<codoacodo/>

Agencia de Aprendizaje a lo largo de la vida

FULL STACK PYTHON
Clase 21

Vue 3

BA

SPA y asincronía

Les damos la bienvenida

Vamos a comenzar a grabar la clase

<codoacodo/>

Clase 20

DOM y Eventos

- Aplicaciones Reactivas. Reactividad en 2 sentidos.
- Propiedades computadas.
- Componentes.
- Watchers.
- Acceder a los elementos del DOM utilizando $refs
- Concepto MVC y MVVM.

Clase 21

SPA y Asincronía

- Introducción a SPA.
- SPA. ¿Qué es y qué beneficios tiene?
- Ejemplo práctico de un SPA en Vue.
- Enviar y pedir datos a un servidor
- Asincronía.
- Consumo de API REST a través de fetch y Axios.

Clase 22

Introducción a Base de Datos

- ¿Qué es una Base de datos?
- BBDD relacionales y no relacionales.
- Entorno MySQL. Instalación. Clientes MySQL.
- DER. Entidad, atributo y tipo de datos. Primary key.
- Lenguaje SQL y DDL.
- Backup y restauración de bases de datos.

MPA: Multiple Page Application

Tradicionalmente, el sistema que se seguía para crear páginas o aplicaciones web se enmarcaba dentro de la categoría de páginas **MPA** (*Multiple Page Application*). Bajo este sistema, el navegador se descarga el fichero **.html**, lo lee y luego realiza las peticiones de los restantes archivos relacionados que encuentra en el documento HTML. Si el usuario pulsa en algún enlace, se descarga el **.html** de dicho enlace (recargando la página completa) y se repite el proceso. Generalmente, es el que se utiliza frecuentemente en sitios web más tradicionales, los que usan mayoritariamente backend.

SPA

SPA (*Single Page Application*) 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 index.html. En la aplicación no hay ningún HTML al que se pueda acceder de manera separada, toda la acción se produce dentro del mismo documento HTML. Pero disponer de una sola página no implica tener una sola vista. En la misma página SPA se van intercambiando vistas distintas a partir de los eventos producidos, obteniéndose un resultado similar al que tendríamos utilizando varias páginas. En realidad todo ocurre en la misma SPA, intercambiando vistas.
Por ejemplo, páginas como WhatsApp Web, Twitter o Google Drive podrían ser **ejemplos de SPA**.

SPA: Single Page Application

Una característica de las SPA es que cargan muy rápido sus vistas. Aunque parezcan páginas distintas, realmente es la misma página, por eso la respuesta, al pasar de una página a otra, es muchas veces prácticamente instantánea.

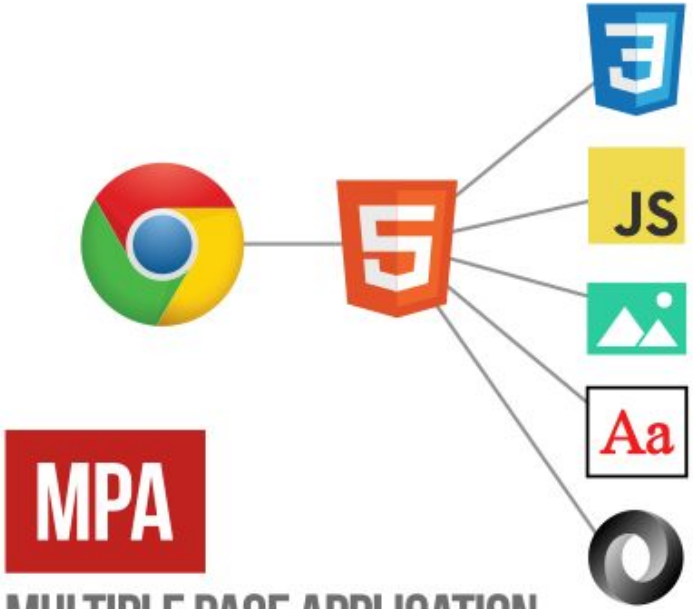
Es normal que al interactuar con una SPA la URL que se muestra en la barra de direcciones del navegador vaya cambiando también. La clave es que, aunque cambie esta URL, la página no se recarga nunca. El hecho de cambiar esa URL es algo importante, ya que el propio navegador mantiene un historial de pantallas entre las que el usuario se podría mover, pulsando el botón de "atrás" o "adelante" en el navegador. +info

Los sitios de tipo SPA son los que se desarrollan con frameworks de Javascript como, por ejemplo, React, Vue o Angular. Ejemplo SPA

SPA: Características importantes

Las SPA poseen, entre otras, las siguientes características:

- 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 sitio.
- Las páginas se cargan de una vez, navegamos sobre contenido que ya está cargado.
- Podemos hacer que la URL cambie y que se conserve el historial de usuario, además poder utilizar los botones adelante / atrás.
- Así funciona Gmail. No se cargan todos los mensajes, se cargan los primeros y el resto de la carga es *on demand*. Lo mismo sucede con las páginas de los bancos y los movimientos de la cuenta, por ejemplo.
- Quien se encarga de gestionar todo esto es JavaScript que le da comportamiento a la página.


JS
Aa
MPA
MULTIPLE PAGE APPLICATION


JS
JS
Aa
SPA
SINGLE PAGE APPLICATION

Asincronía

Mensajes HTTP

Hypertext Transfer Protocol (**HTTP**) (o *Protocolo de Transferencia de Hipertexto*) es un protocolo para la transmisión de documentos hipermedia, como HTML. Fue diseñado para la comunicación entre los navegadores y servidores web, entre otros propósitos. Sigue el **modelo cliente-servidor**, en el que un cliente establece una conexión, realizando una petición a un servidor y espera una respuesta del mismo.

Los **mensajes HTTP** son los medios por los cuales se intercambian datos entre servidores y clientes. Hay dos tipos de mensajes: **peticiones**, enviadas por el cliente al servidor, para pedir el inicio de una acción; y **respuestas**, que son la respuesta del servidor.

¿Qué es una petición 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, .webp 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.

Métodos HTTP

HTTP define una gran cantidad de métodos:

- GET: utilizado únicamente para consultar información al servidor (símil SELECT).
- POST: utilizado para solicitar la creación de un nuevo registro, es decir, algo que no existía previamente (símil INSERT).
- PUT: utilizado para actualizar por completo un registro existente (símil UPDATE).
- PATCH: similar al método PUT, pues permite actualizar un registro existente, sin embargo, este se utiliza cuando es necesario actualizar solo un fragmento del registro y no en su totalidad (símil UPDATE).
- DELETE: utilizado para eliminar un registro existente (símil DELETE).
- HEAD: 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.

Respuesta de la petición HTTP

Como mencionamos anteriormente, una **petición HTTP** es un mensaje que una computadora envía a otra utilizando el protocolo HTTP. La petición HTTP la hacen los clientes de nuestro API. Cuando nuestro API recibe esta petición, la procesa, y luego retorna una respuesta, llamada **respuesta HTTP**.

Los **códigos de estado de respuesta HTTP** indican si se ha completado satisfactoriamente una solicitud HTTP específica. Las respuestas se agrupan en cinco clases:

1. Respuestas informativas (100–199),
2. Respuestas satisfactorias (200–299),
3. Redirecciones (300–399),
4. Errores de los clientes (400–499),
5. y errores de los servidores (500–599).

1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

2XX Success	
200	OK
201	Created
202	Accepted
203	Non-authoritative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX Redirectional	
300	Multiple Choices
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

4XX Client Error Continued	
409	Conflict
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

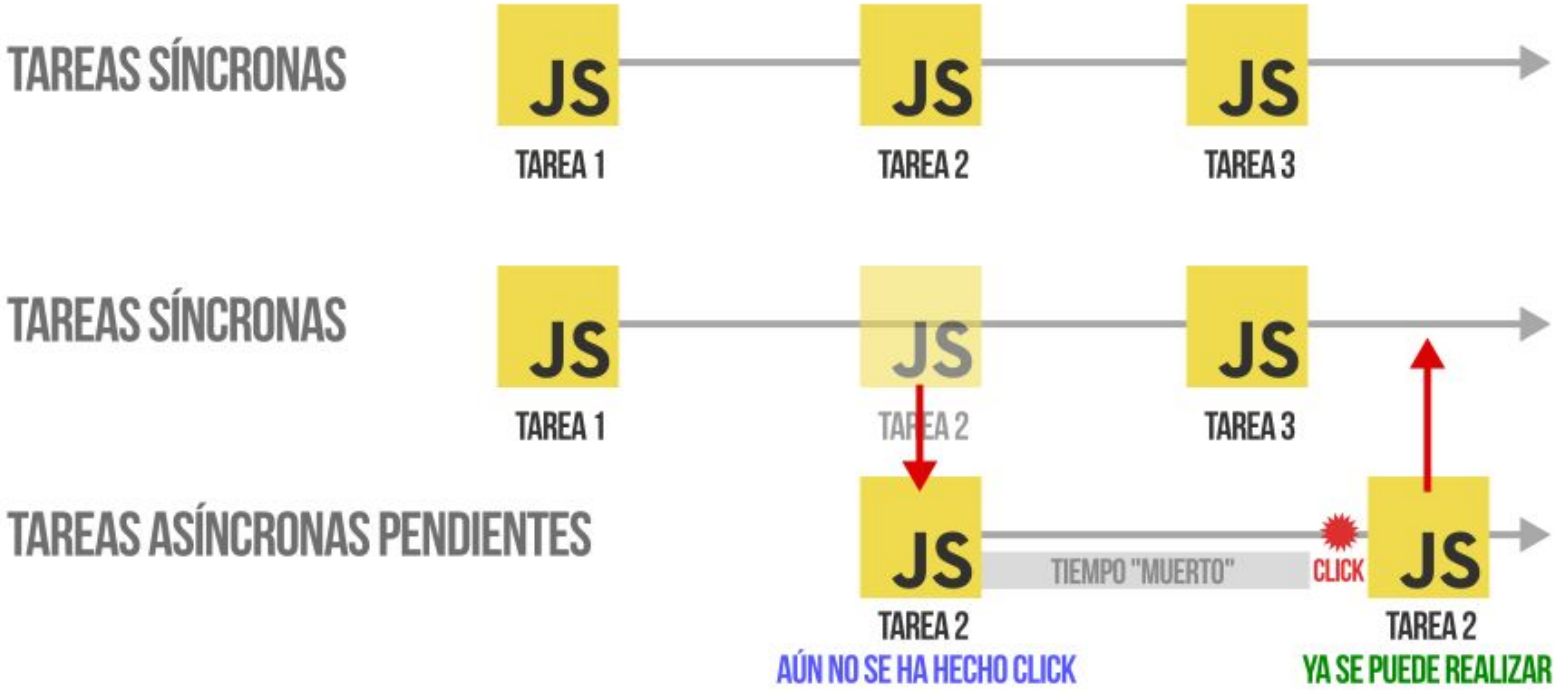
5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

HTTP STATUS CODES

When a browser requests a service from a web server, an error may occur.
This is a list of HTTP status messages that might be returned.

Asincronía

La **asincronía** es uno de los conceptos principales que rige el mundo de JavaScript. Cuando comenzamos a programar, normalmente realizamos tareas de forma **síncrona**, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra.
Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones **asíncronas**, especialmente en ciertos lenguajes como **JavaScript**, donde tenemos que realizar tareas que tienen que esperar a que ocurra un determinado suceso que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.


TAREAS SÍNCRONAS
JS
TAREA 1
JS
TAREA 2
JS
TAREA 3
TAREAS SÍNCRONAS
JS
TAREA 1
JS
TAREA 2
JS
TAREA 3
TAREAS ASÍNCRONAS PENDIENTES
JS
TAREA 2
AÚN NO SE HA HECHO CLICK
TIEMPO "MUERTO"
CLICK
JS
TAREA 2
YA SE PUEDE REALIZAR

¿Cómo gestionar la asincronía?

Teniendo en cuenta lo anterior, debemos aprender a buscar mecanismos para dejar claro en nuestro código JavaScript, que ciertas tareas tienen que procesarse de forma asíncrona para quedarse a la espera, y otras deben ejecutarse de forma síncrona.

En JavaScript existen varias formas de gestionar la asincronía, a continuación veremos algunas:

Fetch

La **API Fetch** proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. Además provee un **método** global **fetch()** que proporciona una forma **asíncrona** de obtener recursos desde la red. +info

El uso de **fetch()** más simple toma un argumento (la ruta del recurso a obtener) y devuelve un objeto **promise** pendiente, que más tarde puede proporcionar la respuesta en un objeto **response**. En otras palabras, ***fetch() promete una respuesta***, que puede eventualmente cumplir, y coloca el recurso solicitado en **response**, o puede fallar, en cuyo caso provee un mecanismo para manejar el error.

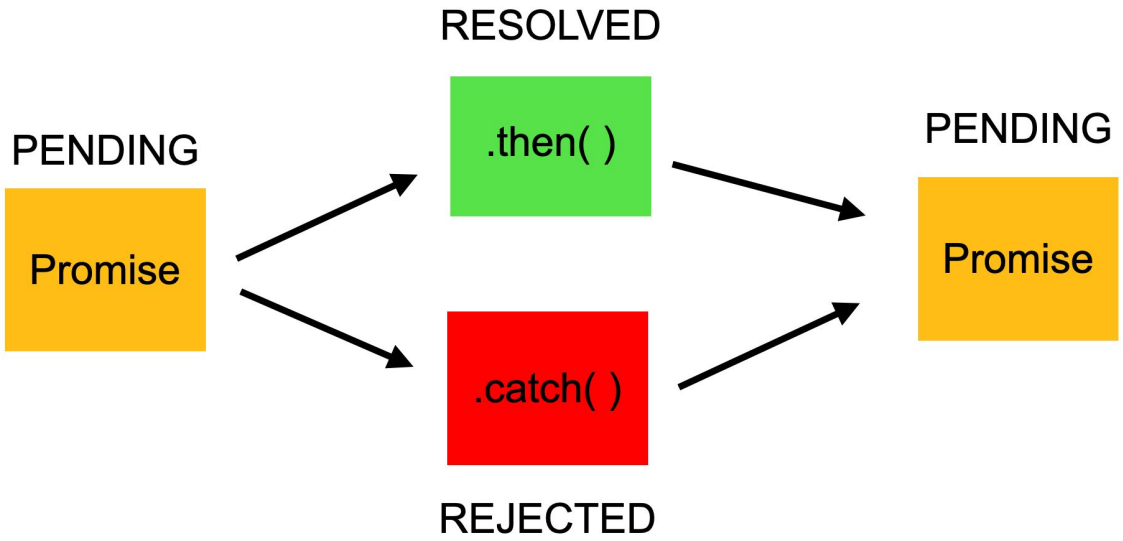
¿Qué es una promesa?

Una **promesa** (***promise***)en JS es similar a una promesa en la vida real. Tiene 2 resultados posibles: se mantendrá cuando llegue el momento, o no. Cuando definimos una promesa en JavaScript, se resolverá cuando llegue el momento, o será rechazada.

Hay 3 estados para un **objeto promise**:

- **Pendiente** (*pending*): estado inicial, antes de que la promesa tenga éxito o falle.
- **Resuelto** (*resolved*): promesa completada.
- **Rechazado** (*rejected*): promesa fallida.

¿Qué es una promesa?



Se llama al método **then( )** después de que se resuelva la promesa. Y se realiza una llamada al método **catch()** cuando es rechazada.

Fetch

Veamos cómo utilizar todo esto. El siguiente código recupera un archivo JSON a través de red y muestra su contenido en la consola.

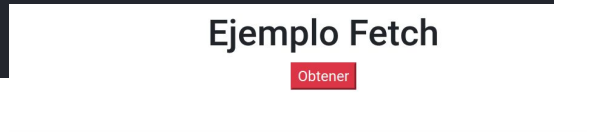
```
fetch('https://api.coindesk.com/v1/bpi/currentprice.json')
    .then(response => response.json())
    .then(data => console.log(data));
```

response es la respuesta HTTP. Posee un encabezado y otros datos propios del protocolo, por eso usamos el método json() para extraer el contenido JSON desde la respuesta. Y **luego** (*then*), lo mostramos en la consola. Los dos métodos **.then()** se ejecutan en el momento que la "*promesa*" anterior se cumple, de manera **asincrónica**.

Fetch

En el siguiente ejemplo utilizamos Bootstrap y para obtener y mostrar en el documento HTML el contenido del archivo de texto *texto.txt*, obtenido con **fetch()**, que se encuentra en la misma carpeta que el index.html:

```
<div class="container my-5 text-center">
  <h1>Ejemplo Fetch</h1>
  <button class="btn-danger"
     onclick="traer()">Obtener</button>
</div>
<div class="mt-5" id="contenido">
  <!--Contenido recuperado con Fetch -->
</div>
```




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

Consumir API externa

fetch() permite recuperar contenido que no se encuentra en el cliente. Para ello, el origen de los datos debe contar con un mecanismo denominado **API** (*application programming interface*). Las API permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

RandomUser es un sitio que implementa una API que, ante una solicitud, regresa datos de usuarios (ficticios) en formato JSON. Podemos usar fetch() para recuperar esos datos y procesarlos en nuestra aplicación. El proceso es muy similar al utilizado para leer un archivo local, pero difiere en la ruta que proporcionamos para obtener el recurso.

Consumir API externa con fetch

La función **traer()** obtiene datos desde una API externa.

Con **fetch(**'https:..'**)** proporcionamos la ruta a la API.

En **.then** guardamos en la variable **res** los datos formateados con el método **.json()**

Y el siguiente **.then** muestra por consola todos los resultados y agrega al documento HTML los valores en la posición 0 correspondientes a la imagen, el nombre y el email.

```
function traer() {
 fetch('https://randomuser.me/api')
   .then(res => res.json())
   .then(res => {
      console.log(res)
      console.log(res.results[0].email)
      contenido.innerHTML= `
        <img src="${res.results[0].picture.large}
          " width="100px"
          class="img-fluid rounded-circle">
        <p>Nombre: ${res.results[0].name.first}</p>
        <p>Mail: ${res.results[0].email}</p>`
   })
}
```

Consumir API externa

Este ejemplo es más completo, ya que consulta una API más de una vez. Se envía un requerimiento y en base a la respuesta se realiza una nueva solicitud.

Con este esquema podríamos pedir información de un post en una red social, y luego solicitar datos sobre el usuario que hizo ese posteo, utilizando el id contenido en el post para identificar al usuario.

Consumir API externa

Utilizaremos datos que devuelve jsonplaceholder.typicode.com/post desde su API, con la siguiente estructura:

Las APIs públicas contienen la documentación necesaria para poder utilizarla.

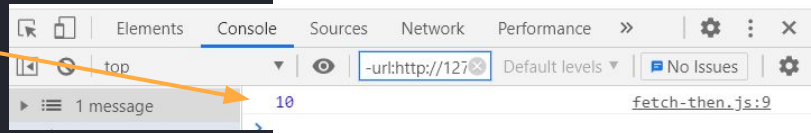
En el caso de este sitio en particular, se proporciona un archivo JSON con el contenido de los posteos realizados. Entre los datos vemos un **userId** y un **id**, que identifican de forma unívoca a los posteos y a los usuarios.

Para acceder a un posteo determinado podemos apuntar fetch() ha la misma dirección, pero agregando el numero de posteo al final: https://jsonplaceholder.typicode.com/posts/2 regresa un objeto que contiene los datos del post con id = 2.

Consumir API externa con fetch

La función **getNombre(post)** obtiene datos del posteo con el **id** = **post** y muestra en la consola el **userId** del mismo:

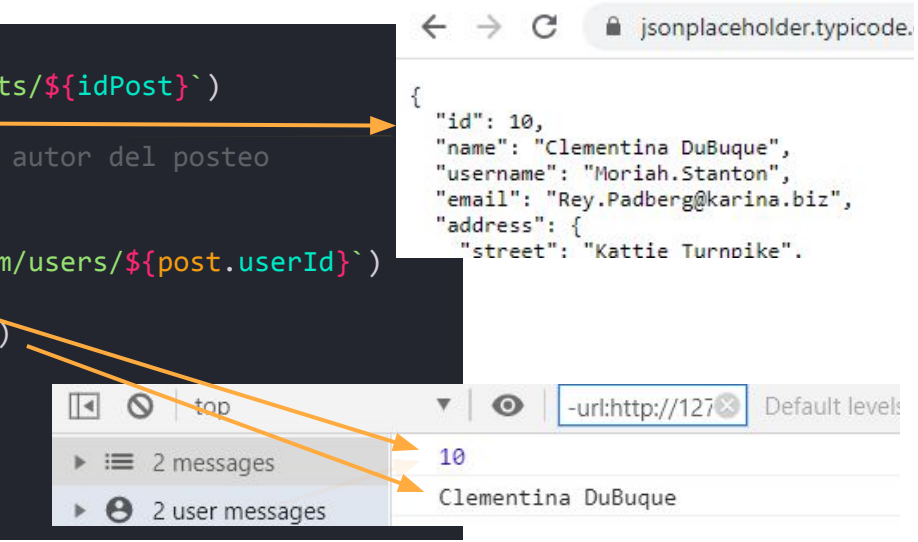
```
const getNombre= (idPost) => {
    // hacemos la solicitud a la API...
    fetch(`https://jsonplaceholder.typicode.com/posts/${idPost }`)
    // la API responde, y convertimos los datos al ormato JSON
    .then(res=> {
        return res.json()
    })
    // Mostramos el userID de ese posteo
    .then(post => {
        console.log(post.userId)
    })
}
getNombre(99); // llamada a la función
```



Consumir API externa con fetch

Esta nueva versión de **getNombre(post),** obtiene del posteo, pero luego realiza un segundo **fetch()** para conseguir los datos del usuario:

```
const getNombre = (idPost) => {
 fetch(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
   .then(res => { return res.json()  })
   // Solicitamos con otro fetch() los datos del autor del posteo
   .then(post => {
      console.log(post.userId)
      fetch(`https://jsonplaceholder.typicode.com/users/${post.userId}`)
        .then(res => { return res.json() })
        .then(user => { console.log(user.name) })
   })
}
getNombre(99); // llamada a la función
```



Fetch | Async, Await y manejo de errores

fetch() proporciona un mecanismo para gestionar el error que ocurre cuando los datos solicitados a la API no pueden ser recuperados. Dado que se trata de una comunicación asincrónica, utilizamos la palabra reservada **async** para indicar que la solicitud no es sincrónica.

Con palabra reservada **await** indicamos que alguna acción debe esperar a que se produzca una respuesta para ser ejecutada.

Y por último, mediante la estructura **try...catch** podemos ejecutar un bloque de código en caso de que la promesa se cumpla, u otro en caso de que la comunicación falle por algún motivo y la promesa sea rechazada.

Fetch | Async, Await y manejo de errores

Código con **fetch()**, **async**, **await** y **manejo de errores**:

```
// async indica que la función es asincrónica
const getNombre = async (idPost) => {
    // El bloque try se intenta ejecutar. En caso de error, se pasa a la sección catch(error)
    try {
        // await hace que el fetch NO SE PRODUZCA hasta que no esté disponible la respuesta
        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)
    // Este bloque solo se ejecuta si no se pudo ejecutar el bloque try. Error contiene el
    // código del error que se ha producido, para que podamos procesarlo.
    } catch (error) { console.log('Ocurrió un error grave', error) }
}
getNombre(99) // El llamado a la función se hace de la forma habitual.
```

Librería Axios

Axios es una librería JavaScript que permite hacer las operaciones como cliente HTTP de manera simple. Podemos configurar y realizar solicitudes a un servidor, y recibir respuestas en un formato más fácil de procesar. +info

Al igual que otras librerías y frameworks, para utilizar Axios en nuestros proyectos debemos incluir un enlace a su CDN y otro a un script JS. Las rutas actualizadas pueden obtenerse desde la web del desarrollador, y se ven así en nuestro código:

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

Librería Axios

Axios reemplaza a **fetch()** por una función propia. A pesar de que el código parece cambiar poco, utilizando **Axios** ya no necesitamos convertir a JSON los datos recibidos, porque la librería se encarga de ello.

Entre otras cosas, Axios separa los encabezados y todo lo que normalmente no utilizamos del paquete recibido, y convierte los datos a JSON.

A modo de ejemplo, vamos a reescribir el código anterior, pero en lugar de usar **fetch()**, **async**, **await** y manejo de errores mediante **try** / **catch**, reemplazamos **fetch()** por **axios()**, la función de la librería que se encarga de solicitar los datos a una AP, dando lugar a un código algo más corto y legible.

Fetch | Async, Await y manejo de errores

Código con **axios()**, **async**, **await** y **manejo de errores**:

```
// async indica que la función es asincrónica
const getNombre = async (idPost) => {
    // El bloque try se intenta ejecutar. En caso de error, se pasa a la sección catch(error)
    try {
        // await hace que axios espere por la respuesta
        const resPost = await axios(`https://jsonplaceholder.typicode.com/posts/${idPost}`)
        console.log(resPost);              //Mostramos el objeto (el post) completo
        console.log(resPost.data.userId)   //Mostramos el userID, contenido en data.userId

        const resUser = await axios(`https://jsonplaceholder.typicode.com/users/${resPost.data.userId}`)
        console.log(resUser)               //Mostramos el objeto (el usuario) completo
        console.log(resUser.data.name)     //Mostramos el nombre del usuario, que está en data.name
    } catch (error) {
        console.log('Ocurrió un error grave', error)
    }
}
getNombre(99);  // El llamado a la función se hace de la forma habitual.
```

Material extra

Material complementario

API Rest con JavaScript:

- 4 Maneras de llamar a una API Rest con JavaScript: https://dev.to/vikingcodeblog/4-maneras-de-llamar-a-una-api-rest-con-javascript-2nhm
- APIs gratuitas para programar tus proyectos: https://github.com/public-apis/public-apis Más de 1.500 APIs gratuitas para utilizar en tus apps (tiempo, películas, libros, eventos, cotizaciones, etc.)
- Create Your Own Fake JSON API: https://mocki.io/fake-json-api (allows you to create a fake JSON API with your own fake data). Otro servicio similar: https://beeceptor.com/

Artículos de interés

Material extra:

- Todo sobre el protocolo HTTP, en MDN
- ¿Qué es una API REST?, en IBM
- ¿Qué es la programación asíncrona?
- El objeto promise, en MDN
- Curso de promesas en JavaScript, en freecodecamp.org
- ¿Qué es una SPA?

Videos:

- ¿Qué son las APIs y para qué sirven?
- ¿Qué es una SPA?

Actividades prácticas

- Para el TPO deberá poder consumir una API Rest desde JavaScript. Recuerde que deberá informar la API utilizada al momento de entregar el trabajo práctico.

No te olvides de dar el presente

Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

Todo en el Aula Virtual.

Muchas gracias por tu atención.

Nos vemos pronto