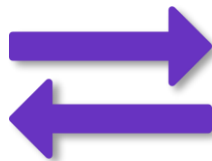
JSON es una sintaxis propia de objetos tipo JavaScript utilizada para **almacenar e intercambiar** datos. Es texto, escrito con notación de objetos JavaScript, con un formato determinado.

Intercambio de datos

Al intercambiar datos entre un navegador y un servidor, los datos **solo pueden ser texto**. Dado que JSON trabaja como texto podemos convertir cualquier objeto JavaScript en JSON y enviar JSON al servidor. También podemos convertir cualquier JSON recibido del servidor en objetos JavaScript. De esta forma podemos trabajar con los datos como objetos JavaScript, sin complicados análisis ni traducciones.



JSON



*Cualquier JSON podrá ser
convertido a JS y cualquier texto
con el formato correspondiente
podrá ser convertido a JSON*



JSON: JavaScript Object Notation

Convirtiendo de JavaScript a JSON:

Si tiene datos almacenados en un objeto JavaScript, puede convertir el objeto en JSON con ***JSON.stringify()***:

```
var myObj = { name: "John", age: 31, city: "New York" };  
var myJSON = JSON.stringify(myObj);  
// myJson= {"name":"John","age":31,"city":"New York"}
```

JS

Más info [aquí](#)

*Ver ejemplo
JstoJSON.html*

Convirtiendo de JSON a JavaScript:

Si tiene datos almacenados en un JSON, puede convertir el objeto en JavaScript con ***JSON.parse()***:

```
var myObj1=JSON.parse(myJSON);  
//myObj1= { name: "John", age: 31, city: "New York" }
```

JS

Más info [aquí](#)

*Ver ejemplo
JSONtoJS.html*

JSON: JavaScript Object Notation

Reglas de sintaxis JSON:

La sintaxis JSON se deriva de la sintaxis de notación de objetos de JavaScript:

- Los datos están en pares de nombre / valor
- Los datos están separados por comas
- Las {} contienen objetos
- Los corchetes contienen Array

```
myJson= {"name":"John","age":31,"city":"New York"}
```

JS

En JSON , los valores deben ser uno de los siguientes tipos de datos:

- string
- number
- object (JSON object)
- array
- boolean
- null

El tipo de archivo de los archivos JSON es ".json"



JSON: JavaScript Object Notation

JSON

```
// Ejemplo de JSON:
{
  "empleados": [
    { "nombre": "Juan", "apellido": "Pérez" },
    { "nombre": "Ana", "apellido": "López" },
    { "nombre": "Pedro", "apellido": "Fernández" }
  ]
}

// Otro ejemplo:
{
  "nombre": "Carlos",
  "apellido": "Rivarola",
  "segundoNombre": null,
  "edad": 30,
  "hijos": ["Ana", "Luisa", "Marcelo"]
}
```

1er ejemplo: En la propiedad “empleados” hay un array de 3 elementos y dentro de cada uno tengo objetos separados por comas. Cada objeto tiene un nombre y un apellido.

2do ejemplo: Vemos que el objeto JSON tiene un primer nombre asociado, junto con otras propiedades. Además hay una propiedad hijos que a su vez tiene un array.

Ver ejemplos [ejemplo.json](#) y [ejemplo2.json](#)

JSON: JavaScript Object Notation

Otro ejemplo JSON: Ingresar aquí: <https://mdn.github.io/learning-area/javascript/ojs/json/superheroes.json>

*Ver ejemplo
superheroes.json*

```
{
  "squadName" : "Super Hero Squad",
  "homeTown" : "Metro City",
  "formed" : 2016,
  "secretBase" : "Super tower",
  "active" : true,
  "members" : [
    {
      "name" : "Molecule Man",
      "age" : 29,
      "secretIdentity" : "Dan Jukes",
      "powers" : [
        "Radiation resistance",
        "Turning tiny",
        "Radiation blast"
      ]
    },
    {
      "name" : "Madame Uppercut",
      "age" : 39,
      "secretIdentity" : "Jane Wilson",
      "powers" : [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    }
  ]
}
```

Google Chrome



```
squadName:      "Super Hero Squad"
homeTown:        "Metro City"
formed:          2016
secretBase:      "Super tower"
active:          true
▼ members:
  ▼ 0:
    name:         "Molecule Man"
    age:          29
    secretIdentity: "Dan Jukes"
    ▼ powers:
      0:           "Radiation resistance"
      1:           "Turning tiny"
      2:           "Radiation blast"
  ▼ 1:
    name:         "Madame Uppercut"
    age:          39
    secretIdentity: "Jane Wilson"
    ▼ powers:
      0:           "Million tonne punch"
      1:           "Damage resistance"
      2:           "Superhuman reflexes"
```

Firefox Developer



Otro ejemplo JSON: <https://github.com/midesweb/taller-angular/blob/master/11-mi-API/peliculas.json>

*Ver ejemplo
películas.json*

JSON: JavaScript Object Notation

API pública Randomuser: <https://randomuser.me/api>

Muestra datos de usuarios aleatorios, se utiliza para hacer pruebas. Es un string de JSON con un formato particular. Devuelve un usuario aleatorio, un array con un solo elemento.

Conviene leerlo desde **Firefox Developer Edition**, ya que la visualización es más simple.

Nosotros podremos **consumir la API**, esto quiere decir leerla y traerla a nuestra aplicación. Ingresando en <https://randomuser.me/api/?results=5> podremos obtener 5 resultados, por ejemplo.

Peticiones HTTP



Un **navegador**, durante la carga de una página, suele realizar múltiples **peticiones HTTP** a un servidor para solicitar los archivos que necesita renderizar en la página. Es el caso de, en primer lugar, el documento **.html** de la página (*donde se hace referencia a múltiples archivos*) y luego todos esos archivos relacionados: los ficheros de estilos **.css**, las imágenes **.jpg**, **.png** u otras, los scripts **.js**, las tipografías **.ttf**, **.woff** o **.woff2**, etc.

Una **petición HTTP** es como suele denominarse a la acción por parte del navegador de solicitar a un servidor web un documento o archivo, ya sea un fichero **.html**, una imagen, una tipografía, un archivo **.js**, etc. Gracias a dicha petición, el navegador puede descargar ese archivo, almacenarlo en un **caché temporal de archivos del navegador** y, finalmente, mostrarlo en la página actual que lo ha solicitado. HTTP define una gran cantidad de métodos que son utilizados para diferentes circunstancias:

- **GET:** permite **consultar información** al servidor, muy parecido a realizar un SELECT a la base de datos.
- **POST:** solicita la **creación de un nuevo registro**, es decir, algo que no existía previamente, es equivalente a realizar un INSERT en una base de datos.
- **PUT:** permite **actualizar por completo un registro existente**, parecido a realizar un UPDATE en la base de datos.
- **PATCH:** similar al método PUT, pero se utiliza cuando **es necesario actualizar solo un fragmento del registro** y no en su totalidad, es equivalente a realizar un UPDATE a la base de datos.
- **DELETE:** permite **eliminar un registro existente**, es similar a DELETE en la base de datos.
- **HEAD:** es utilizado para **obtener información sobre un determinado recurso** sin retornar el registro. Este método se utiliza a menudo para probar la validez de los enlaces de hipertexto, la accesibilidad y las modificaciones recientes.

Fetch

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas.

También provee un método global ***fetch()*** que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Más información sobre Fetch:

https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Utilizando_Fetch

Una petición básica de fetch es realmente simple de realizar:

```
fetch('https://api.coindesk.com/v1/bpi/currentprice.json')  
  .then(response => response.json())  
  .then(data => console.log(data));
```

JS

Aquí estamos recuperando un archivo JSON a través de red y mostrando en la consola. El uso de ***fetch()*** más simple toma un argumento (la ruta del recurso que quieres buscar) 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()***.

Fetch

Con Fetch, así como podemos leer información que proviene de una API externa vamos a poder leer un archivo de texto y mostrarlo por consola.

En el siguiente ejemplo utilizamos clases de Bootstrap y dentro del **body** incorporaremos dos etiquetas **divs**: la primera para el título y el botón que me permitirá traer el contenido y la segunda para el contenido en cuestión:

```
<div class="container my-5 text-center">  
  <h1>Ejemplo Fetch</h1>  
  <button class="btn-danger w-100" onclick="traer()">Obtener</button>  
</div>  
<div class="mt-5" id="contenido">  
  <!-- Insertaremos contenido del archivo de texto para utilizar Fetch -->  
</div>
```

HTML

Ejemplo Fetch

Obtener

Importante: Este ejemplo solamente funcionará con LiveServer de VSC

[Ver ejemplo fetch.html](#)

Fetch

Crearemos un script dentro del mismo HTML con el siguiente código:

```
<script>
  var contenido = document.querySelector('#contenido');
  function traer() {
    fetch('texto.txt')
      .then(data => data.text())
      .then(data => {
        console.log(data)
        contenido.innerHTML= `${data}`
      })
  }
</script>
```

JS

Fetch me va a permitir hacer "promesas":

Se denominan promesas porque puede que no se ejecuten, porque por ejemplo si del otro lado no tengo nada el .then no va a ocurrir.

Explicación:

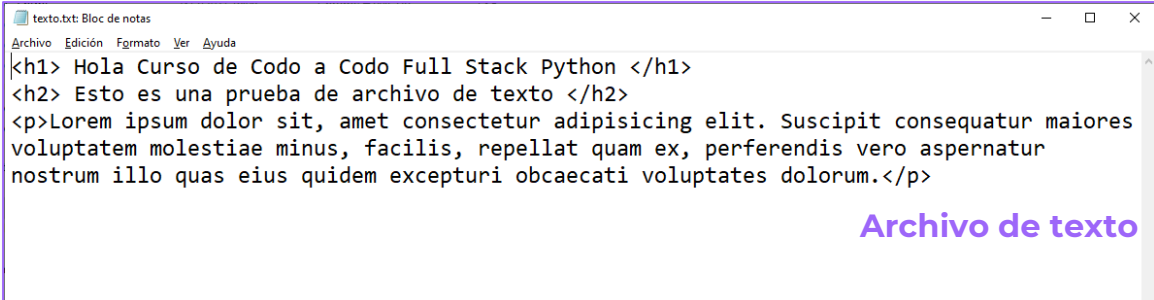
La variable contenido me permite traer el elemento con el ID #contenido.

La función traer() me permite obtener los datos:

- Con **fetch('texto.txt')** hago referencia al archivo que quiero traer.
- Con **.then** establezco que la variable data va a guardar el contenido traído por fetch a través del método .text()
- Con el siguiente **.then** muestro por consola y agrego a mi HTML el contenido de data (traído como **template string**)

Fetch

El resultado final será el siguiente:

A screenshot of a text editor window titled 'texto.txt: Bloc de notas'. The editor contains the following HTML code:

```
<h1> Hola Curso de Codo a Codo Full Stack Python </h1>
<h2> Esto es una prueba de archivo de texto </h2>
<p>Lorem ipsum dolor sit, amet consectetur adipisicing elit. Suscipit consequatur maiores voluptatem molestiae minus, facilis, repellat quam ex, perferendis vero aspernatur nostrum illo quas eius quidem excepturi obcaecati voluptates dolorum.</p>
```

Archivo de texto

Ejemplo Fetch

Ejemplo final

Obtener

Hola Curso de Codo a Codo Full Stack Python

Esto es una prueba de archivo de texto

Lorem ipsum dolor sit, amet consectetur adipisicing elit. Suscipit consequatur maiores voluptatem molestiae minus, facilis, repellat quam ex, perferendis vero aspernatur nostrum illo quas eius quidem excepturi obcaecati voluptates dolorum.

[Ver ejemplo fetch.html](#)

Fetch: consumir API externa

En este caso consumiremos la API <https://randomuser.me/api> que nos permitirá traer datos escritos en formato tipo JSON:

JS

```
function traer_dos() {  
  //fetch('texto.txt')  
  fetch('https://randomuser.me/api')  
    .then(res => res.json())  
    .then(res => {  
      console.log(res)  
      console.log(res.results[0].email)  
      // contenido.innerHTML= `${res.results[0].email}`  
      contenido.innerHTML= `  
          
        <p>Nombre: ${res.results[0].name.first}</p>  
        <p>Mail: ${res.results[0].email}</p>  
      `;  
    })  
}
```

Explicación:

La función traer_dos() me permite obtener los datos, pero esta vez desde una API externa:

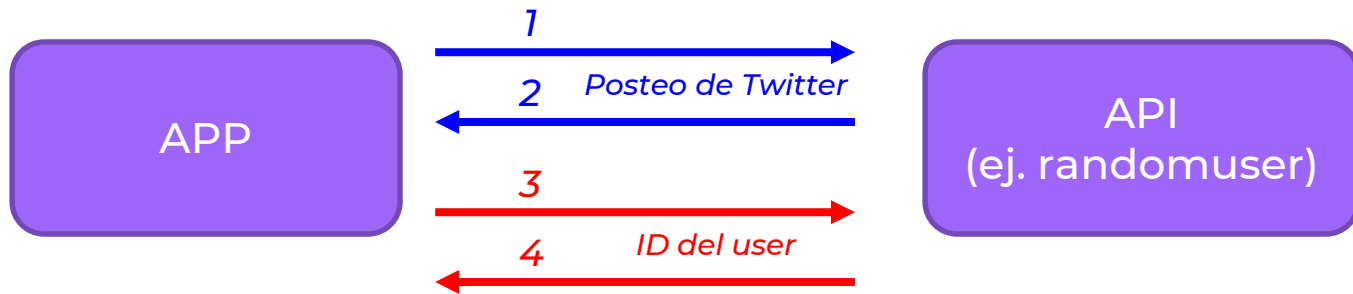
- Con **fetch('https://')** hago referencia a la API externa.
- En **.then** establezco que la variable res va a guardar el dato desde el método json(), que extrae el contenido del archivo **.json**
- Con el siguiente .then muestro por consola todos los resultados y agrego a mi HTML de los resultados en la posición 0 la imagen, el nombre y el mail.

Ver ejemplo fetch.html + inspeccionar para ver los cambios

Ver carpeta Fetch-API-master

Otro ejemplo de uso de Fetch con API externa

Este ejemplo es más completo. Consultará con una Api más de una vez, enviará un requerimiento, nos devolverá el dato y en base a esa respuesta voy a volver a enviar esa solicitud.



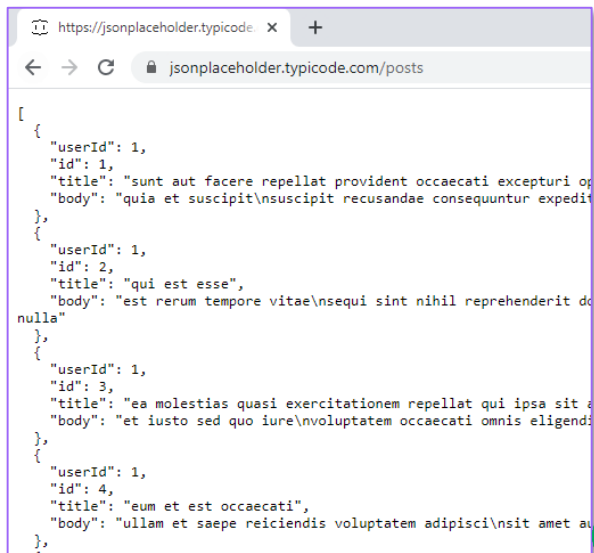
Desde la APP (1) hacemos una solicitud, vamos a traer información de la API externa. Esa API me devuelve información (2). Luego, en función de la respuesta que nos dio la API vamos a hacer otra solicitud (3) y esperar la respuesta (4). Lo que está en **rojo** dependerá de que ocurra lo que está en **azul**. La **segunda** respuesta depende de la **primera**.

Por ejemplo: podríamos pedir información de un posteo de Twitter, que lo devuelva pero luego pedir información del usuario que hizo ese posteo, para lo cual pido el ID de referencia del usuario donde podré ver sus datos.

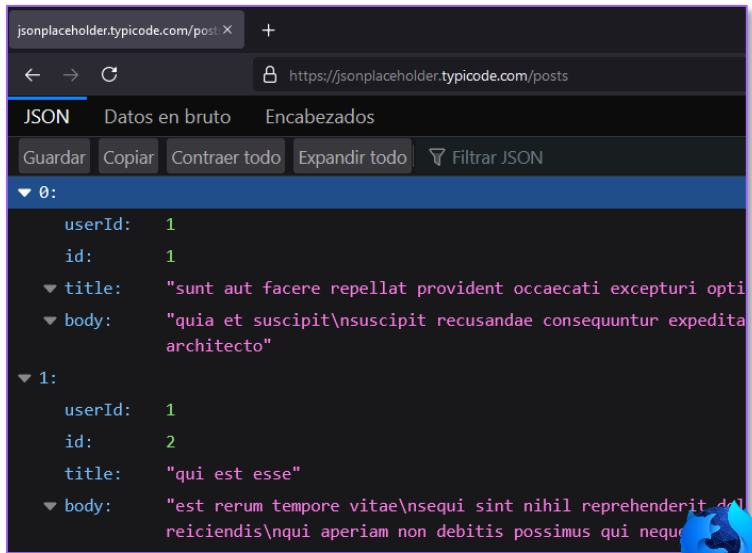
[Ver ejemplo fetch-then \(.html y .js\)](#)

Otro ejemplo de uso de Fetch con API externa

Para este ejemplo aprovecharemos la estructura de tipo JSON que nos ofrece <https://jsonplaceholder.typicode.com/posts>



```
[
  {
    "userId": 1,
    "id": 1,
    "title": "sunt aut facere repellat provident occaecati excepturi opti",
    "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita"
  },
  {
    "userId": 1,
    "id": 2,
    "title": "qui est esse",
    "body": "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor"
  },
  {
    "userId": 1,
    "id": 3,
    "title": "ea molestias quasi exercitationem repellat qui ipsa sit",
    "body": "et iusto sed quo iure\nvoluptatem occaecati omnis eligendi"
  },
  {
    "userId": 1,
    "id": 4,
    "title": "eum et est occaecati",
    "body": "ullam et saepe reiciendis voluptatem adipisci\nsit amet autem"
  }
]
```



JSON	Datos en bruto	Encabezados
Guardar	Copiar	Contraer todo
Expandir todo	Filtrar JSON	

```
▼ 0:
  userId: 1
  id: 1
  title: "sunt aut facere repellat provident occaecati excepturi opti"
  body: "quia et suscipit\nsuscipit recusandae consequuntur expedita"
  architecto"
▼ 1:
  userId: 1
  id: 2
  title: "qui est esse"
  body: "est rerum tempore vitae\nsequi sint nihil reprehenderit dolor"
  reiciendis\nqui aperiam non debitis possimus qui neque"
  suscipit
```



Estas APIs públicas contienen documentación donde te explican cómo utilizarla.

Para acceder a un posteo determinado debemos agregar el número de posteo, ejemplo: <https://jsonplaceholder.typicode.com/posts/2>, me devolverá un objeto que se corresponde con ese post.

Otro ejemplo de uso de Fetch con API externa

Vamos a obtener el ID de usuario de un determinado posteo. Para ello en nuestros archivos HTML y JS haremos lo siguiente:

```
<body>
  <h1>Ejemplo Fetch then</h1>
  <script src="fetch-then.js"></script>
</body>
```

HTML

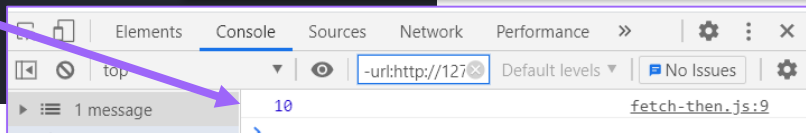
En este caso haremos un Fetch de un id de posteo que pasamos a la función por parámetro (99). Nos retorna ese objeto JSON y pedimos el id de usuario...

```
const getNombre= (idPost) => {
  // hacemos la solicitud a la API...
  fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
  // la API responde en formato JSON
  .then(res=> {
    return res.json()
  })
  // Pedimos el userID de ese posteo
  .then(post => {
    console.log(post.userId)
  })
}

getNombre(99); // llamada a la función
```

JS

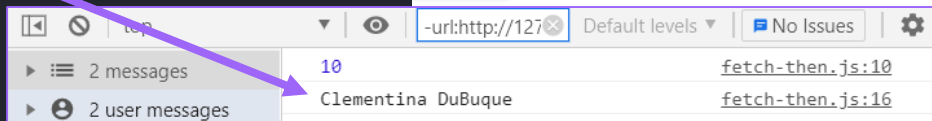
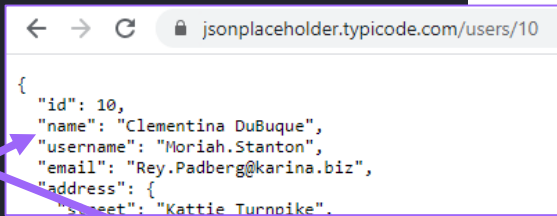
```
{
  "userId": 10,
  "id": 99,
  "title": "temporibus sit alias delectus eligendi possimus magni",
  "body": "quo deleniti praesentium dicta non quod naut est molestias"
}
```



Otro ejemplo de uso de Fetch con API externa

```
const getNombre = (idPost) => {  
  // hacemos la solicitud a la API...  
  fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)  
    // la API responde en formato JSON  
    .then(res => {  
      return res.json()  
    })  
    // Pedimos el userID de ese posteo  
    .then(post => {  
      console.log(post.userId)  
      fetch(`https://jsonplaceholder.typicode.com/users/${post.  
userId}`)  
        .then(res => {  
          return res.json()  
        })  
        .then(user => {  
          console.log(user.name)  
        })  
    })  
  }  
}  
getNombre(99); // llamada a la función
```

Pero en realidad lo que deseamos es mostrar la información del usuario que hizo ese post. En el post tengo solamente el id de usuario. La información del usuario podremos ir a consultarla al JSON que contiene los usuarios. <https://jsonplaceholder.typicode.com/users>



Fetch Async Await y manejo de errores

Uno de los inconvenientes que tenemos con este tipo de comunicaciones es que si no recibimos respuesta de la API me va a dar un error y falla al momento de hacer el Fetch y el resto del código no va a ejecutarse.

Este problema se genera porque la comunicación **no es asíncronica**. Para resolver esto vamos a crear una función asíncronica (**async**), con lo cual vamos a tener que esperar (**await**) a que pase algo para ser ejecutada y si no se recibe respuesta que no avance.

Además incorporaremos un **try...catch**, que es una instrucción utilizada para el manejo de errores:

```
try {  
    //Instrucciones que pueden dar error  
} catch (error) {  
    //Acciones en caso de error  
}
```

Estructura básica de try...catch

Más información:

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/async_function

<https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/try...catch>

Fetch Async Await y manejo de errores

JS

```
// Asincrónico  
const getNombre = async (idPost) => {
```

Indicamos que la
función será asincrónica

Misma
estructura

```
  try {  
    const resPost = await fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)  
    const post = await resPost.json()  
    console.log(post.userId);  
  
    const resUser = await fetch(`https://jsonplaceholder.typicode.com/users/${post.userId}`)  
    const user = await resUser.json()  
    console.log(user.name);  
  } catch (error) {  
    console.log('Ocurrió un error grave', error);  
  }  
}  
getNombre(99); // llamada a la función
```

Pedimos que espere la
respuesta a la consulta

Se guarda la información
del error que ocurra

En caso de colocar una URL
equivocada el bloque try...catch
permitirá manejar el error

Si no suceden errores el bloque catch **no se ejecuta**.

Comparándolo con la escritura anterior, esta forma es más prolija y más controlable.

Ver ejemplo fetch-async-await (.html y .js)

Librería Axios

Axios es una **librería JavaScript** que puede ejecutarse en el navegador y que nos permite hacer sencillas las operaciones como cliente HTTP, por lo que podremos configurar y realizar solicitudes a un servidor y recibiremos respuestas fáciles de procesar.

Más información [aquí](#).

Para utilizar Axios debemos:

1. Ingresar a <https://cdnjs.com/libraries/axios>
2. Copiar el primer código y agregarlo al final del <body> en el documento HTML, antes del script del archivo externo:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/axios/0.21.1/axios.min.js"></script>  
<script src="async-await-axios.js"></script>
```

Para trabajar con Axios aprovecharemos el ejemplo anterior, donde vamos a tener una función asíncronica, vamos a seguir manejando errores pero a diferencia de la anterior vamos a recibir la respuesta pero no se la vamos a pedir a **Fetch**, sino que se la vamos a pedir a **Axios**.

Librería Axios

JS

```
// Asincrónico con Axios
const getNombre = async (idPost) => {
  try {
    const resPost = await axios(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
    console.log(resPost); //traemos el objeto completo
    console.log(resPost.data.userId); //traemos solo el userID, guardado dentro de data

    const resUser = await axios(`https://jsonplaceholder.typicode.com/users/${resPost.data.userId}`)
    console.log(resUser); //traemos el objeto completo
    console.log(resUser.data.name); //traemos solo el nombre, guardado dentro de data
  } catch (error) {
    console.log('Ocurrió un error grave', error);
  }
}
getNombre(99); // llamada a la función
```

¿Qué cambia con respecto al ejemplo anterior? En primer lugar utilizamos la librería **Axios** en vez de **Fetch** y ya no necesitamos convertir a JSON, sino que directamente vamos a poder acceder al dato a través de la respuesta. Haremos referencia al objeto **data** para obtener el **userID**. Haremos lo mismo para pedir los datos del usuario.

[Ver ejemplo async-await-axios \(.html y .js\)](#)

SPA: Single Page Application

SPA es un tipo de aplicación web donde todas las pantallas las muestra **en la misma página**, sin recargar el navegador.

Técnicamente, una SPA es un sitio donde existe un único punto de entrada, generalmente el archivo **index.html**. En la aplicación no hay ningún otro archivo HTML al que se pueda acceder de manera separada y que nos muestre un contenido o parte de la aplicación, toda la acción se produce dentro del mismo **index.html**.

Varias vistas, no varias páginas

Aunque solo tengamos una página, lo que sí tenemos en la aplicación son **varias vistas**, entendiendo por vista algo como lo que sería una pantalla en una aplicación de escritorio. En la misma página, por tanto, se irán intercambiando **vistas distintas**, produciendo el efecto de que tienes varias páginas, cuando realmente todo es la misma, intercambiando vistas.

*El efecto de las SPA es que **cargan muy rápido sus pantallas**. Aunque parezcan páginas distintas, realmente es la misma página, por eso la respuesta es muchas veces instantánea para pasar de una página a otra. Es normal que al interactuar con una SPA la URL que se muestra en la barra de direcciones del navegador vaya cambiando también. La clave es que, aunque cambie esta URL, la página no se recarga nunca. El hecho de cambiar esa URL es algo importante, ya que el propio navegador mantiene un historial de pantallas entre las que el usuario se podría mover, pulsando el botón de "atrás" en el navegador o "adelante".*

Fuente (para ampliar): <https://desarrolloweb.com/articulos/que-es-una-spa.html>

SPA: Single Page Application



Ver ejemplo SPA (La cocina de Juan): <https://dreamy-pike-507001.netlify.app/>

Características importantes del ejemplo:

- Aplican la lógica de tener separada la vista de los datos.
- Navegamos por la Web como si fuera una aplicación de escritorio. La respuesta es muy rápida, no estamos cargando cada página cuando accedemos a otra parte del menú.
- Las páginas se cargan de una vez, cuando estemos navegando vamos a hacerlo sobre contenido que ya está cargado.
- Podríamos hacer que la URL cambie y que se conserve el historial de usuario, además que puedan utilizar los botones de adelante y atrás.
- Así funciona **Gmail**, es una sola página, no es que se cargan todos los mensajes, se cargan los primeros, el resto de la carga es **on demand**. Lo mismo sucede con las páginas de los bancos cuando muestran los movimientos de la cuenta por partes.
- Quien se encarga de gestionar todo esto es JavaScript que le da comportamiento a la página.
- Podríamos comunicar, entonces, el **front** con el **back** a través de una API propia o de terceros.

Para ampliar:

¿Qué es una web SPA? - Single Page Application:
<https://www.youtube.com/watch?v=Fr5QGdJZBVo>

Ejemplos, cursos y guías de VUE.js. APIS

- **Guía de VUE.js:** <https://es.vuejs.org/v2/guide/index.html#>
- **Ejemplos VUE:** <https://vuejsexamples.com/>
- **Escuela VUE:** <https://escuelavue.es/series/>
- **¿Qué son las APIs y para qué sirven?:** <https://youtu.be/u2Ms34GE14U>
- **Curso de Vue JS - Tutorial en Español [Desde Cero]:**
<https://www.youtube.com/playlist?list=PLPl81lqbj-4J-gfAERGDCdOQtVgRhSvIT>
- **VUE Mastery (curso):** <https://www.vuemastery.com/courses/intro-to-vue-js/vue-instance/>
- **Lenguaje JS - ¿Qué es VUE?:** <https://lenguajejs.com/vuejs/introduccion/que-es-vue/>
- **Async & Await en Javascript con Fetch y Axios [Español]**
<https://www.youtube.com/watch?v=stiPdISkTOI>
- **Ver ejemplo en carpetas “JSON Breaking Bad” y “JsonFetchVue”**