



FULL STACK

**Comenzamos en unos  
minutos**

---

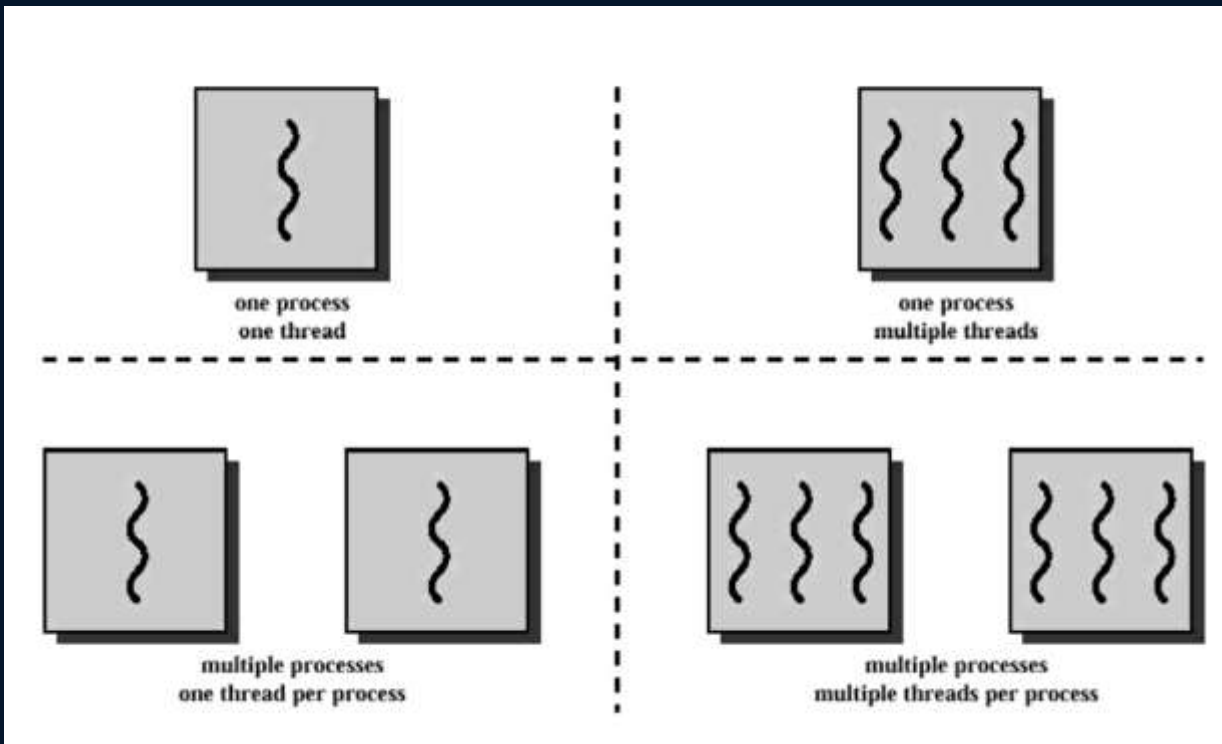
ACADEMY  
by NUMEN

# Javascript: Clase 8

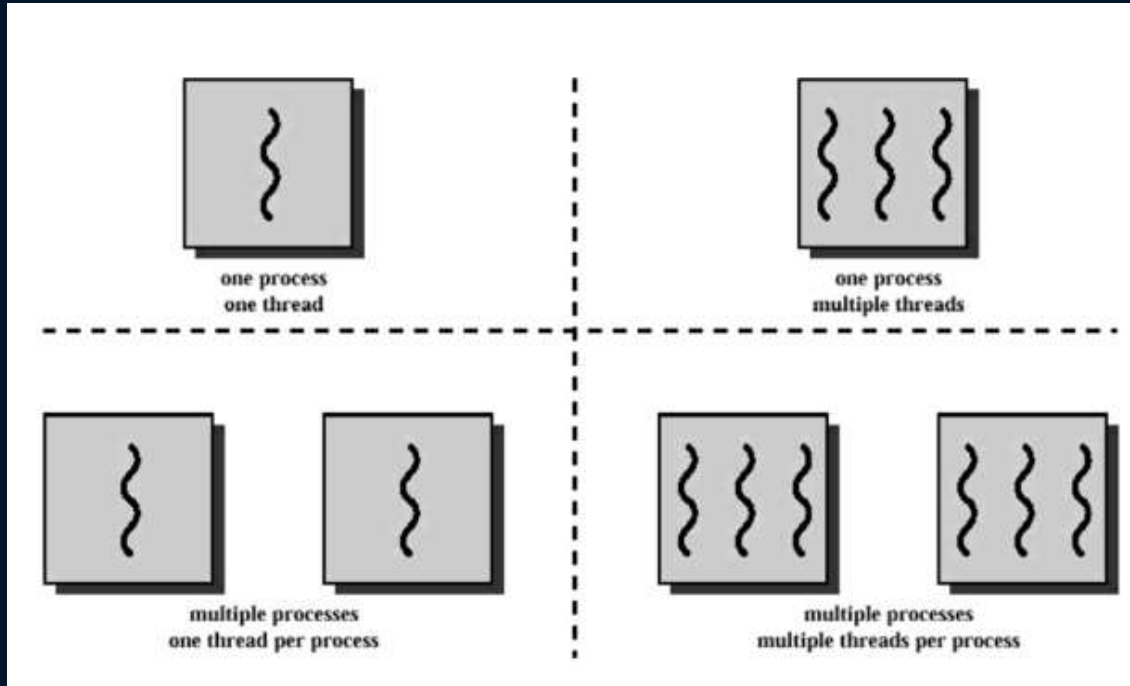
# Asincronia

# Single Threaded y Sincrónico

En ciencias de la computación un thread (o hilo de ejecución) es la secuencia de instrucciones más pequeña que puede ser manejada por un planificador de recursos (él que se encarga de repartir el tiempo disponible de los recursos del sistema entre todos los procesos) del Sistema Operativo.



JavaScript es **Single Threaded** y sincrónico, es decir que sólo puede hacer un sólo comando o instrucción en cada momento y que lo hace en orden, empieza la instrucción siguiente cuando termina la anterior. Esto puede sonar confuso, porque vemos que, en el browser por ejemplo suceden muchas cosas al *mismo tiempo* o bien, cuando tiramos una función asíncrona y esta se realiza mientras nosotros hacemos otras cosas, etc... esto sucede porque en general usamos javascript en conjunto con otros **procesos**, que pueden ser o no single threaded y en conjunto nos da la sensación que está ocurriendo todo al mismo tiempo, aunque es muy probable que no sea así.



# Asincronía en JavaScript



Como mencionamos anteriormente, JavaScript es un lenguaje de programación de un solo subproceso o hilo (single thread), lo que significa que sólo puede ejecutar una cosa a la vez, de manera sincrónica , por lo que si tuviéramos muchos thread que ejecutar, estos se ejecutarían uno detrás del otro, lo cual crearía una “pila” de procesos a ser ejecutados. A esta pila de procesos se le suele llamar como el “**stack**”. Ahora bien, cuando un intérprete (como el intérprete de JavaScript en un navegador web) hace un llamado para ejecutar múltiples procesos o funciones (un proceso o función que a la vez llama a otro proceso o función, y así sucesivamente) , esto lo hace a través de un mecanismo denominado “pila de llamadas” o “**Call Stack**”

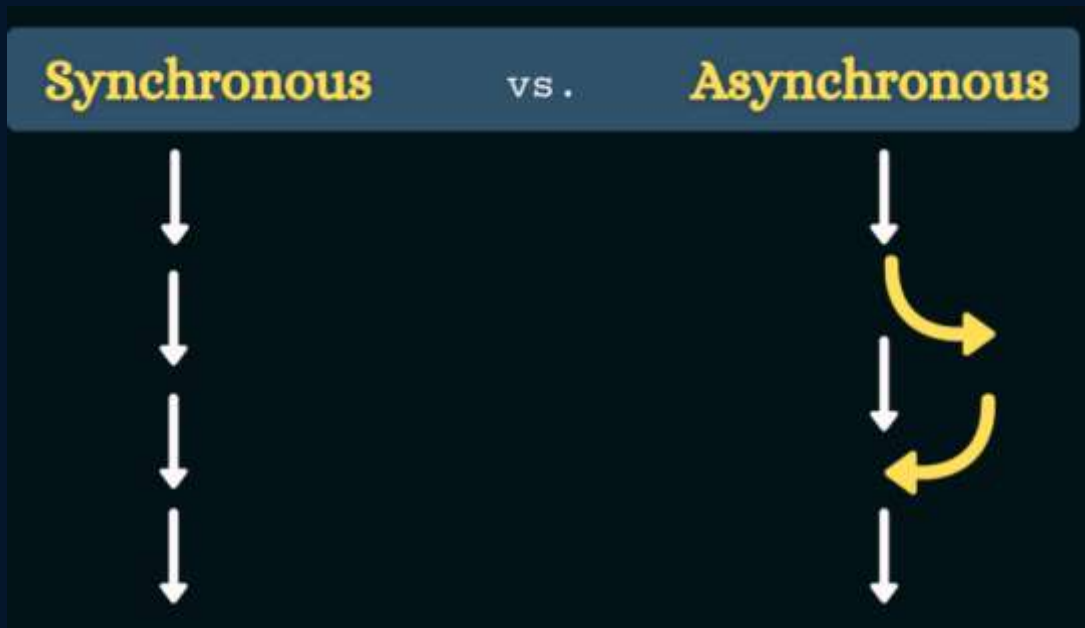




# JavaScript: Sincrónico y Asíncrono

**Sincrónico** : La respuesta sucede en el presente, una operación síncrona esperará el resultado.

**Asincrónico** : La respuesta sucede a futuro, una operación asíncrona no esperará el resultado.



# setTimeout()

El `setTimeout()` es un método que recibe una función y nos permite establecer el momento de ejecución.

Este método recibe 2 parámetros: Por un lado la función a ejecutar y por otro lado el tiempo que queremos que demore en ejecutarse.

```
setTimeout(function() {  
    console.log('Ejecutando un setTimeout')  
}, 1000);
```

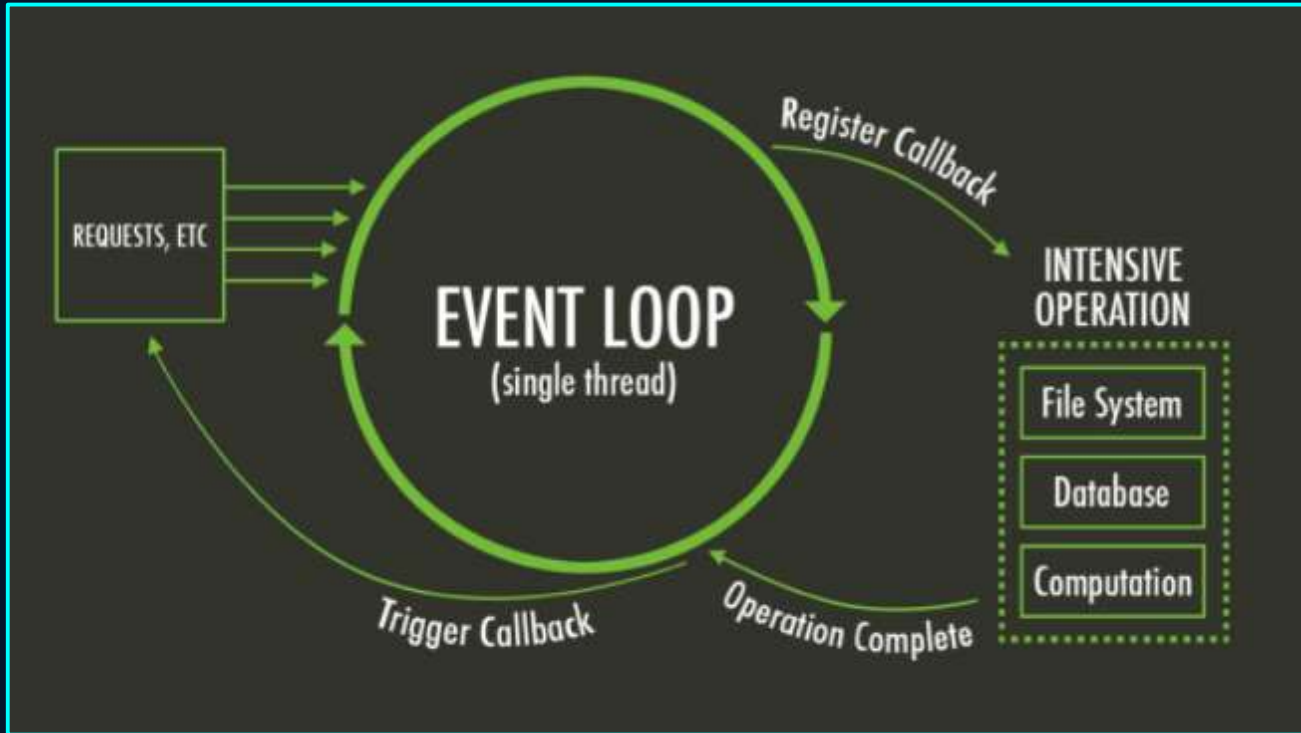
# setInterval()

El `setInterval()` es un método similar al `setTimeout()` con la diferencia que en lugar de ejecutarse una única vez, se repite una y otra vez cada vez que transcurre el tiempo que se le indico como parámetro.

```
setInterval(() => {  
    console.log('Ejecutando un setInterval')  
}, 1000);
```

```
setInterval(() => {  
    console.log(new Date().toLocaleTimeString())  
}, 1000);
```

# Ciclo de eventos / Event loop

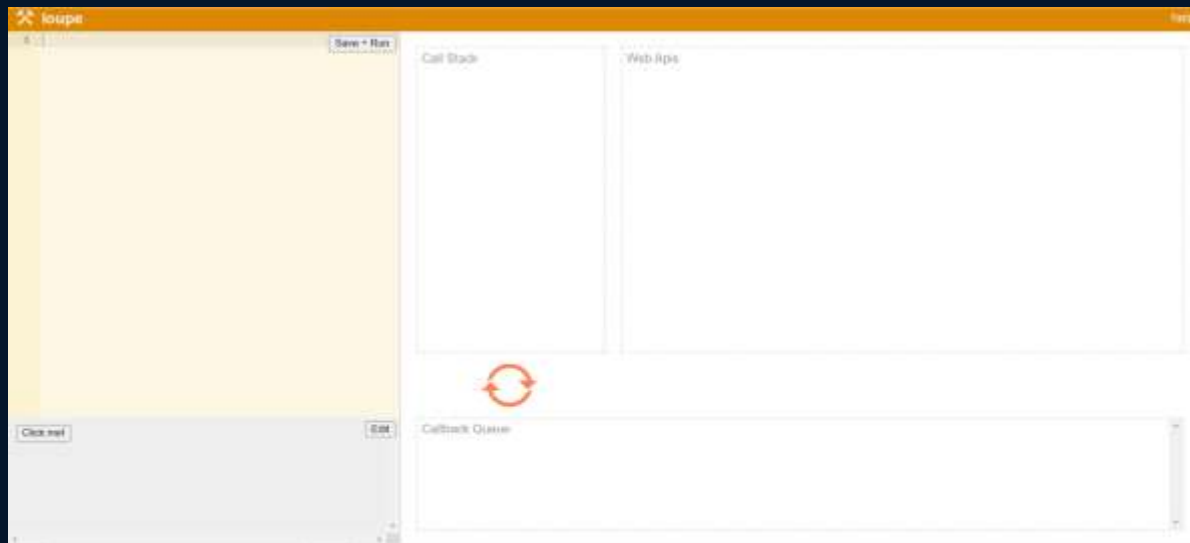


# Modelo de Event Loop

En este programa podremos ver de manera dinámica cómo funciona el **event loop** tanto con llamadas sincrónicas como asincrónicas.

Vamos a ello!!

<http://latentflip.com/loupe/?code=!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D>



# LIFO: Last in - First out

## Modelo Síncrono

Este es el formato en el que Javascript procesa las instrucciones. Cuando una instrucción se dispara, se une a la cola de espera, pero se pone en primer lugar. De este modo, la última instrucción en unirse es la primera en ejecutarse.

Esto permite a las instrucciones que estaban en espera ejecutarse apenas se resuelven evitando que se bloqueen otras instrucciones ya sea en la espera de resolución como en la espera de ejecución.

```
console.log('Codigo Sincrono');  
console.log('Inicio');  
  
function dos() {  
    console.log('Dos');  
}  
  
function uno() {  
    console.log('Uno');  
    dos();  
    console.log('Tres');  
}  
  
uno();  
console.log('Fin');
```

# FIFO (First In First Out)

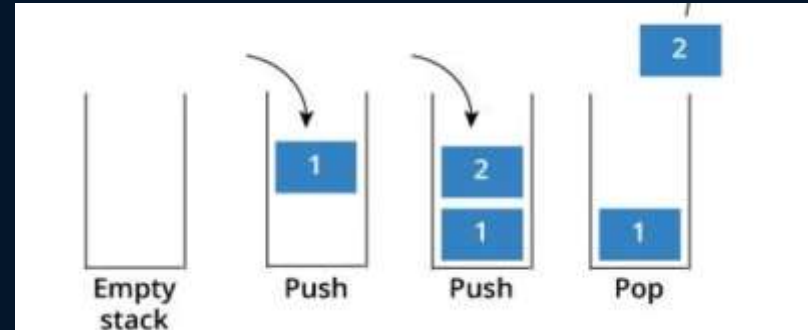
## Modelo Asíncrono

En el modelo asíncrono, la situación cambia, ya que aquellas funciones que se ejecutan con el `setTimeout`, pasarán a realizarse paralelamente en un entorno diferente. Así, el intérprete seguirá leyendo y ejecutando las siguientes líneas de código. Solo cuando haya ejecuta la última línea de código, tomará los resultados de la función `setTimeout` y los agregara al stack para procesarlos. Y estas estarán en el orden en que fueron procesadas

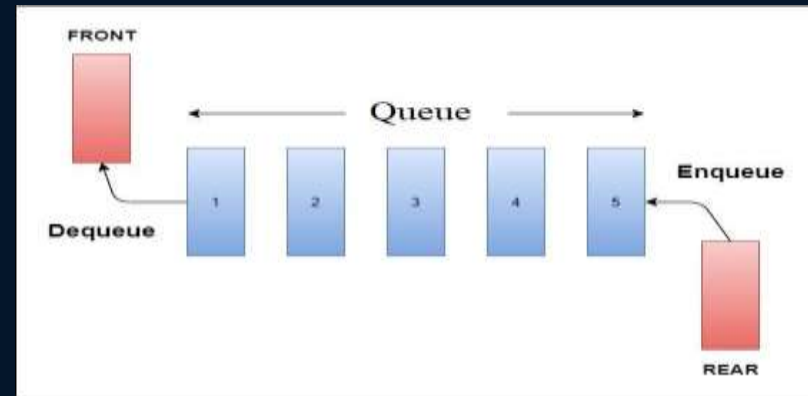
```
console.log('Codigo Asincrono');  
console.log('Inicio');  
  
function dos() {  
    setTimeout(function () {  
        console.log('Dos');  
    }, 1000)  
}  
  
function uno() {  
    setTimeout(function () {  
        console.log('Uno');  
    }, 0)  
    dos();  
    console.log('Tres');  
}  
  
uno();  
console.log('Fin')
```

# Pilas/Stacks VS Colas/Queues

Una pila es una estructura para almacenar datos que opera de forma lineal y unidireccional. Esto significa que solo hay una forma para agregar elementos y estos se incorporan en un orden determinado en una sola dirección (*de inicio a fin*). Modalidad LIFO (Last in First Out) El último elemento que agregamos será el primero que saquemos de la pila.



Una cola es una estructura de datos que opera de forma lineal y unidireccional (*se agregan elementos de inicio a fin*). La gran diferencia radica en la forma en que estos elementos son sacados después. Opera con la modalidad FIFO (First In First Out), es decir, **siempre el primer elemento que agreguemos, será el primero que saquemos de ella.**





# Peticiones y objeto de respuesta

# Peticiones al servidor

Hay ciertas ocasiones en las que necesitamos recurrir a grandes cantidades de información que solo están disponibles en bases de datos en el servidor. Con esa información podemos crear tarjetas, tablas, listas, y muchas cosas más. Tener toda esa información en el servidor nos permite poder recurrir a ella en todo momento.

El servidor puede alojar una enorme cantidad de información que de tenerla dentro de nuestro proyecto haría que este fuese muy pesado.

A su vez se puede extraer parte de la información de una base de datos para crear fracciones más pequeñas y de más fácil acceso para su consumición: Las APIs



# ¿Qué es una API?

Una API es un conjunto de definiciones y protocolos que se utilizan para desarrollar e **integrar** el software de las aplicaciones. API significa interfaz de programación de aplicaciones.



# ¿Que es un end point?

Un endpoint es cualquier dispositivo que sea físicamente la parte final de una red. Las computadoras de escritorio, las tablets, los smartphones, los dispositivos de oficina de red, como los routers, las impresoras y las cámaras de seguridad también son considerados endpoints. Los servidores también pueden ser considerados endpoints porque también están conectados a la red.

Básicamente cualquier dispositivo final conectado a la red es un endpoint.

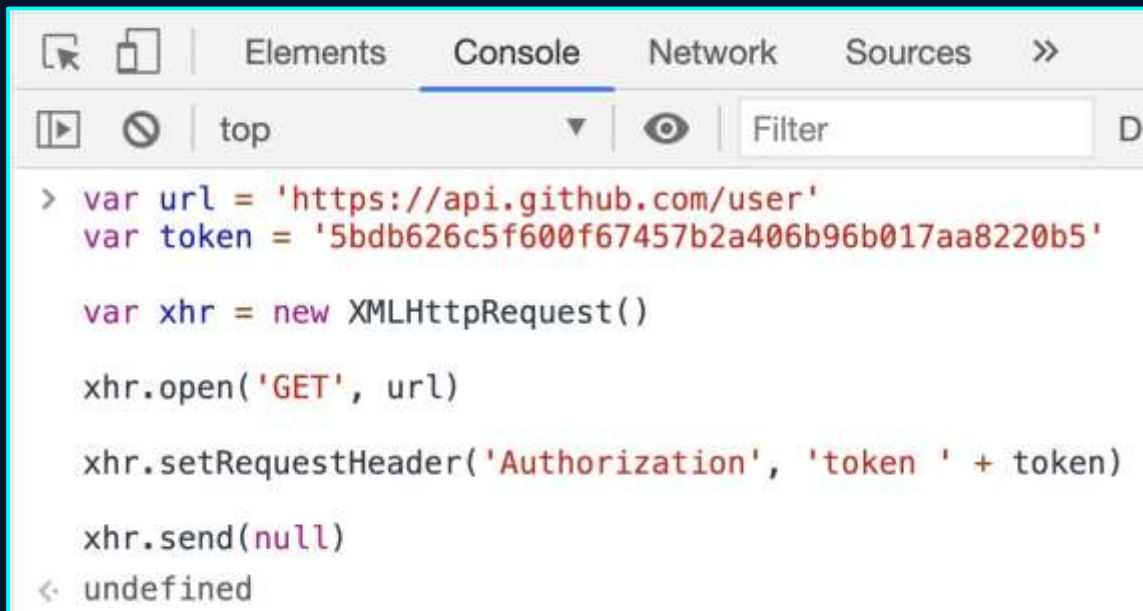
## User Endpoints

GET	/users/self	...	Get information about the owner of the access token.
GET	/users/ user-id	...	Get information about a user.
GET	/users/self/media/recent	...	Get the most recent media of the user.
GET	/users/ user-id /media/recent	...	Get the most recent media of a user.
GET	/users/self/media/liked	...	Get the recent media liked by the user.
GET	/users/search	...	Search for a user by name.

# El objeto XMLHttpRequest

Antiguamente, las peticiones en JavaScript se realizaban por medio de una instancia del objeto **XMLHttpRequest**. Este objeto nos permitía acceder a propiedades con información relativa al endpoint con el que deseamos conectarnos.

Esta forma demandaba muchas líneas de código para realizar incluso las peticiones más simples y desde el año 2015 fue reemplazada por las peticiones de tipo **fetch()**.

A screenshot of a web browser's developer console. The 'Console' tab is selected, showing a JavaScript code snippet. The code defines a URL, a token, creates an XMLHttpRequest object, opens a GET request, sets an Authorization header, and sends the request. The final output in the console is 'undefined'.

```
> var url = 'https://api.github.com/user'
   var token = '5bdb626c5f600f67457b2a406b96b017aa8220b5'

   var xhr = new XMLHttpRequest()

   xhr.open('GET', url)

   xhr.setRequestHeader('Authorization', 'token ' + token)

   xhr.send(null)
< undefined
```

# Peticiones fetch()

Existe un método en JavaScript llamado `fetch()`. Este método nos permite conectarnos con un endpoint y obtener un objeto de respuesta. En este caso, el endpoint seleccionado corresponde a una API con temática de la famosa serie Breaking Bad. Aquí nos estamos trayendo la información de todos los personajes de la serie.

```
fetch('https://www.breakingbadapi.com/api/characters')  
  .then((response) => console.log(response))
```

Al realizar la petición observamos que, si imprimimos en la consola la respuesta de dicha petición, obtenemos un objeto de respuesta con muchas propiedades.

ANALICEMOS UN POCO DICHO OBJETO!!



El objeto de respuesta posee un cuerpo denominado **body** que posee toda la información de los personajes que necesitamos en forma de texto plano.

También posee una cabecera llamada **headers** donde especifica en qué tipo de notación vamos a intercambiar información. Esta puede ser en formato JSON, XML, entre otras.

La propiedad **URL** especifica el endpoint al que hicimos la petición.

```
▼ Response ⓘ  
  body: (...)  
  bodyUsed: false  
  ▶ headers: Headers {}  
  ok: true  
  redirected: false  
  status: 200  
  statusText: "OK"  
  type: "cors"  
  url: "https://www.breakingbadapi.com/api/characters"  
  ▶ [[Prototype]]: Response
```

Por último las propiedades **status** y **statusText** nos brindan información sobre el estado de la respuesta, tema que ampliaremos más en la siguiente diapositiva.

# Códigos de estado de respuesta HTTP

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 ).

Más información en <https://developer.mozilla.org/es/docs/Web/HTTP/Status>

# JSON

Como mencionamos previamente, el **body** de la petición nos trae la información desde la API en forma de texto plano. Pero este formato no es trabajable con JavaScript, ya que no podemos hacer iteraciones sobre texto plano.

Para poder trabajar con él necesitamos pasarlo a una notación que se parezca a un formato de objetos y arreglos.

Para esto, **Douglas Crockford**, el autor del libro JavaScript The Good Parts, un clásico de las buenas prácticas, desarrolló el sistema de notación **JSON**, que significa **JavaScript Object Notation**.

```
{
  "localizaciones": [
    {
      "latitude": 40.416875,
      "longitude": -3.703308,
      "city": "Madrid",
      "description": "Puerta del Sol"
    },
    {
      "latitude": 40.417438,
      "longitude": -3.693363,
      "city": "Madrid",
      "description": "Paseo del Prado"
    },
    {
      "latitude": 40.407015,
      "longitude": -3.691163,
      "city": "Madrid",
      "description": "Estación de Atocha"
    }
  ]
}
```

Para pasar el texto de la respuesta a JSON, simplemente aplicamos el método `json()` al objeto de respuesta de la petición. Si observamos la **data** convertida, podemos ver que se transformó en un enorme arreglo de objetos donde cada objeto es un personaje de la serie.

```
fetch('https://www.breakingbadapi.com/api/characters')  
  .then((response) => response.json())  
  .then((data) => console.log(data));
```

Con esta **data** convertida podemos hacer lo que deseemos, ya sea, crear una lista de tarjetas, donde cada tarjeta contiene la información de un objeto del arreglo, también podemos crear una tabla, una lista de información o cualquier formato de presentación de información que se nos ocurra.

# Creando una galería de tarjetas

```

const d = document,
$section = d.createElement('section'),
$fragment = d.createDocumentFragment();

fetch('https://www.breakingbadapi.com/api/characters')
  .then((response) => response.json())
  .then((data) => {
    console.log(data)
    data.map(personaje => {
      const $figure = d.createElement('figure'),
        $img = d.createElement('img'),
        $figcaption = d.createElement('figcaption'),
        $h3 = d.createElement('h3'),
        $p = d.createElement('p');

      $img.setAttribute('src', personaje.img)
      $img.setAttribute('alt', personaje.name)
      $img.setAttribute('width', '200px')
      $img.setAttribute('height', '220px')
      $h3.textContent = personaje.name
      $p.textContent = `Alias: ${personaje.nickname}`

```

```

$figure.style.border = '2px solid black'
$figure.style.width = "200px"
$figure.style.display = "flex"
$figure.style.flexDirection = "column"
$figure.style.alignItems = "center"
$figure.style.textAlign = "center"
$img.style.objectFit = 'cover'

$figure.appendChild($img);
$figure.appendChild($figcaption);
$figcaption.appendChild($h3);
$figcaption.appendChild($p)

$fragment.appendChild($figure);
})
$section.appendChild($fragment);
$section.style.display = "flex";
$section.style.flexWrap = "wrap";

d.body.appendChild($section)
}
);

```



# Arquitectura REST

# REST

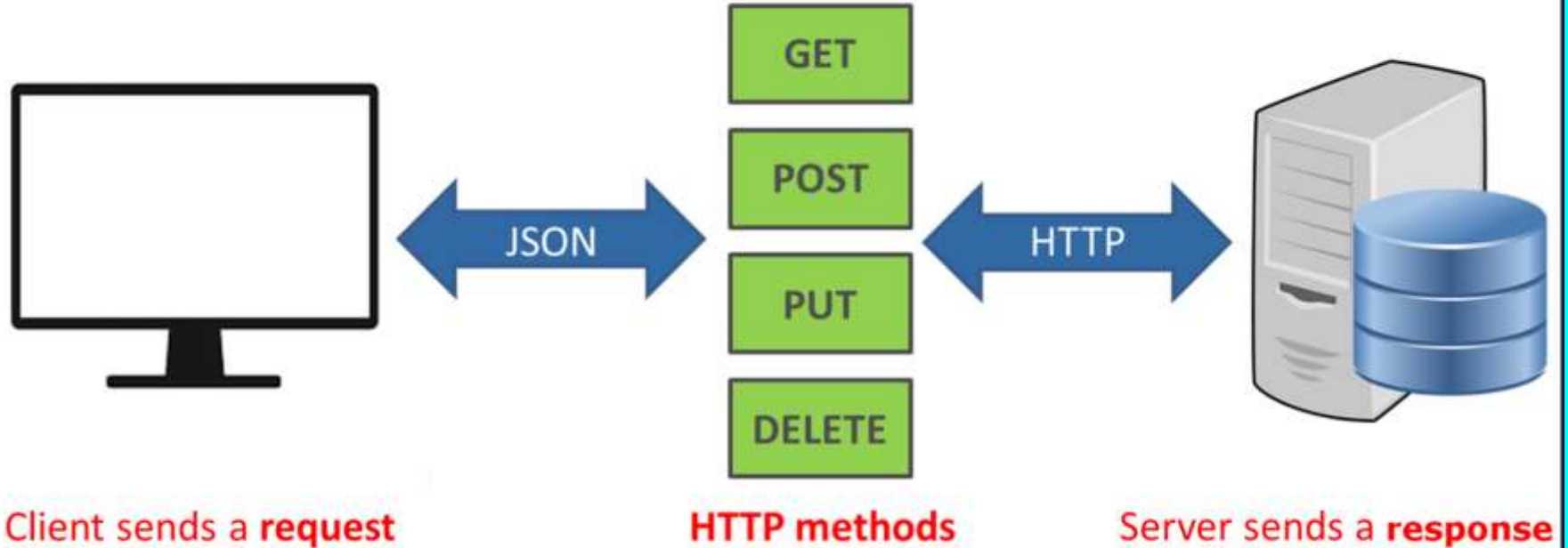
REST es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON. Es una alternativa en auge a otros protocolos estándar de intercambio de datos como SOAP (Simple Object Access Protocol), que disponen de una gran capacidad pero también mucha complejidad.

A veces es preferible **una solución más sencilla de manipulación de datos como REST.**





# Arquitectura Rest



# ¿Que es CRUD?



---

**Create Read Udate Delete**

Armar un crud en JavaScript puro es algo engorroso, pero cuando conozcamos React, veremos que podremos lograrlo con unas pocas líneas de código!!



Así que... NOS VEMOS EN REACT!!

ACADEMY  
by NUMEN