

JS

UT07: SOLICITUDES
DE RED

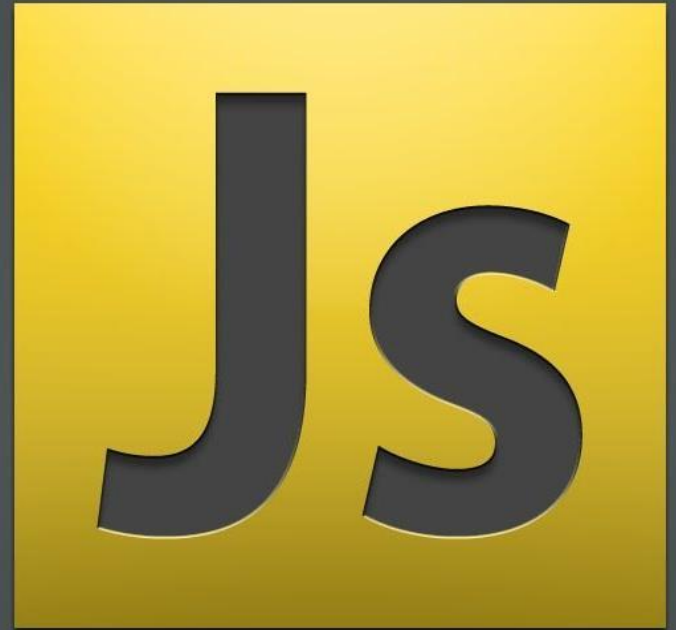
ÍNDICE

- 1.- Introducción a las APIs REST
- 2.- Promesas
- 3.- Promise API
- 3.- Fetch
- 4.- FormData
- 5.- Fetch: progreso de la descarga
- 6.- Fetch: Abort y Cross-Origin Requests
- 7.- Objetos URL



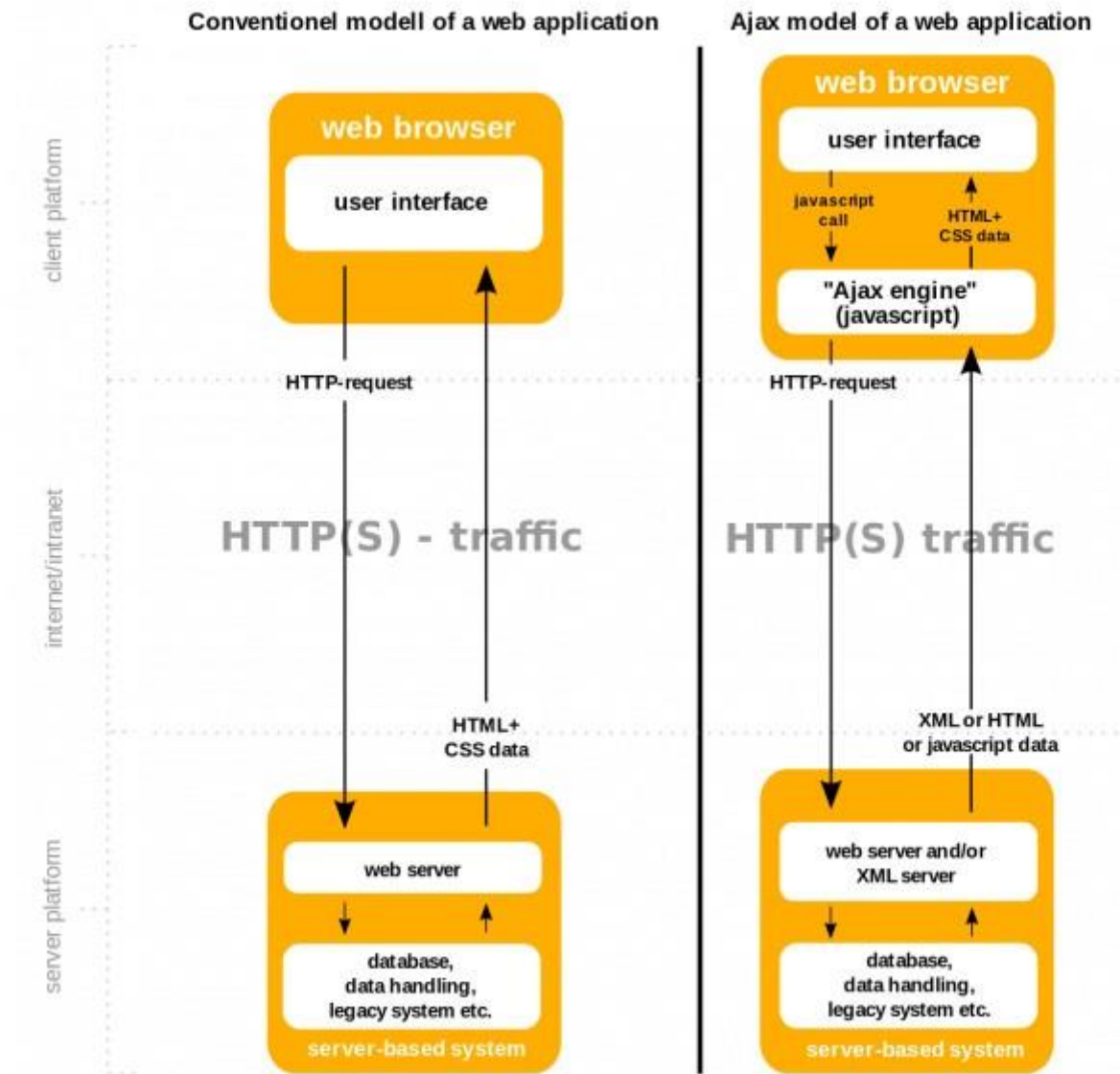
1

INTRODUCCIÓN A LAS APIS REST



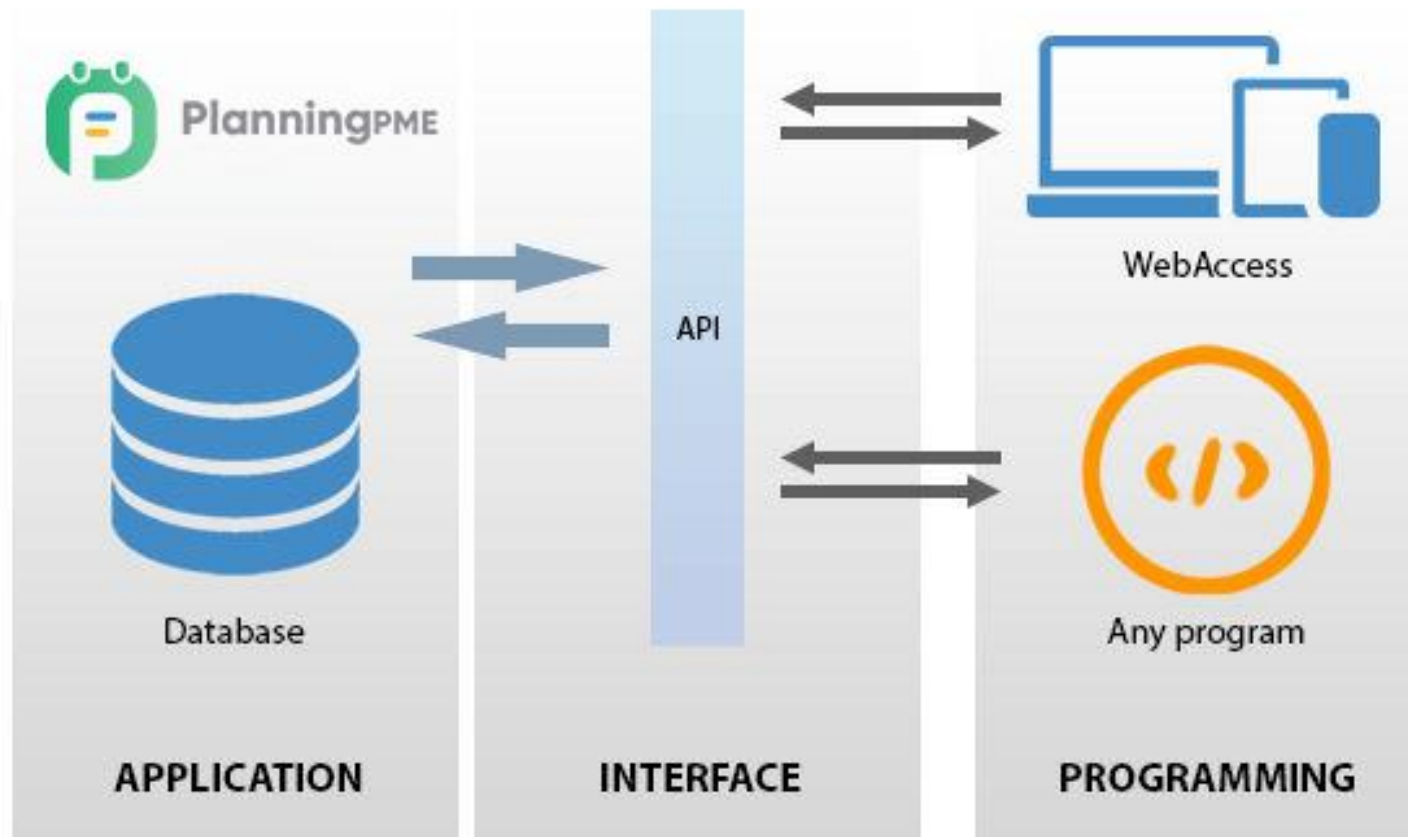
JavaScript permite realizar consultas a un servidor y cargar nueva informaci3n en la ṕgina.

AJAX (Asynchronous JavaScript And XML) es el nombre que se le asigna a las peticiones aśncronas realizadas desde JavaScript (aunque actualmente se usa JSON en lugar de XML).



Aunque las solicitudes desde JavaScript se pueden utilizar para descargar cualquier tipo de recurso, uno de sus principales usos es realizar consultas a APIs REST.

Una API (Application Programming Interface) es un conjunto de funcionalidades o recursos que expone un sistema para poder interactuar con él desde otro sistema, independientemente del lenguaje de programación o tecnología de cada uno de ellos.



REST (***Representational State Transfer***) es un estilo de arquitectura del software para comunicaciones cliente servidor apoyado en el protocolo HTTP.

REST se basa en las URLs, métodos HTTP y estados de respuesta.

URL (Uniform Resource Locator)

Una URL es la dirección que se le da a un recurso en la red. REST redefine este concepto utilizándolo para identificar recursos pero también **asignándoles nombres representativos**.

Así, las consultas a la API son fácilmente **comprensibles**.

Por ejemplo:

<https://swapi.dev/api/people/1/>

https://api.twitter.com/2/users/:id/timelines/reverse_chronological

Métodos HTTP

Los métodos HTTP se utilizan para indicar qué se quiere hacer con un recurso determinado.

Se utilizan cuatro métodos principalmente, asociados con las operaciones CRUD:

GET: para obtener o leer un recurso.

PUT: actualiza o reemplaza un recurso

DELETE: elimina un recurso del servidor

POST: crea un recurso en el servidor

Estados de respuesta

El resultado de la consulta a la API se indica en el campo de estado de la respuesta HTTP.

Los estados definidos por el estándar HTTP son:

- 1xx Informational
- 2xx Success
- 3xx Redirection
- 4xx Client Error
- 5xx Server Error

Algunos ejemplos de estados utilizados en REST:

200 (OK): la operaci3n solicitada se ha realizado con 3xito

201 (Created): se ha creado el recurso con 3xito en el servidor

202 (Accepted): utilizada t́picamente para solicitudes que llevan un tiempo para procesar e indica que ha sido aceptada.

204 (No Content): usualmente en respuesta a solicitudes PUT, POST y DELETE para indicar que la API REST no devuelve ninǵn mensaje en el cuerpo del mensaje.

301 (Moved Permanently): indica que el modelo de la API ha sido rediseńado y ha cambiado la URI de acceso al recurso.

307 (Temporary Redirect): la API REST no procesará la solicitud del cliente. Este tendrá que volver a enviar la solicitud a la URI indicada en el cuerpo de la respuesta. Sin embargo, futuras solicitudes deberán seguir utilizando la URI original.

400 (Bad Request): código de error genérico cuando no se adapta ningún otro.

401 (Unauthorized): el usuario no ha facilitado el método de autenticación requerido por la API y no tiene acceso al recurso.

403 (Forbidden): el usuario no tiene permiso para acceder al recurso.

404 (Not Found): indica que la API REST no puede mapear la URI con un recurso, pero puede que sí pueda en un futuro, por lo que sí se permitirían futuras solicitudes.

405 (Method Not Allowed): el ḿtodo indicado en la solicitud no est́ permitido para ese recurso, aunque ś lo estarían otros ḿtodos. En la cabecera de la respuesta se suelen incluir los ḿtodos permitidos.

500 (Server Error): ćdigo genérico para indicar alǵn tipo de error en el servidor.

501 (Not Implemented): el servidor no reconoce la solicitud o el ḿtodo, pero probablemente seŕ una funcionalidad futura.

MORE
INFO



<https://restfulapi.net/http-status-codes/>

Para realizar consultas REST desde el navegador podemos utilizar el **addon de Firefox RESTClient**.

The image shows two overlapping screenshots. The background screenshot is the RESTClient interface, which includes a 'Request' section with a method dropdown set to 'GET' and a URL field containing 'http://www.example.com'. Below this is a 'Body' section with a text area. The foreground screenshot is the Firefox Add-ons page for the RESTClient extension. It features the extension's icon (a sun-like symbol), the title 'RESTClient, a debugger for RESTful web services.' by Chao ZHOU, and a star rating of 4.1 stars based on 145 reviews. A table on the right shows the distribution of star ratings: 5 stars (88), 4 stars (25), 3 stars (10), 2 stars (4), and 1 star (18).

RESTClient, a debugger for RESTful web services.
by Chao ZHOU

Rating	Count
5 Stars	88
4 Stars	25
3 Stars	10
2 Stars	4
1 Star	18

Alternativamente, se puede utilizar **Postman**, una aplicación de escritorio con muchas más opciones.

The image shows the Postman website on the left and the Postman desktop application on the right. The website has a navigation bar with links to Product, Pricing, Enterprise, Resources and Support, and Explore. Below the navigation bar is a search bar and a 'Sign In' button. A large orange button labeled 'Sign Up for Free' is also present. The desktop application interface shows a workspace for 'Twitter API v2' with a collection of requests. The selected request is 'Single Tweet' with a GET method and the URL 'https://api.twitter.com/2/tweets/{id}'. The 'Params' tab is active, showing a query parameter 'id' with the value '1403216129661628420'. The 'Body' tab shows a JSON response with a 'data' field containing a tweet object. The 'Documentation' panel on the right provides details about the endpoint and its parameters.

Debu _
APIs together

Over 20 million developers use Postman. Get started by signing up or downloading the desktop app.

[Sign Up for Free](#)

Download the desktop app

What is Postman?

Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so you can create better APIs—faster.





Ejemplo de consulta a StarWars API

Authentication ▾ Headers ▾ View ▾

Favorites ▾ Data migration RESTClient

[-] Request

Method GET ▾ URL  https://swapi.dev/api/people/1  ▾

SEND

Body

Request Body

[-] Response

Headers Response Preview

Status Code: 200 OK
allow: GET, HEAD, OPTIONS
content-type: application/json
date: Wed, 28 Dec 2022 07:56:35 GMT
etag: "ee398610435c328f4d0a4e1b0d2f7bbc"
server: nginx/1.16.1
strict-transport-security: max-age=15768000
vary: Accept, Cookie
x-firefox-spdy: h2
x-frame-options: SAMEORIGIN

[-] Response

Headers Response Preview


```
1 {  
2   "name": "Luke Skywalker",  
3   "height": "172",  
4   "mass": "77",  
5   "hair_color": "blond",  
6   "skin_color": "fair",  
7   "eye_color": "blue",  
8   "birth_year": "1988Y",  
9   "gender": "male",  
10  "homeworld": "https://swapi.dev/api/planets/1/",  
11  "films": ["https://swapi.dev/api/films/1/", "https://swapi.dev/api/films/2/", "https://swapi.dev/api/films/3/", "https://swapi.dev/api/films/6/"],  
12  "species": [],  
13  "vehicles": ["https://swapi.dev/api/vehicles/14/", "https://swapi.dev/api/vehicles/30/"],  
14  "starships": ["https://swapi.dev/api/starships/12/", "https://swapi.dev/api/starships/22/"],  
15  "created": "2014-12-09T13:50:51.644000Z",  
16  "edited": "2014-12-20T21:17:56.891000Z",  
17  "url": "https://swapi.dev/api/people/1/"
```

[-] Curl1

Command



```
curl -X GET -k -i 'https://swapi.dev/api/people/1'
```

UT07: SOLICITUDES DE RED | FETCH | XXXX

 Authentication ▾ Headers ▾ View ▾

Favorites ▾ Data migration RESTClient

[-] Request

Method GET ▾ URL  https://swapi.dev/api/people/1251  ▾

SEND

Body

Request Body

[-] Response

Headers



Response

Preview

Status Code	: 404 Not Found
allow	: GET, HEAD, OPTIONS
content-type	: application/json
date	: Wed, 28 Dec 2022 07:58:30 GMT
etag	: "8bee5c3ad44d6c57f19e49e8e76ee09e"
server	: nginx/1.16.1
vary	: Accept, Cookie
x-firefox-spdy	: h2
x-frame-options	: SAMEORIGIN

[-] Curl

Command

```
curl -X GET -k -i 'https://swapi.dev/api/people/1251'
```

Algunas APIs permiten pasar parámetros en la URL

Searching

All resources support a `search` parameter that filters the set of resources returned. This allows you to make queries like:

```
https://swapi.dev/api/people/?search=r2
```

All searches will use case-insensitive partial matches on the set of search fields. To see the set of search fields for each resource,


[-] Request

Method

GET

▼

URL

 https://swapi.dev/api/people?search=luke

Body

Request Body

[-] Response

Headers

Response

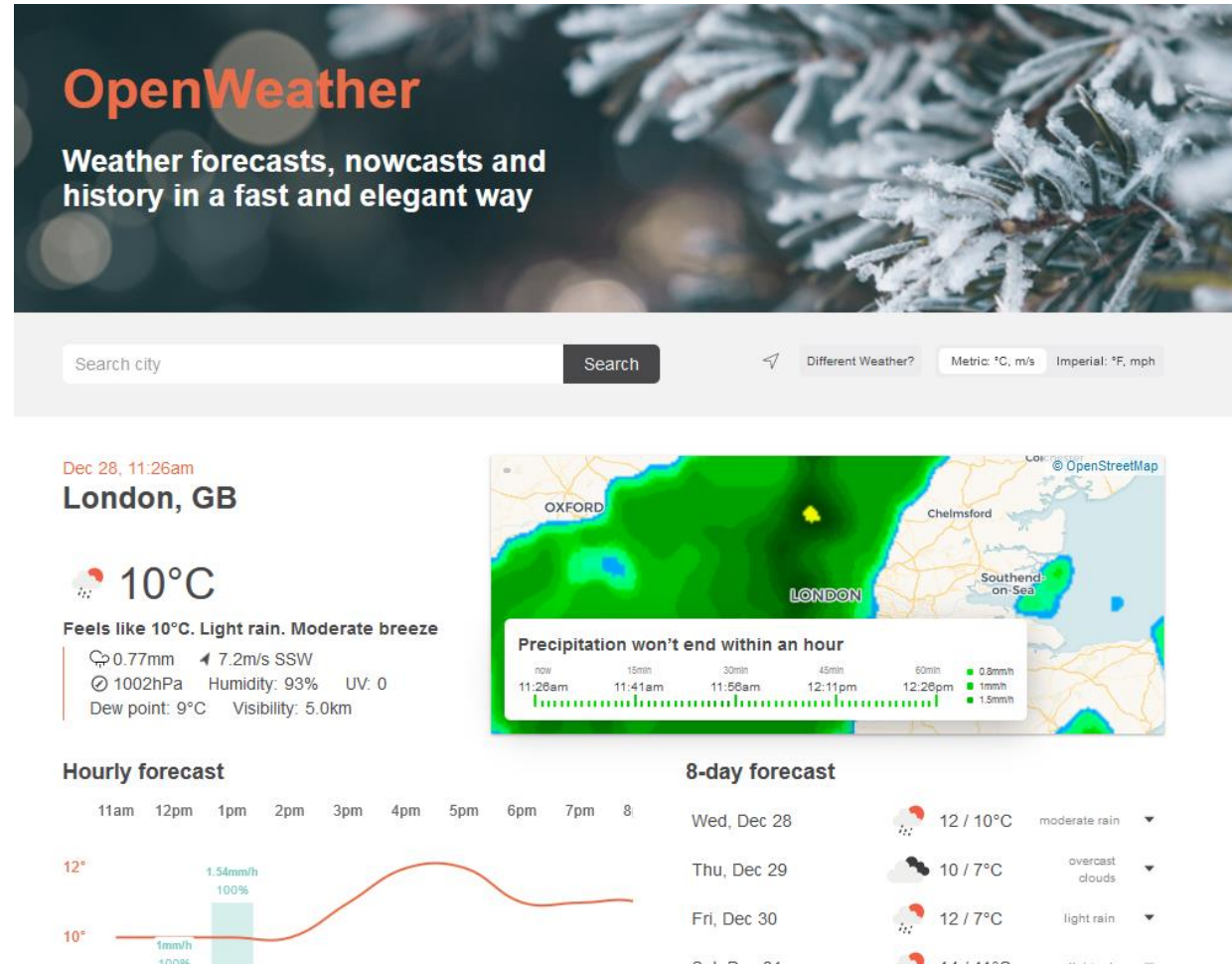
Preview

```
1 {
2   "count": 1,
3   "next": null,
4   "previous": null,
5   "results": [{
```

La mayoría de las APIs requieren algún tipo de autenticación, usualmente mediante una **API Key**.

Ejemplo:

OpenWeather



Primero creamos una cuenta

Create New Account

We will use information you provided for management and administration purposes, and for keeping you informed by mail, telephone, email and SMS of other products and services from us and our partners. You can proactively manage your preferences or opt-out of communications with us at any time using [Privacy Centre](#). You have the right to access your data held by us or to request your data to be deleted. For full details please see the OpenWeather [Privacy Policy](#).

☒ I am 16 years old and over


☐ I agree with [Privacy Policy](#), [Terms and conditions of sale](#) and [Websites terms and conditions of use](#)


I consent to receive communications from OpenWeather Group of Companies and their partners:

☐ System news (API usage alert, system update, temporary system shutdown, etc)

☐ Product news (change to price, new product features, etc)

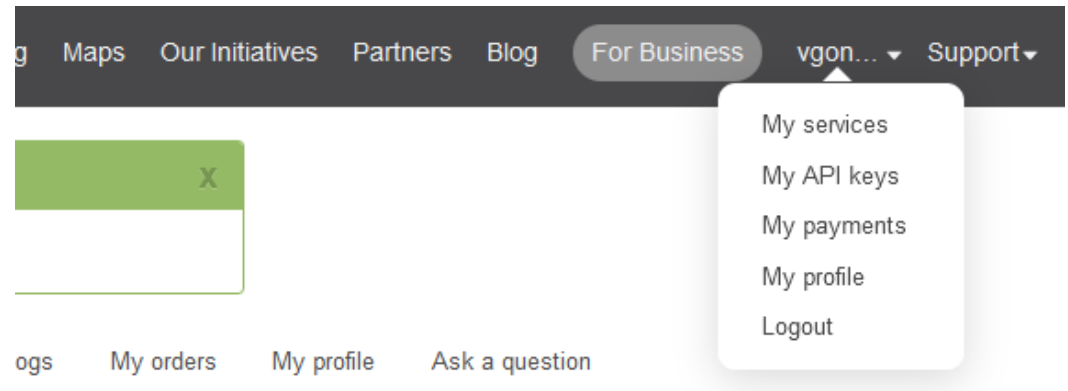
☐ Corporate news (our life, the launch of a new service, etc)

 No soy un robot


reCAPTCHA
[Privacidad](#) - [Términos](#)

Create Account

Y ya podremos acceder a la sección **My API Keys**





I weather for any location

hine, has allowed us to enhance the data in the [Historical](#)

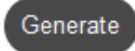
En el caso de esta API puedes tener diversas API Keys las cuales se gestionarán desde aquí.

New Products Services **API keys** Billing plans Payments Block logs My orders My profile Ask a question

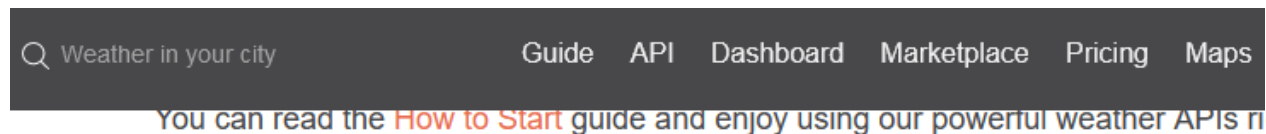
You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key	Name	Status	Actions
ae6ba5815beaa9[REDACTED]	Default	Active	 

Create key

API key name 

Falta saber cómo hay que enviar la API Key en las consultas, por lo que acudimos a la documentación.



Current & Forecast weather data collection

Current Weather Data

[API doc](#)[Subscribe](#)

- Access current weather data for any location including over 200,000 cities
- We collect and process weather data from different sources such as global and local weather models, satellites, radars and a vast network of weather stations
- JSON, XML, and HTML formats
- Included in both free and paid subscriptions

Hourly Forecast 4 days

[API doc](#)[Subscribe](#)

- Hourly forecast is available for 4 days
- Forecast weather data for 96 timestamps
- JSON and XML formats
- Included in the Developer, Professional and Enterprise subscription plans

En el primer ejemplo ya podemos ver que la API Key simplemente se pasa como parámetro en la URI

API call

```
https://api.openweathermap.org/data/2.5/weather?lat={lat}&  
lon={lon}&appid={API key}
```



Vamos a probarla. Necesitamos indicar las coordenadas geográficas de la ubicación a consultar, por ahora las buscamos por internet, pero podríamos obtenerlas con otra consulta a **Geocoding API** tal como se indica en la documentación.

Parameters

lat, lon

required

Geographical coordinates (latitude, longitude). If you need the geocoder to automatic convert city names and zip-codes to geo coordinates and the other way around, please use our [Geocoding API](#).

Coordenadas de León (España)

Aquí podrás obtener las coordenadas geográficas de León, España, de mane grados decimales para que puedas localidar León, España, en Google Maps.

Coordenadas geográficas de León, España, en grados decimales:

- Longitud: -5.5703200
- Latitud: 42.6000300

[-] Request

Method

GET

URL

<https://api.openweathermap.org/data/2.5/weather?lat=42.60&lon=-5.57&appid=ae6ba58159>

Body

Request Body

[-] Response

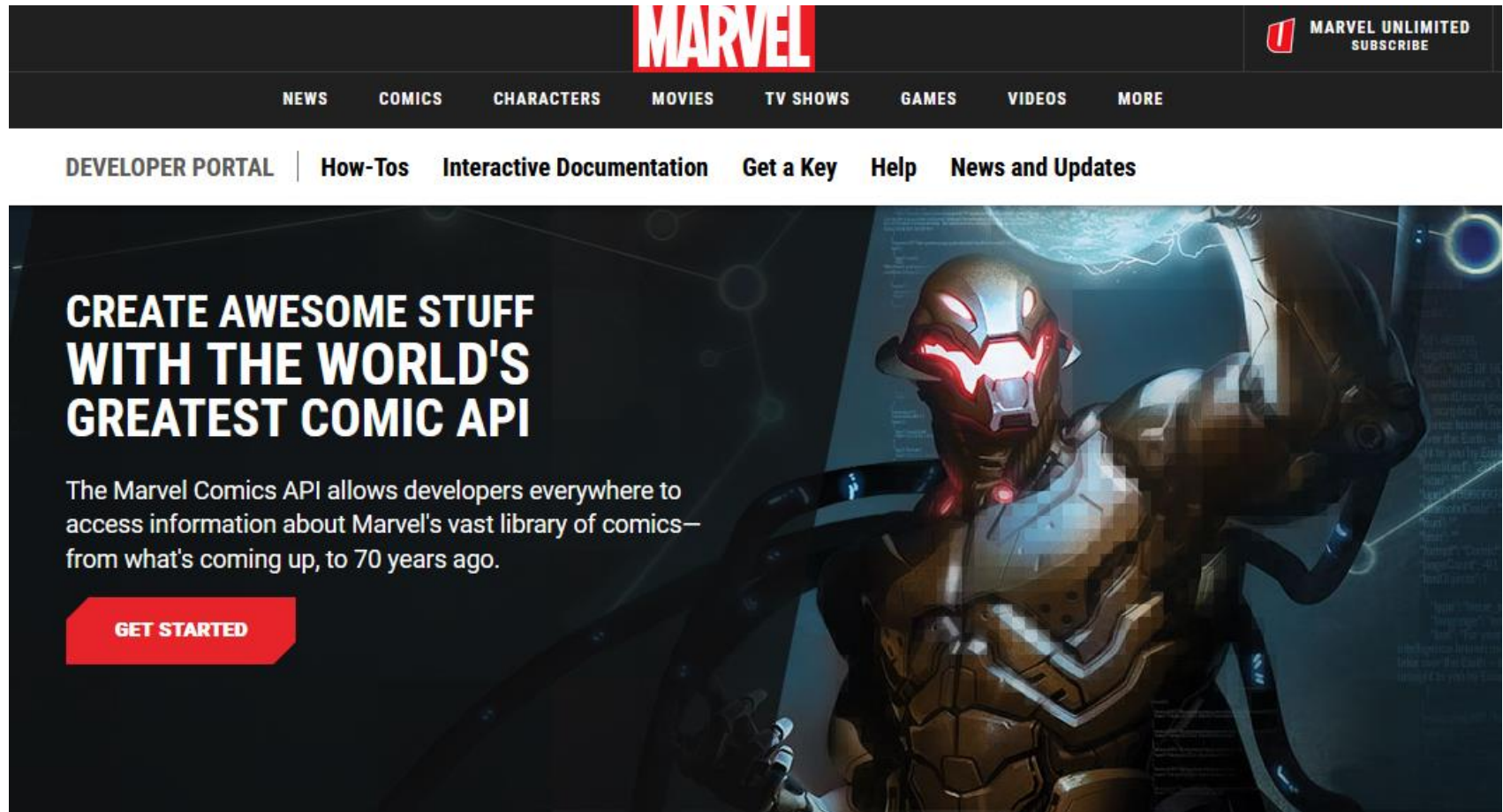
Headers

Response

Preview

```
1 {  
2   "coord": {  
3     "lon": -5.5667,  
4     "lat": 42.6  
5   },  
6   "weather": [{  
7     "id": 804,  
8     "main": "Clouds",  
9     "description": "overcast clouds",  
10    "icon": "04d"  
11  }  
12 }
```

Otro ejemplo de API: Marvel API



The image shows the Marvel Developer Portal homepage. At the top, there is a dark navigation bar with the Marvel logo in the center. To the right of the logo is a red button that says "MARVEL UNLIMITED SUBSCRIBE". Below the navigation bar is a horizontal menu with links: NEWS, COMICS, CHARACTERS, MOVIES, TV SHOWS, GAMES, VIDEOS, and MORE. Below this menu is a white bar with the text "DEVELOPER PORTAL" followed by a vertical line and then "How-Tos", "Interactive Documentation", "Get a Key", "Help", and "News and Updates". The main content area has a dark background with a large image of Iron Man on the right. On the left, there is white text that reads "CREATE AWESOME STUFF WITH THE WORLD'S GREATEST COMIC API". Below this text is a paragraph: "The Marvel Comics API allows developers everywhere to access information about Marvel's vast library of comics—from what's coming up, to 70 years ago." At the bottom left of this section is a red button that says "GET STARTED".

MARVEL

MARVEL UNLIMITED
SUBSCRIBE

NEWS COMICS CHARACTERS MOVIES TV SHOWS GAMES VIDEOS MORE

DEVELOPER PORTAL | How-Tos Interactive Documentation Get a Key Help News and Updates

**CREATE AWESOME STUFF
WITH THE WORLD'S
GREATEST COMIC API**

The Marvel Comics API allows developers everywhere to access information about Marvel's vast library of comics—from what's coming up, to 70 years ago.

GET STARTED



Password

SIGN IN

[Need help signing in?](#)

CREATE AN ACCOUNT

Read more about how to authorize referring domains in browser-based apps and web sites. »

La autenticación depende de si nuestra aplicación está en el cliente o el servidor.

Authentication for Client-Side Applications

Requests from client-side (browser-based) applications must originate from a pre-authorized web site or browser extension URL. You may add or edit your authorized domains in your API account panel. You may use the "*" wildcard to denote subdomains or paths. For example:

marvel.com - will authorize requests from Marvel.com but no subdomains of Marvel.com

developer.marvel.com - will authorize requests from developer.marvel.com

***.marvel.com** - will authorize requests from any Marvel.com subdomain as well as Marvel.com

***.marvel.com/apigateway** - will authorize requests from the apigateway path on any Marvel.com subdomain as well as Marvel.com

Authentication for Server-Side Applications

Server-side applications must pass two parameters in addition to the apikey parameter:

ts - a timestamp (or other long string which can change on a request-by-request basis)

hash - a md5 digest of the ts parameter, your private key and your public key (e.g. md5(ts+privateKey+publicKey))

For example, a user with a public key of "1234" and a private key of "abcd" could construct a valid call as follows:

`http://gateway.marvel.com/v1/public/comics?ts=1&apikey=1234&hash=ffd275c5130566a2916217b101f26150` (the hash value is the md5 digest of 1abcd1234)

En este caso las pruebas con RestClient no funcionarán, ya que lo interpreta como una aplicación del lado del servidor.

[-] Request

Method

GET

▼

URL

 http://gateway.marvel.com/v1/public/comics?apikey=91f5c853261777b74d9f7a884470b760

Body

Request Body

[-] Response

Headers

Response

Preview

1

```
{"code": "MissingParameter", "message": "You must provide a hash."}
```

Podŕamos generar el hash como se indica en la documentaci3n.

For example, a user with a public key of "1234" and a private key of "abcd" could construct a valid call as follows:

`http://gateway.marvel.com/v1/public/comics?ts=1&apikey=1234&hash=ffd275c5130566a2916217b101f26150` (the hash value is the md5 digest of 1abcd1234)

Uso la web <https://www.md5.cz/>

function md5()

Online generator [md5 hash](#) of a string

md5 ('70c4902f[REDACTED]4d9f7a884470b760)

hash darling, hash!

You are awesome! Here is your MD5 checksum:

[REDACTED]b780f9c0950a



[-] Request

Method GET

URL

<http://gateway.marvel.com/v1/public/comics?ts=1&apikey=9f15c695a1b0760&hash=edb4e2c193ff9550a>

Body

Request Body

[-] Response

Headers

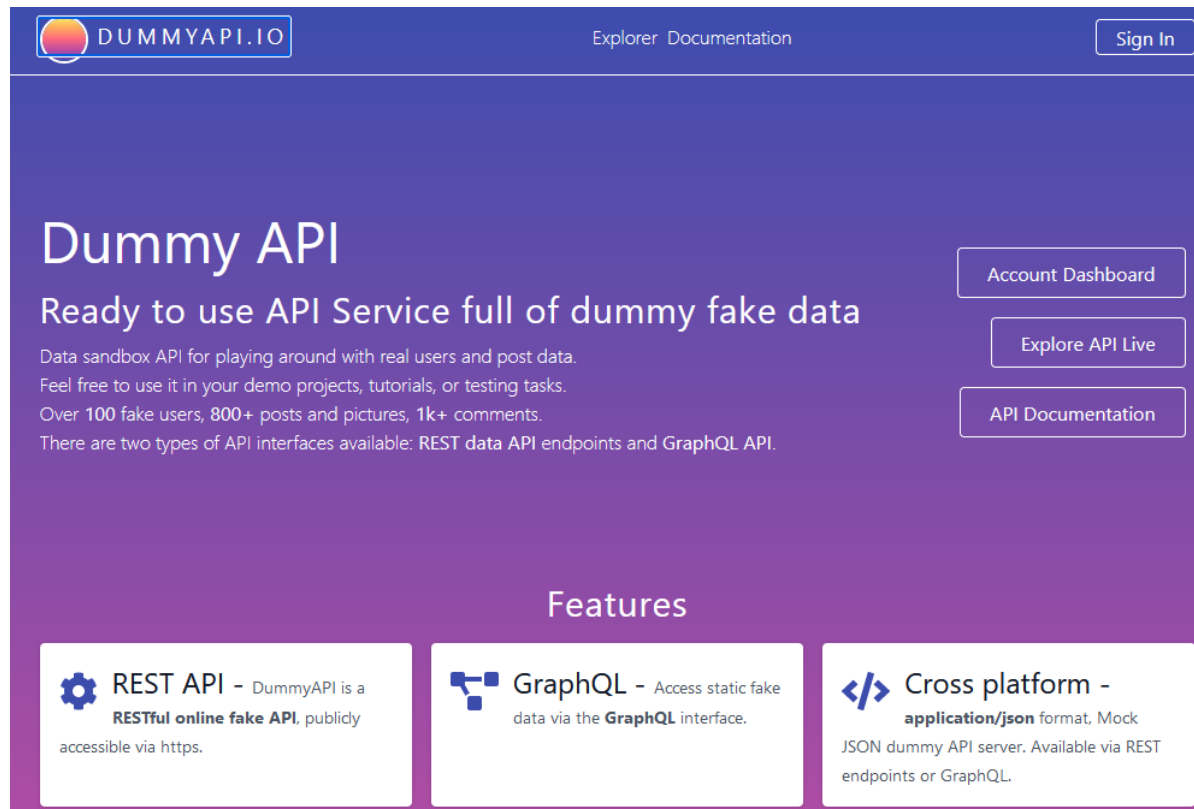
Response

Preview

```
1 {
2   "code": 200,
3   "status": "Ok",
4   "copyright": "© 2022 MARVEL",
5   "attributionText": "Data provided by Marvel. © 2022 MARVEL",
6   "attributionHTML": "<a href='\"http://marvel.com\">Data provided by Marvel. © 2022 MARVEL</a>",
7   "etag": "aa275b9eab52a5c839c2115837c34d3a77405c43",
8   "data": {
9     "offset": 0,
10    "limit": 20,
11    "total": 53775,
```


En algunas APIs la autenticación se realiza mediante las cabeceras HTTP.

Ejemplo: Dummy API



Generamos un API Id



Account

User App ID

Here you can generate personal **APP token** for API.

- ✓ It is necessary to calculate API usage statistic to avoid automatic scrapers.
- ✓ Define a personal environment for each user. Where you can make CRUD operation on entities(user/post/comments etc.). This changes will be visible only for you.
- ✓ Use App ID value to set **app-id** header for all request to API.

App IDs list

Dec 29 2022 09:04:04	
63ad49f41cc3[REDACTED]1	

Generate App ID

Sponsor Account Setup

Free account is limited to 500 calls per day.

In case you want to exceed this limit, you can became our **patron** on **patreon**. We have a cheapest possible tire **1\$** per month, so you can decide a price by yourself.

BECOME A PATRON

UDP! Free to use with no limits. While new API version is in beta testing!

Si analizamos la documentación veremos que se debe indicar en cada consulta mediante la **cabecera app-id**.

Headers

It is required to set **app-id** Header for each request.

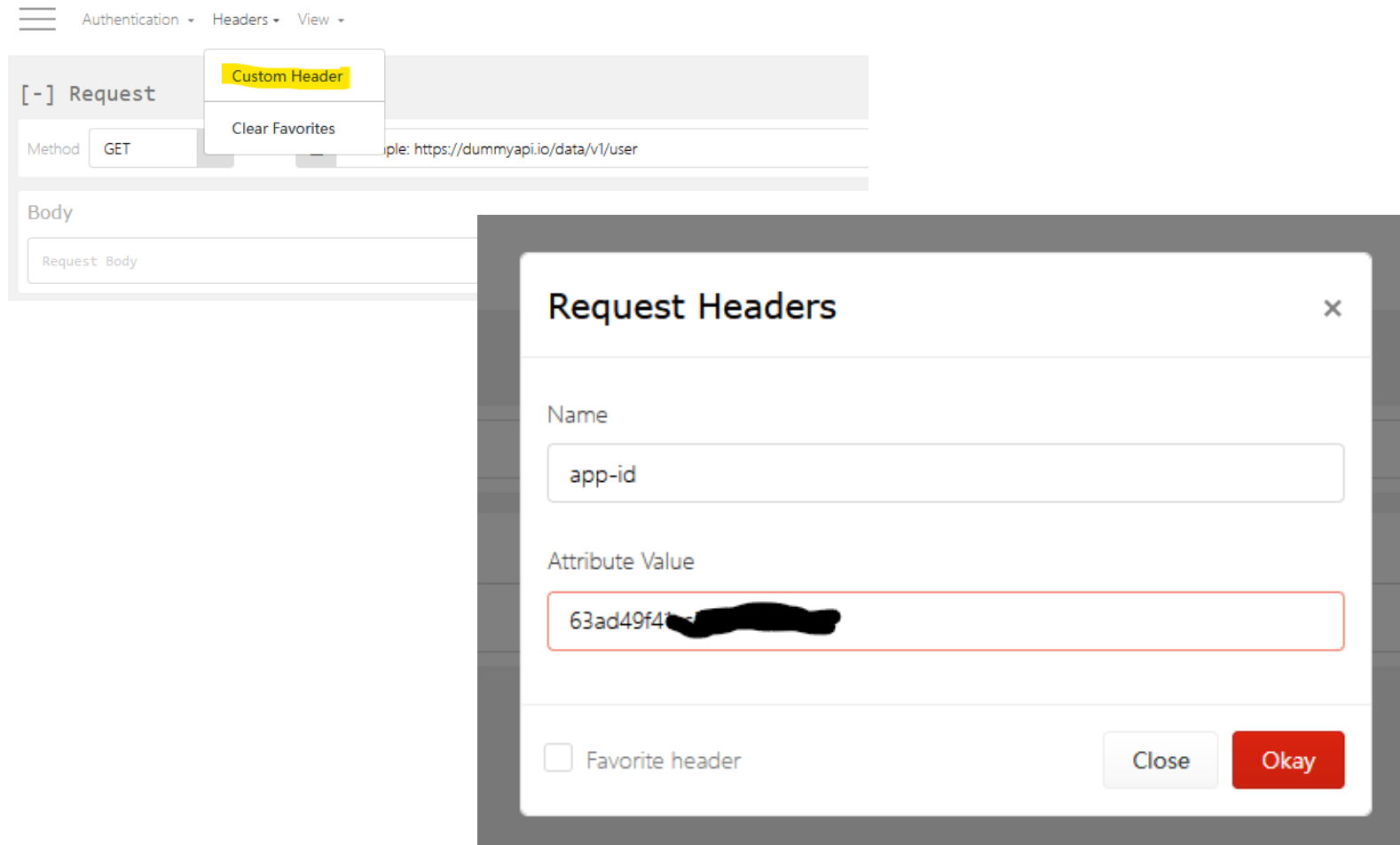
It helps us to determine your personal environment. So only you can access data that were created or update.

You can get personal App ID value on your account page.

You can have as much App ID as you want and use it in parallel(for different projects, envs etc).

Example: app-id: 0JyYiOQXQQr5H9OEn21312

Creamos la cabecera en *Headers* -> *Custom header*



[-] Request

Method

GET



URL

<https://dummyapi.io/data/v1/user>

Headers

app-id: 63ad49



Body

Request Body

[-] Response

Headers

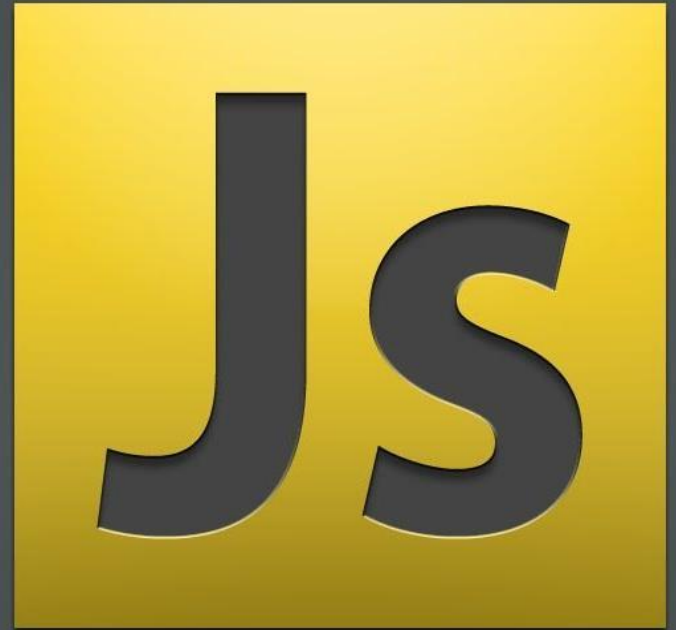
Response

Preview

```
1 {  
2   "data": [{  
3     "id": "60d0fe4f5311236168a109ca",  
4     "title": "ms",  
5     "firstName": "Sara",  
6     "lastName": "Andersen",  
7     "picture": "https://randomuser.me/api/portraits/women/58.jpg"
```

2

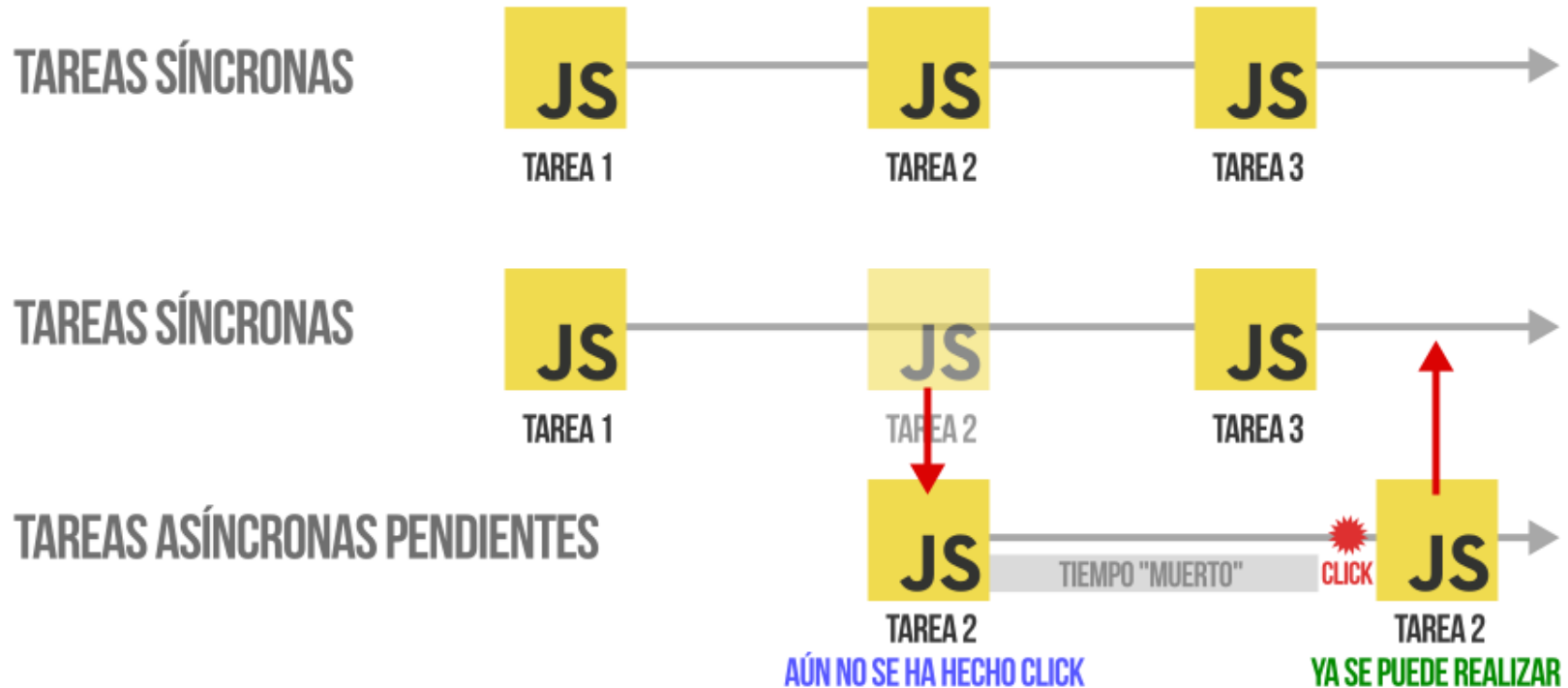
PROMESAS



Es muy frecuente que las páginas web realicen operaciones asíncronas durante su funcionamiento.

Una **acción asíncrona** es una tarea que no se realiza inmediatamente, sino que se puede demorar un poco (por ejemplo, la carga de un fichero o script).

El programa no espera a que finalice, sino que sigue ejecutándose y ya acabará cuando sea.



Ejemplo: carga de un script de forma dinámica desde JavaScript

```
function loadScript(src) {  
    let script = document.createElement('script');  
    script.src = src;  
    document.head.append(script);  
}  
  
loadScript('/my/script.js');  
console.log("Esta línea se ejecuta sin esperar a que  
cargue");
```

Si en el ejemplo anterior llamara a una función que está en el script descargado, obtendría un error, ya que el navegador no habrá tenido tiempo para descargar el script.

```
function loadScript(src) {  
    let script = document.createElement('script');  
    script.src = src;  
    document.head.append(script);  
}  
  
loadScript('/my/script.js');  
myFunc();    // Esta función está en el script. ERROR
```

¿Cómo hacemos entonces si queremos ejecutar un código que dependa de que haya finalizado una acción asíncrona? La solución son los **callbacks**.

Un callback es una función, generalmente anónima, que se pasa como argumento a la función asíncrona para que se ejecute cuando haya finalizado la acción.

Utilizando *callbacks*, el ejemplo anterior quedaría así

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  script.onload = () => callback(script);  
  document.head.append(script);  
}  
  
loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {  
  alert(`Genial, el script ${script.src} está cargado`);  
  alert( _ ); // _ es una función declarada en el script  
cargado  
});
```

Es muy común cargar elementos secuencialmente, para ello habría que poner un callback dentro del otro.

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  script.onload = () => callback();  
  document.head.append(script);  
}  
  
// Solución: ponemos la segunda llamada dentro del callback  
loadScript( urlScript1, function() {  
  console.log('1er callback. Cargamos el segundo script')  
  loadScript( urlScript2, function() {  
    console.log('2º callback');  
  } )  
} )
```

En los ejemplos anteriores no se han tenido en cuenta los errores, ¿qué pasa si el script da un error al cargar?

Esto se puede gestionar **enviando un parámetro de error a la función de *callback*.**

Por norma general, el primer parámetro de la función se reserva para el error y el segundo para enviar los datos en caso de resultado exitoso. A esto se le llama **callback error primero.**

```
function loadScript(src, callback) {  
  let script = document.createElement('script');  
  script.src = src;  
  
  // El evento error se dispara si hay algún error al cargar el recurso  
  script.onload = () => callback( null, script );  
  script.onerror = () => callback( new Error(`Error: ${src}`) );  
  
  document.head.append(script);  
}  
  
// Solución: ponemos la segunda llamada dentro del callback  
loadScript( urlScript1, function( error, script ) {  
  if (error) {  
    console.log('Aquí manejaríamos el error del script 1');  
  } else {  
    console.log('El script 1 se ha cargado con éxito')  
  }  
} );
```

Aunque los callbacks son útiles cuando hay una o dos acciones asíncronas, si necesitamos ejecutar múltiples acciones asíncronas de forma secuencial el código se puede complicar mucho.

Esto es lo que se llama **callback hell** o **pirámide infernal**.


```
loadScript( urlScript1, function( error, script ) {  
  if (error) {  
    console.log('Manejo de error del script1');  
  } else {  
    console.log('El script 1 se ha cargado con éxito')  
    loadScript( urlScript2, function( error, script ) {  
      if (error) {  
        console.log("Manejo de error del script 2");  
      } else {  
        console.log("El script 2 se ha cargado con éxito");  
        loadScript( urlScript3, function( error, script ) {  
          if (error) {  
            console.log("Manejo de error del script 3");  
          } else {  
            console.log("El script 3 se ha cargado con éxito");  
            loadScript( urlScript4, function( error, script ) {  
              if (error) {  
                console.log("Manejo de error del script 4");  
              } else {  
                console.log("El script 4 se ha cargado");  
              }  
            })  
          }  
        })  
      }  
    })  
  }  
});
```

La solución para evitar todos los problemas derivados del uso de *callbacks* son las **promesas**, un mecanismo de JavaScript introducido en ES6 (2015) que es específico para gestionar eventos asíncronos.

La forma de declarar una promesa es la siguiente:

```
let promise = new Promise( function(resolve, reject) {  
    // El código del ejecutor  
});
```

Como se puede ver, una promesa tiene tres partes:

- **Ejecutor:** es el código asíncrono que se va a ejecutar. En el ejemplo anterior con *callbacks* sería la carga del script.
- **resolve:** es la función que se pasa automáticamente al ejecutar y a la que hay que invocar desde dentro de este código cuando el evento asíncrono haya concluido con éxito.
- **reject:** de forma análoga a *resolve*, esta función debe ser invocada si hay un error con el evento asíncrono.

Una vez creada la promesa, el código se seguirá ejecutando.

Para indicar qué hay que hacer cuando la promesa se haya cumplido hay que utilizar la función **then()**.

De forma análoga se puede indicar el código a ejecutar cuando una promesa ha finalizado con error mediante la función **catch()**.

El código anterior quedaría de la siguiente forma si utilizáramos promesas.

```
const promise = new Promise( function(resolve, reject) {  
  // Este es el código que se ejecuta de forma asíncrona  
  let script = document.createElement('script');  
  script.src = url;  
  document.head.append(script);  
  
  // Cuando se ha ejecutado el código se invoca la función resolve()  
  // para indicar que la promesa se ha cumplido  
  script.onload = () => resolve();  
} );  
  
// Con el then indicamos qué hay que hacer una vez que se haya cumplido la  
promesa  
promise.then( () => {  
  console.log('Esto se ejecuta una vez que la promesa se haya cumplido.');
```

Y si también gestionamos errores:

```
const url = urlScript4;

const promise = new Promise( function(resolve, reject) {
  let script = document.createElement('script');
  script.src = url;
  document.head.append(script);

  script.onload = () => resolve();
  script.onerror = () => reject();
} );

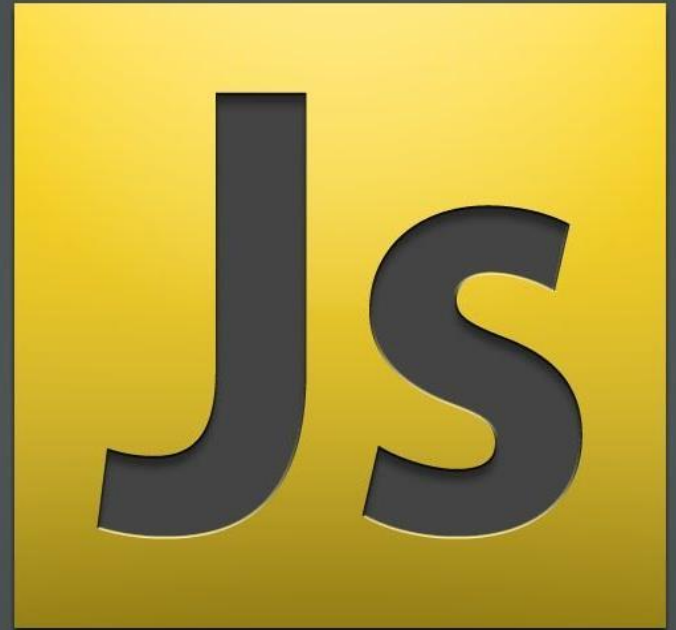
promise
  .then( () => {
    console.log('La promesa se ha cumplido.');
```

```
    console.log(promise);
  } )
  .catch( () => {
    console.log('La promesa NO SE HA CUMPLIDO.');
```

```
    console.log(promise);
  } );
```

3

PROMISE API



La clase Promise dispone de 6 ḿtodos est́ticos que pueden ser muy ́tiles cuando trabajamos con promesas.

- Promise.all
- Promise.allSettled
- Promise.race
- Promise.any
- Promise.resolve
- Promise.reject

Promise.all

Toma un iterable (usualmente un array de promesas) y devuelve una nueva promesa que es devuelta cuando todas las promesas listadas se resuelvan.

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)),
  new Promise(resolve => setTimeout(() => resolve(2), 2000)),
  new Promise(resolve => setTimeout(() => resolve(3), 1000))
]).then(alert);
```

Otro ejemplo

```
let urls = [  
  'https://api.github.com/users/iliakan',  
  'https://api.github.com/users/remy',  
  'https://api.github.com/users/jeresig'  
];  
  
// "mapeamos" cada url a la promesa de su fetch  
let requests = urls.map(url => fetch(url));  
  
// Promise.all espera hasta que todas la tareas est́en resueltas  
Promise.all(requests)  
  .then(responses => responses.forEach(  
    response => alert(`${response.url}: ${response.status}`)  
  ));
```

Promise.allSettled

Promise.all solo se resuelve si todas sus promesas finalizan con éxito.

Promise.allSettled solo espera a que todas las promesas se resuelvan sin importar sus resultados. En este caso devuelve un array que tiene:

```
{status:"fulfilled", value:result} para respuestas exitosas,  
{status:"rejected", reason:error} para errores.
```

```
let urls = [  
  'https://api.github.com/users/iliakan',  
  'https://api.github.com/users/remy',  
  'https://no-such-url'  
];  
  
Promise.allSettled(urls.map(url => fetch(url)))  
  .then(results => {  
    results.forEach((result, num) => {  
      if (result.status == "fulfilled") {  
        alert(`${urls[num]}: ${result.value.status}`);  
      }  
      if (result.status == "rejected") {  
        alert(`${urls[num]}: ${result.reason}`);  
      }  
    });  
  });
```

Promise.race

Similar a Promise.all, pero espera solamente por la primera respuesta y obtiene su resultado (o error)

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() =>
    resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() =>
    reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() =>
    resolve(3), 3000))
]).then(alert); // 1
```

Aquí se resolvería solo la primera promesa, por lo que el resto se ignorarían.

Promise.any

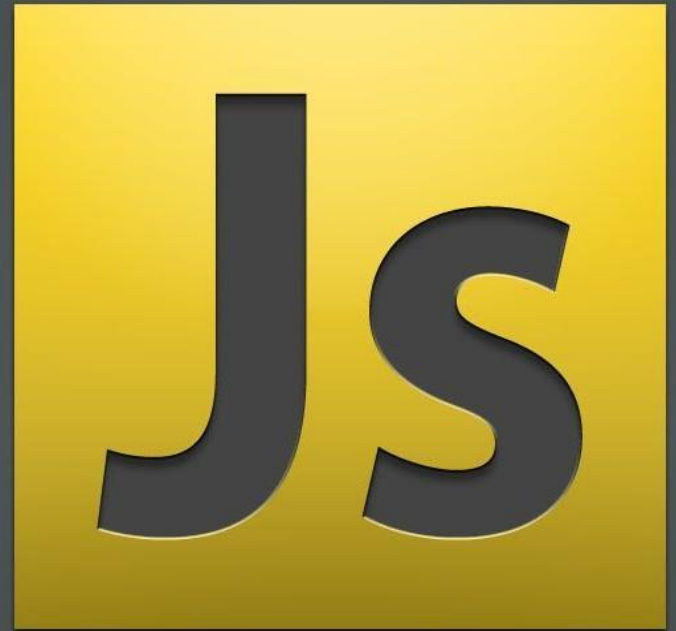
Similar al anterior, pero espera por la primera **promesa cumplida**.

Promise.resolve / Promise.reject

Permiten crear una promesa resuelta o rechazada respectivamente. Apenas son utilizados.

4

FETCH



Actualmente, el método utilizado en JavaScript para realizar peticiones de red es **fetch**, que ha reemplazado al antiguo XMLHttpRequest.

Su sintaxis básica es:

```
let promise = fetch(url, [options])
```

Donde *url* es la dirección URL a la que se desea acceder y *options* los parámetros opcionales, como puede ser el método a utilizar o los encabezados de la petición.

Este método **devuelve una promesa**.

Ejemplo de solicitud.

```
const url = 'https://swapi.dev/api/people/1';

async function getData() {
  console.log(url);
  let response = await fetch( url );
  console.log(response);
}
getData();
```

Algo importante es que la promesa devuelta por fetch resuelve la respuesta con un objeto de tipo **Response**, que **contiene los encabezados de la petición**.

```
▼ Response { type: "cors", url: "https://swapi.dev/api/people/1", redirected: false, status: 200, ok:
Headers(1), body: ReadableStream, bodyUsed: false }
  ► body: ReadableStream { locked: false }
    bodyUsed: false
  ▼ headers: Headers { "content-type" → "application/json" }
    ► <entries>
    ► <prototype>: HeadersPrototype { append: append(), delete: delete(), get: get(), ... }
    ok: true
    redirected: false
    status: 200
    statusText: "OK"
    type: "cors"
    url: "https://swapi.dev/api/people/1"
    ► <prototype>: ResponsePrototype { clone: clone(), arrayBuffer: arrayBuffer(), blob: blob(), ... }
```

Las propiedades más interesantes de este objeto son **status** y **ok**, que contienen el estado de la respuesta y un booleano que será true si el estado es 200 a 299.

- **1xx:** Mensaje informativo.
- **2xx:** Exito
 - 200 OK
 - 201 Created
 - 202 Accepted
 - 204 No Content
- **3xx:** Redirección
 - 300 Multiple Choice
 - 301 Moved Permanently
 - 302 Found
 - 304 Not Modified
- **4xx:** Error del cliente
 - 400 Bad Request
 - 401 Unauthorized
 - 403 Forbidden
 - 404 Not Found
- **5xx:** Error del servidor
 - 500 Internal Server Error
 - 501 Not Implemented
 - 502 Bad Gateway
 - 503 Service Unavailable

Si la respuesta es exitosa, debemos utilizar un segundo método para obtener el cuerpo de la respuesta.

Este método varía en función del formato de la misma.

- `response.text()`: en formato texto
- `response.json()`: la devuelve como un JSON
- `response.formData()`: la devuelve como objeto FormData
- `response.blob()`: objeto de tipo Blob (datos binarios)
- `response.arrayBuffer()`: representación de datos de bajo nivel (arrayBuffer)

Obteniendo el cuerpo de la solicitud con await.

```
const url = 'https://swapi.dev/api/people/1';

async function getData() {
  let response = await fetch( url, );
  let json = await response.json();
  console.log(json);
}
getData();
```

Alternativa usando promesas

```
const url = 'https://swapi.dev/api/people/1';

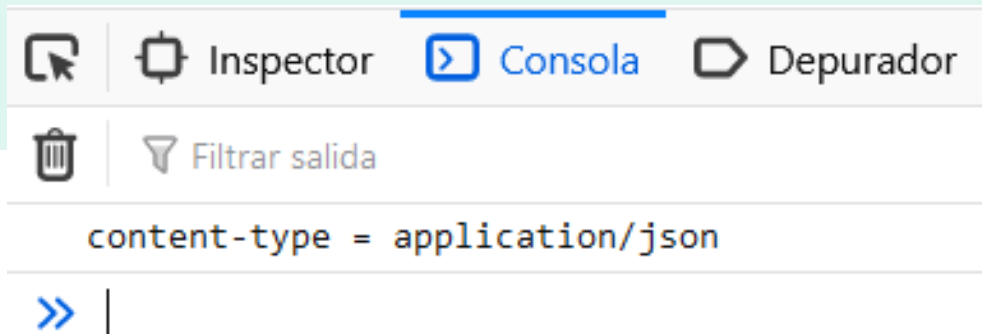
async function getData() {
  fetch( url )
    .then( response => response.json() )
    .then( data => console.log(data) )
}
getData();
```

Se pueden ver los encabezados HTTP de respuesta dentro de **response.headers**

```
const url = 'https://swapi.dev/api/people/1';

async function getData() {
  fetch( url )
    .then( response => {
      for (let [key, value] of response.headers) {
        console.log(`${key} = ${value}`);
      }
    } )
}

getData();
```



Alternativamente también podemos indicar los encabezados que queremos añadir a la petición.

Esto se indica en el segundo parámetro de **fetch**

```
let response = fetch(protectedUrl, {  
  headers: {  
    Authentication: 'secret'  
  }  
});
```

Si queremos realizar una **petición POST**, lo podremos indicar también en las opciones.

```
let user = {
  nombre: 'Juan',
  apellido: 'Perez'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
console.log(result.message);
```

Observa como utilizamos **JSON.stringify()** para convertir el objeto con los datos en una cadena de texto.

JSON.stringify()

El método `JSON.stringify()` convierte un objeto o valor de JavaScript en una cadena de texto JSON, opcionalmente reemplaza valores si se indica una función de reemplazo, o si se especifican las propiedades mediante un array de reemplazo.

Pruébalo

JavaScript Demo: JSON.stringify()

```
1 console.log(JSON.stringify({ x: 5, y: 6 }));  
2 // expected output: '{"x":5,"y":6}'  
3  
4 console.log(JSON.stringify([new Number(3), new String('false'), new Boolean(false)]));  
5 // expected output: '[3,"false",false]'  
6  
7 console.log(JSON.stringify({ x: [10, undefined, function(){}], Symbol('')] }));  
8 // expected output: '{"x":[10,null,null,null]}'  
9
```

Si quisiéramos realizar el paso opuesto, convertir una cadena JSON en objetos, podemos usar la función **JSON.parse()**

JSON.parse()

Resumen

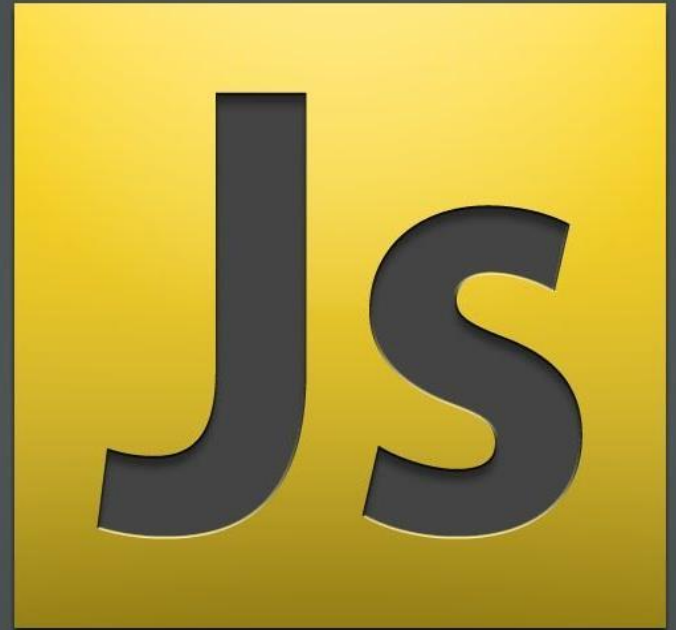
El método `JSON.parse()` analiza una cadena de texto como JSON, transformando opcionalmente el valor producido por el análisis.

Sintaxis

```
JSON.parse(text[, reviver])
```

5

FormData



XXXX