

# Asincronía

¿Qué es? ¿Para qué sirve?

# 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, de modo que el orden o flujo del programa es sencillo y fácil de observar en el código:

```
primera_funcion(); // Tarea 1: Se ejecuta primero
```

```
segunda_funcion(); // Tarea 2: Se ejecuta cuando termina primera_funcion()
```

```
tercera_funcion(); // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```



Lenguaje no bloqueante

# Lenguaje no bloqueante

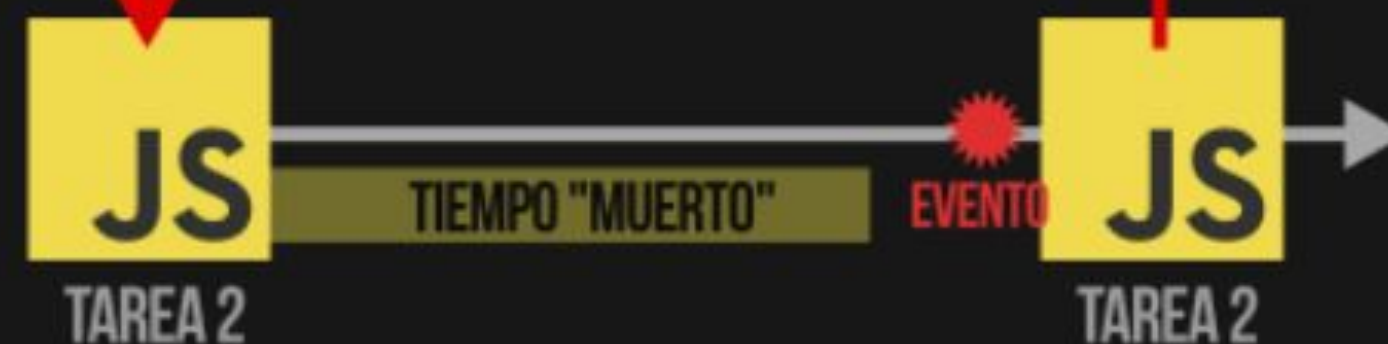
Cuando hablamos de Javascript, habitualmente nos referimos a él como un lenguaje no bloqueante. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.

Imaginemos que la `segunda_funcion()` del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. Si hablásemos de un lenguaje bloqueante, hasta que el usuario no haga click, Javascript no seguiría ejecutando las demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea

## tareas síncronas





## tareas síncronas



**tareas asíncronas pendientes**

**aún no se ha hecho click**

**ya se puede realizar**





Javascript es un lenguaje **no bloqueante**, lo que hará es **mover esa tarea** a una lista de **tareas pendientes** a las que irá «prestándole atención» a medida que **lo necesite**







Asincronía



Pueden existir muchas tareas que se cumplan y muchas que no y que no sepamos cuanto tiempo van a necesitar para ello necesitamos **promesas**







En javascript no todas las  
tareas son asíncronas

# Tareas asíncronas

Un `fetch()` a una URL para obtener un archivo `.json`.

Un `play()` de un `.mp3` que creamos mediante un `new Audio()`.

Una tarea programada con `setTimeout()` que se ejecutará en el futuro.

Una comunicación desde Javascript a la API del sintetizador de voz del navegador.

Una comunicación desde Javascript a la API de un sensor del smartphone.

# Gestionar la asincronía

En Javascript existen varias formas de gestionar la asincronía, donde quizás las más populares son las siguientes

Método	Descripción	Tema
Mediante <code>callbacks</code>	Probablemente, la forma más clásica de gestionar la asincronía en Javascript.	<a href="#">Ver Callbacks</a>
Mediante <code>promesas</code>	Una forma más moderna y actual de gestionar la asincronía.	<a href="#">Ver Promesas</a>
Mediante <code>async/await</code>	Seguimos con promesas, pero con <code>async/await</code> añadimos azúcar sintáctico.	<a href="#">Ver Async/Await</a>
Mediante <code>top-level await</code>	Una variación de la anterior, donde no es necesario usar <code>async</code> en determinados contextos.	

# Callbacks

las funciones **callback** no son más que un tipo de funciones que se pasan por parámetro a otras funciones.

# Callbacks

Imaginemos el siguiente bucle tradicional para recorrer un Array:

```
const list = ["A", "B", "C"];
```

```
for (let i = 0; i < list.length; i++) {  
  console.log("i=", i, " list=", list[i]);  
}
```

En la variable `i` tenemos la posición del array que estamos recorriendo. Este valor irá desde 0 hasta 2, mientras que con `list[i]` accedemos a la posición del array para obtener el elemento, es decir, desde A hasta C.

# utilizando callbacks

Ahora veamos, como podemos hacer este mismo bucle utilizando el método `forEach()` del array al cuál le pasamos una función callback:

```
const list = ["A", "B", "C"];

function action(element, index) {

  console.log("i=", index, "list=", element);

}

list.forEach(action);
```

# utilizando callbacks

`forEach()` se va a ejecutar por cada uno de los elementos del array, y en cada iteración de dicha función callback, los parámetros `element` e `index` van a tener un valor especial

```
list.forEach((element, index) => {  
  
  console.log("i=", index, "list=", element)  
  
});
```



# setTimeout()

Podemos definir cuánto tiempo esperará una función a ejecutarse

# setTimeout()

la función nos exige dos parámetros:

La función callback a ejecutar

El tiempo time que esperará antes de ejecutarse

# setTimeout()

Simplemente, le decimos a `setTimeout()` que ejecute la función callback que le hemos pasado por primer parámetro cuando transcurran 2000 milisegundos (es decir, 2 segundos)

```
function action() {  
    console.log("He ejecutado la función");  
}  
  
setTimeout(action, 2000);
```

# Utilizando flechas...

Utilizando arrow functions se puede simplificar el callback y hacer mucho más «fancy» y legible:

```
setTimeout(() => {  
  console.log("He ejecutado la función");  
}, 2000);
```



¿Y si mezclamos asincronía  
con callbacks?



# Asincronía y callbacks

Las funciones callback pueden utilizarse como un primer intento de manejar la asincronía en un programa.

```
function doTask(number, callback) {  
  /* Código de la función */  
}
```

```
doTask(42, function(err, result) {  
  /* Trabajamos con err o result según nos interese */  
});
```

```
const doTask = (iterations, callback) => {
  const numbers = [];

  for (let i = 0; i < iterations; i++) {
    const number = 1 + Math.floor(Math.random() * 6);
    numbers.push(number);
    if (number === 6) {
      callback({
        error: true,
        message: "Se ha sacado un 6"
      });
      return;
    }
  }

  /* Termina bucle y no se ha sacado 6 */
  return callback(null, {
    error: false,
    value: numbers
  });
}
```

¿Jugamos a los dados?



# Promesas

¿Qué son? ¿Para qué sirven?





# Promesas en Javascript

Como su propio nombre indica, una promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas, una promesa puede cumplirse o no.

# Promesas en Javascript

Las promesas en Javascript se representan a través de un objeto, y cada promesa estará en un estado concreto: pendiente, aceptada o rechazada.

Métodos	Descripción
<code>.then(FUNCTION resolve)</code>	Ejecuta la función callback <code>resolve</code> cuando la promesa se cumple.
<code>.catch(FUNCTION reject)</code>	Ejecuta la función callback <code>reject</code> cuando la promesa se rechaza.
<code>.then(FUNCTION resolve, FUNCTION reject)</code>	Método equivalente a las dos anteriores en el mismo <code>.then()</code> .
<code>.finally(FUNCTION end)</code>	Ejecuta la función callback <code>end</code> tanto si se cumple como si se rechaza.

# Haciendo cumplir una promesa

La forma general de consumir una promesa es utilizando el `.then()` con un sólo parámetro, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla

```
fetch("/robots.txt").then(function(response) {  
    /* Código a realizar cuando se cumpla la promesa */  
});
```



¿Y si mezclamos todo?

# new Promise()

Al new Promise() se le pasan por parametros una función y un callback

El primer callback, resolve, lo utilizaremos cuando se cumpla la promesa.

El segundo callback, reject, lo utilizaremos cuando se rechace la promesa.

```
const doTask = (iterations) => {  
  return new Promise((resolve, reject) => {  
    const numbers = [];  
  
    for (let i = 0; i < iterations; i++) {  
      const number = 1 + Math.floor(Math.random() * 6);  
      numbers.push(number);  
      if (number === 6) {  
        reject({  
          error: true,  
          message: "Se ha sacado un 6"  
        });  
      }  
    }  
  
    resolve({  
      error: false,  
      value: numbers  
    });  
  })  
}
```

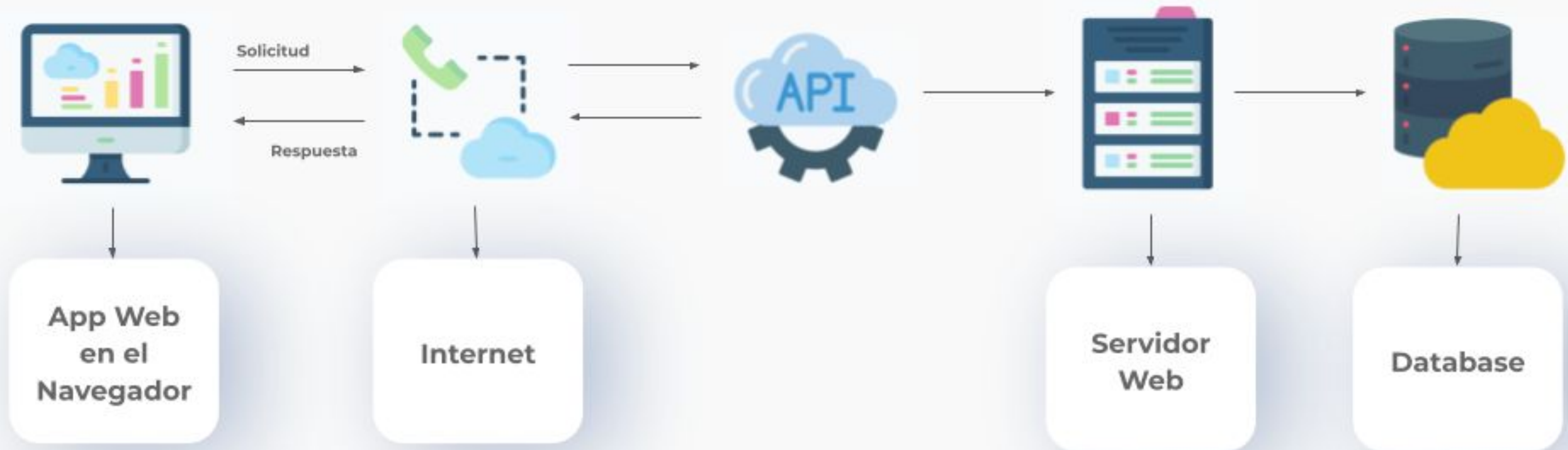


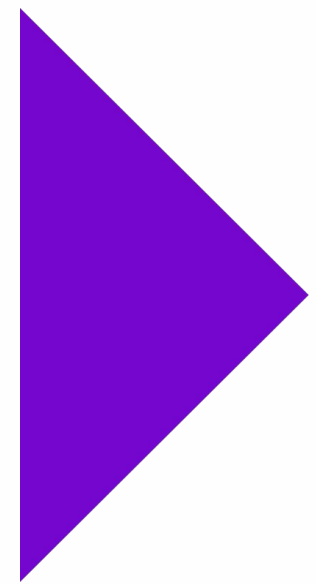
# ¿Qué es una API?

Application Programming Interfaces, que en español significa interfaz de programación de aplicaciones. Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.



# ¿Como funcionan las API's?



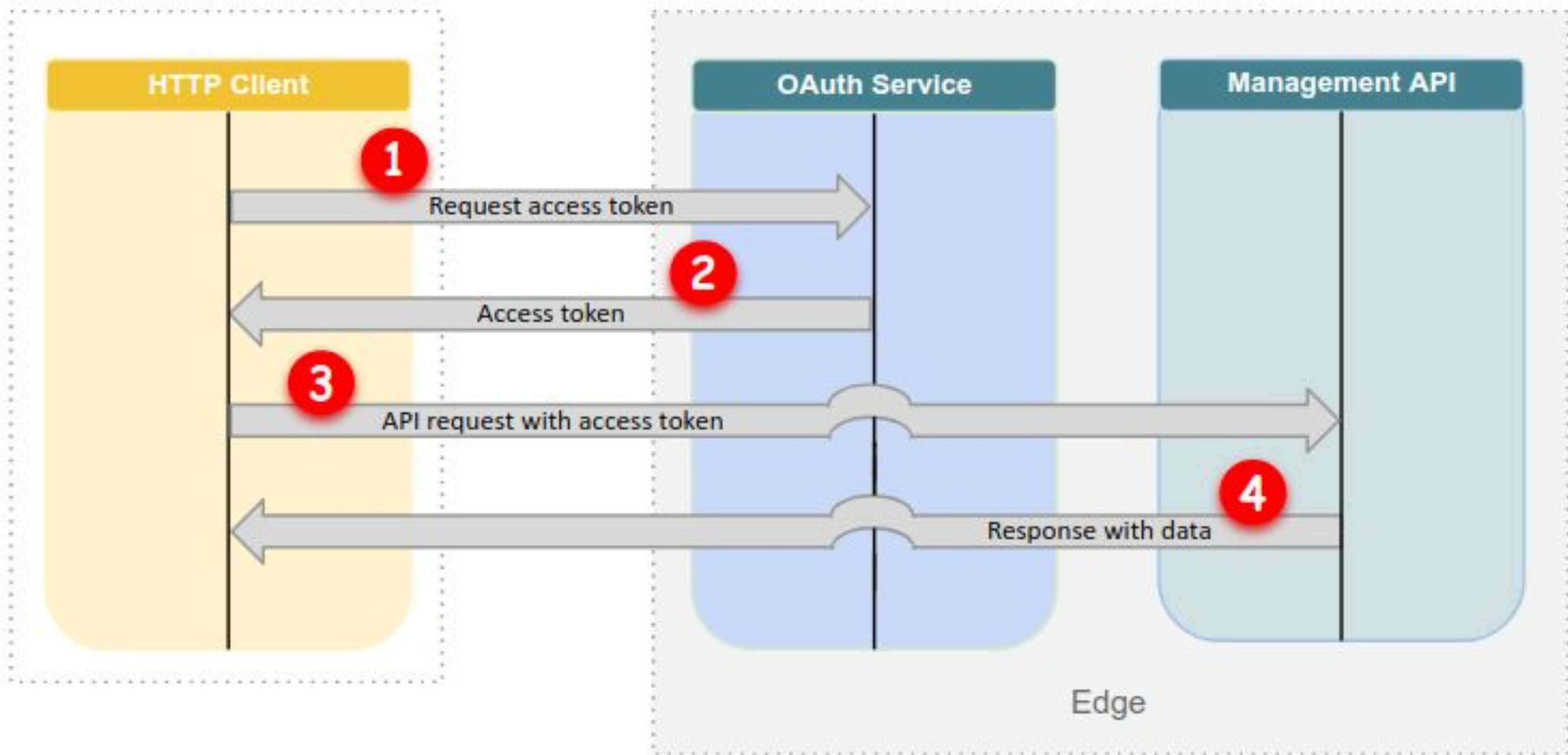


# Ojo... cada API tiene sus requisitos

Para que las distintas aplicaciones se conecten entre sí es necesario  
que se envíen ciertos requisitos...

**¿Si alguien te enviara un  
mensaje pidiendote la clave  
bancaria... se la pasarías?**

Bueno una API piensa igual que vos



# API errores & respuestas comunes

**200**

Ejecución exitosa, todo OK

**400**

El pedido era correcto, pero falló en un parámetro

**401**

No nos autenticamos correctamente

**403**

Prohibido, es decir, no tenemos permisos suficientes

**404**

Escribimos mal la API por lo tanto no existe

**429**

Demasiados pedidos, se saturó la API

**500**

Error del lado del servidor, nada que hacer de nuestro lado mas que levantar un ticket y esperar...

# ¿Dudas?

