Buenos Aires aprende
Agencia de Habilidades para el Futuro

Talento Tech

Front-End JS

Clase 14 - “Asincronía”

Buenos Aires Ciudad

¡Les damos la bienvenida!

 Vamos a comenzar a grabar la clase

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Buenos Aires Ciudad

Talento Tech

Clase 13.

JS 5 - LocalStorage, SessionStorage y Carrito de Compras

1. Introducción a LocalStorage y SessionStorage
2. Diferencias entre LocalStorage y SessionStorage
3. Implementación de un carrito de compras utilizando LocalStorage o SessionStorage

Clase 14.

JS 6 - Asincronía

1. Asincronía
2. Consumo de API REST a través de fetch
3. Procesamiento de los datos
4. Incluir los datos consumidos y procesados por medio de fetch en nuestro proyecto

Clase 15.

API y Procesamiento de Datos

1. Asincronía Desarrollo de un proyecto integrador que combine HTML, CSS, y JavaScript
2. Consumo de API REST
3. Incorporación y procesamiento de los datos en nuestro HTML
4. Incorporación de buenas prácticas de accesibilidad y SEO
5. Presentación del proyecto final


Asynchronous JS

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 la persona 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 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, por ejemplo, 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 este 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.

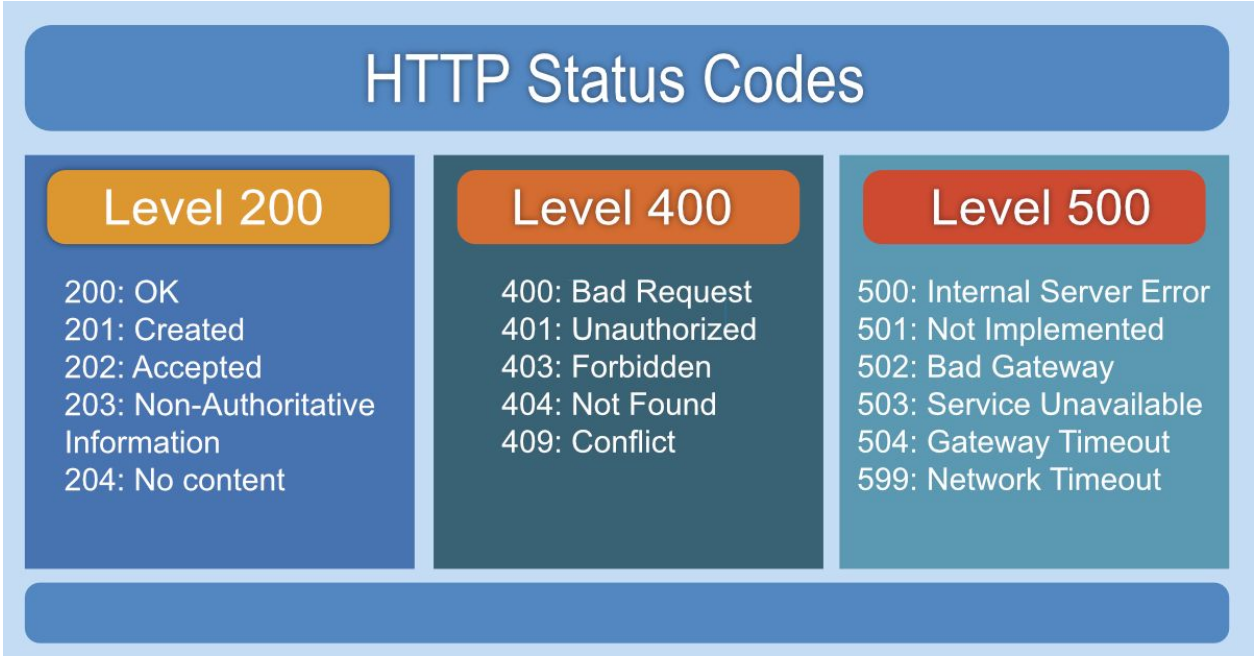
Respuestas de la petición HTTP

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).

HTTP STATUS CODES

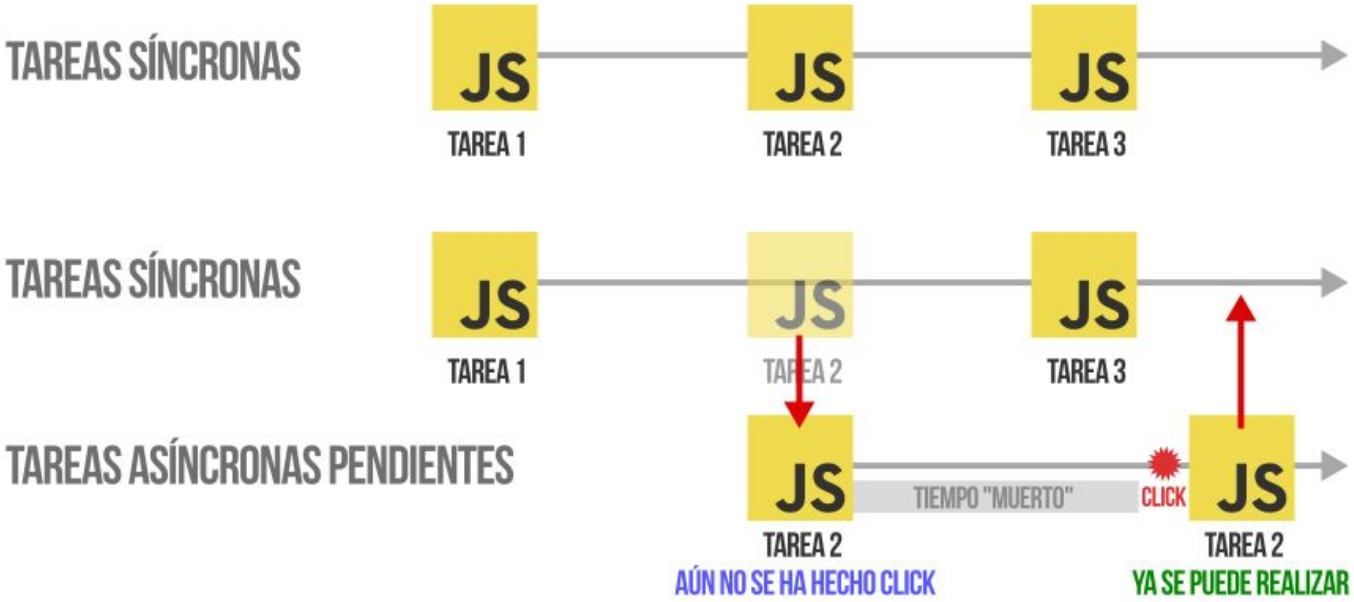


Asincronía

La asincronía es uno de los conceptos principales que rige el mundo de JavaScript. Cuando comenzamos a programar, normalmente realizamos tareas de forma síncrona (o al mismo tiempo), 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, para luego reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.

Asincronía


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 en caso de falla provee un mecanismo para manejar el error.

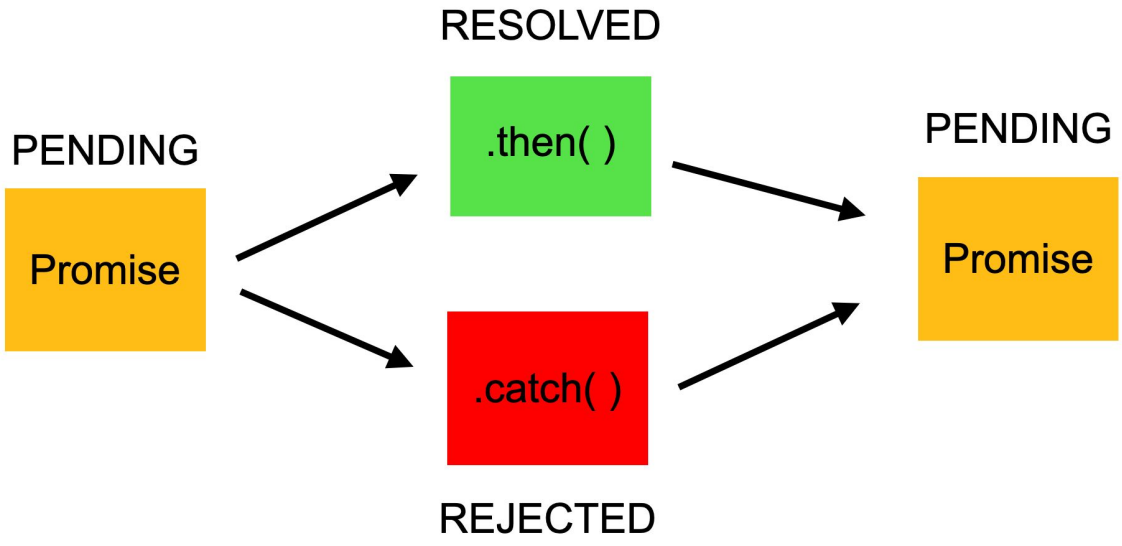
¿Qué es una promesa?

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. O, caso contrario, 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:

```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>
```

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

Consumo de API externa

El método 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.

Consumo de API con Fetch

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

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

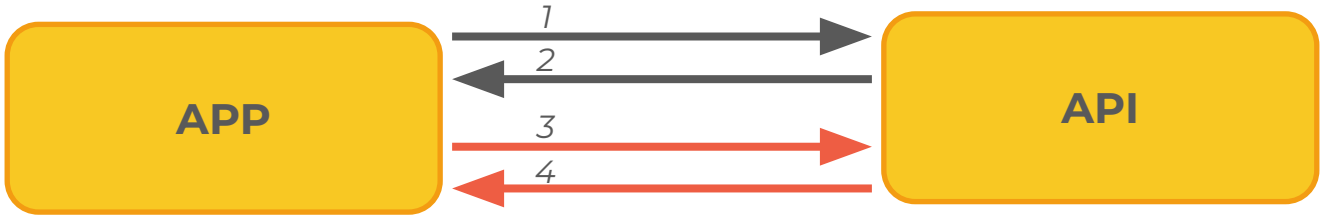
En .then guardamos en la variable res los datos formateados con el método .json()

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>`
   })
}
```

Consumo de 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.

Consumo de 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.

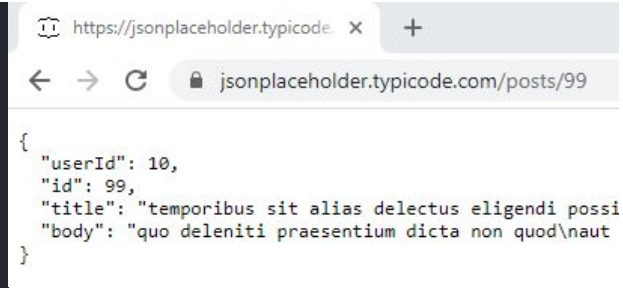
En el caso de este sitio en particular, se proporciona un archivo JSON con el contenido de los posteos realizados. Entre los datos vemos un userId y un id, que identifican de forma unívoca a los posteos y a los usuarios.

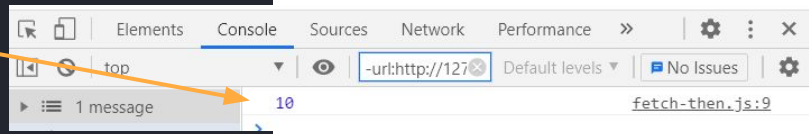
Para acceder a un posteo determinado podemos apuntar fetch() ha la misma dirección, pero agregando el número de posteo al final: https://jsonplaceholder.typicode.com/posts/2 regresa un objeto que contiene los datos del post con id = 2.

Consumo de API externa

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
```





Buenos Aires aprende
Agencia de Habilidades para el Futuro
Buenos Aires Ciudad

Talento Tech

Fetch | Async, await y manejo de errores

El método 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.
```

¡Vamos a la práctica! 

Ejercicios prácticos

Optativos | No entregables

Aplicar el consumo de API Fetch en tu proyecto personal

En este ejercicio, vas a integrar el consumo de una API REST utilizando `fetch()` en tu proyecto personal de e-commerce o cualquier otro proyecto que estés desarrollando. Los pasos a seguir son:

1. **Elegí una API pública** (como Fake Store API) que te proporcione datos de productos, usuarios, o cualquier otro recurso que quieras mostrar en tu proyecto.
2. Usá **fetch()** para hacer una solicitud a la API y obtener los datos.
3. Mostrá los datos obtenidos en tu proyecto, ya sea en forma de lista de productos, usuarios, o lo que elijas.
4. Asegurate de manejar los posibles errores utilizando `.catch()` y mpstrá un mensaje si algo falla.

Tips:

- **Selección de API**: Buscá una API pública que tenga los datos que querés incluir en tu proyecto. Algunas opciones son APIs de productos, películas, usuarios, etc.
- **Integración con el proyecto**: Pensá cómo podés integrar los datos obtenidos con otras funcionalidades de tu proyecto.

Ejercicios prácticos

Optativos | No entregables

Crear un carrito de compras dinámico con productos de una API

Vas a crear un carrito de compras dinámico que permita a los usuarios agregar productos a su carrito utilizando datos obtenidos de una API externa. Los pasos específicos son:

1. Utilizá fetch() para obtener una lista de productos desde una API (puede ser la misma API de productos del Ejercicio 1).
2. Mostrá los productos en la página en forma de tarjetas o lista.
3. Agregá un botón "Añadir al carrito" para cada producto. Al hacer clic en el botón, el producto debe añadirse al carrito.
4. Usá LocalStorage para almacenar los productos que el usuario agregue al carrito, de manera que si recarga la página, los productos sigan allí.
5. Mostrá la cantidad de productos que hay en el carrito en todo momento, actualizándola cada vez que se añada un nuevo producto.

Tips:

- Manipulación del DOM: Usá métodos como createElement() y appendChild() para crear dinámicamente las tarjetas de productos y los botones.
- LocalStorage: Almacená los productos agregados en LocalStorage utilizando JSON.stringify() y JSON.parse() para convertir los objetos a formato JSON y viceversa.