

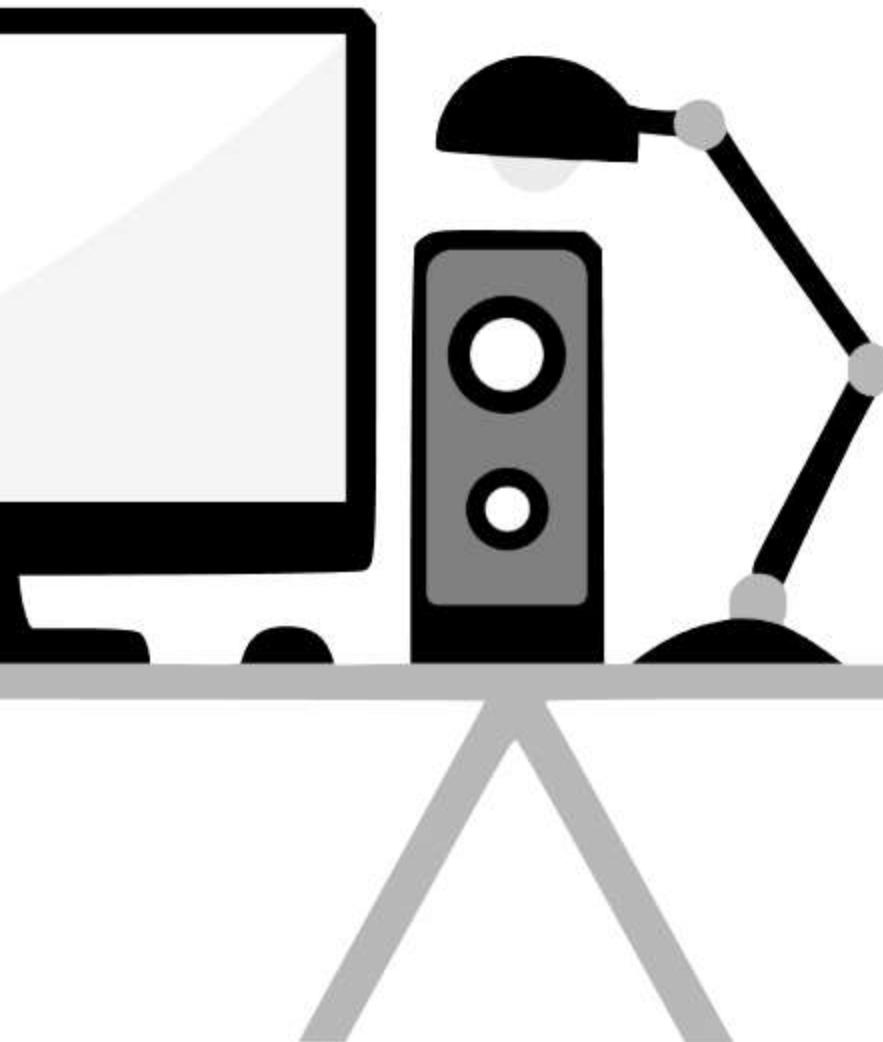


*sustantiva*  
C O N • S E N T I D O

# DESARROLLO DE INTERFACES INTERACTIVAS CON FRAMEWORK VUE

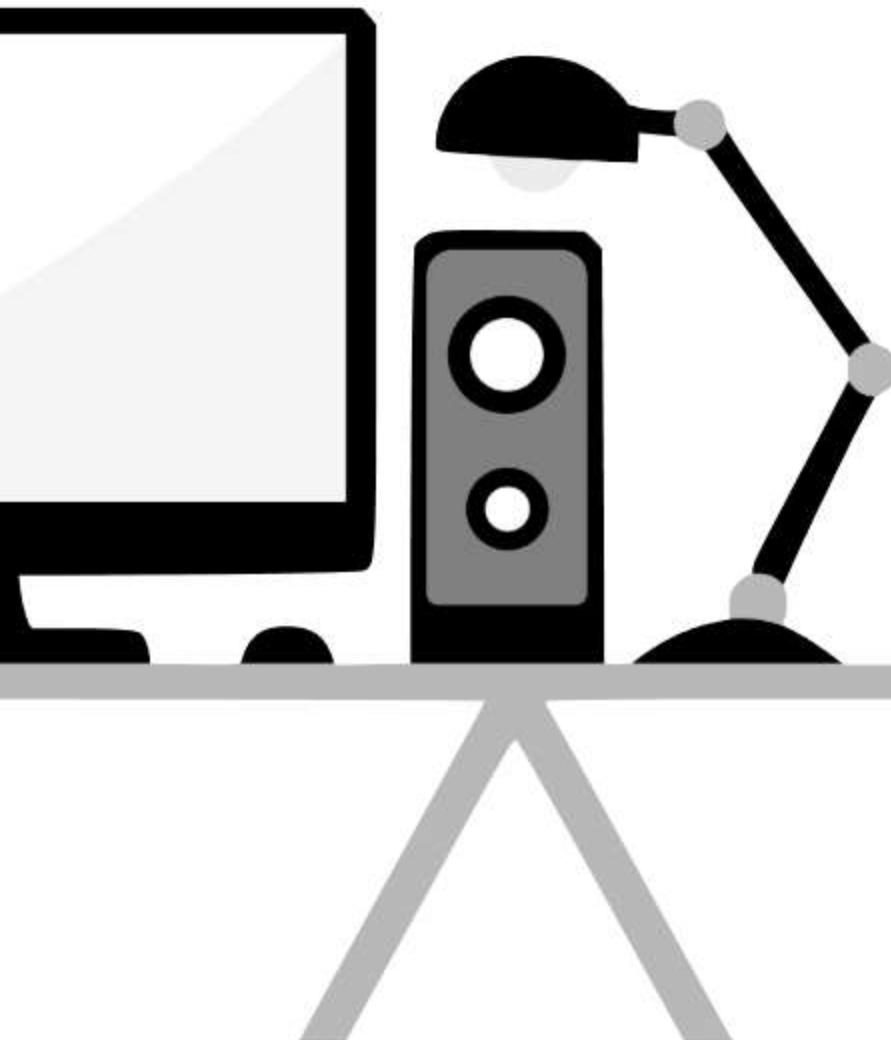
Lección 02

# Contenidos



1. Consumo de datos desde una API Qué es una API
2. Conceptos básicos de comunicación cliente/servidor.
3. Header Body Status Definición de CRUD
4. Web Services, Rest y verbos HTTP básicos
5. GET POST PUT DELETE
6. Interactuando con una API.
7. Conociendo Postman y Reqres <https://reqres.in/>
8. Qué es JSON. Sintaxis e importancia.
9. Autenticación con JWT
10. Encriptación de datos en el cliente
11. Consumo de una API con Axios

# Contenidos



- 12.- Consumiendo un servicio Manejo de promesas y callbacks
- 13.- Manejo de errores
- 14.- El Backend
- 15.-Qué es un backend
- 16.- Tipos de Backend
- 17.- Firebase como Backend
- 18.- Características de Firebase (Autenticación, Base de datos, Hosting, Cloud functions)
- 19.- Integrando Firebase con Vue

# Consumo de Datos desde una Api – Qué es una api

El término API es una abreviatura de 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.

Así pues, podemos hablar de una API como una especificación formal que establece cómo un módulo de un software se comunica o interactúa con otro para cumplir una o muchas funciones. Todo dependiendo de las aplicaciones que las vayan a utilizar, y de los permisos que les dé el propietario de la API a los desarrolladores de terceros.

# Consumo de Datos desde una Api - Qué es una api

En javascript, accedemos a una api a través del comando fetch y en vue existen muchas opciones de como interactuar con apis, dependiendo de si usamos un framework como nuxt, usamos fetch directamente desde la composition api, usamos paquetes para gestionarlas como axios o usamos composables.

La forma más simple y directa es a través del script setup y así, por ejemplo, podríamos tener lo siguiente:

```
function fetch(url) {
  const data = ref(null)
  const error = ref(null)

  function doFetch() {
    // reset state before fetching..
    data.value = null
    error.value = null
    // unref() unwraps potential refs
    fetch(unref(url))
      .then(res => res.json())
      .then(json => (data.value = json))
      .catch(err => (error.value = err))
  }

  if (isRef(url)) {
    // setup reactive re-fetch if input URL is a ref
    watchEffect(doFetch)
  } else {
    // otherwise, just fetch once
    // and avoid the overhead of a watcher
    doFetch()
  }

  return { data, error }
}
```

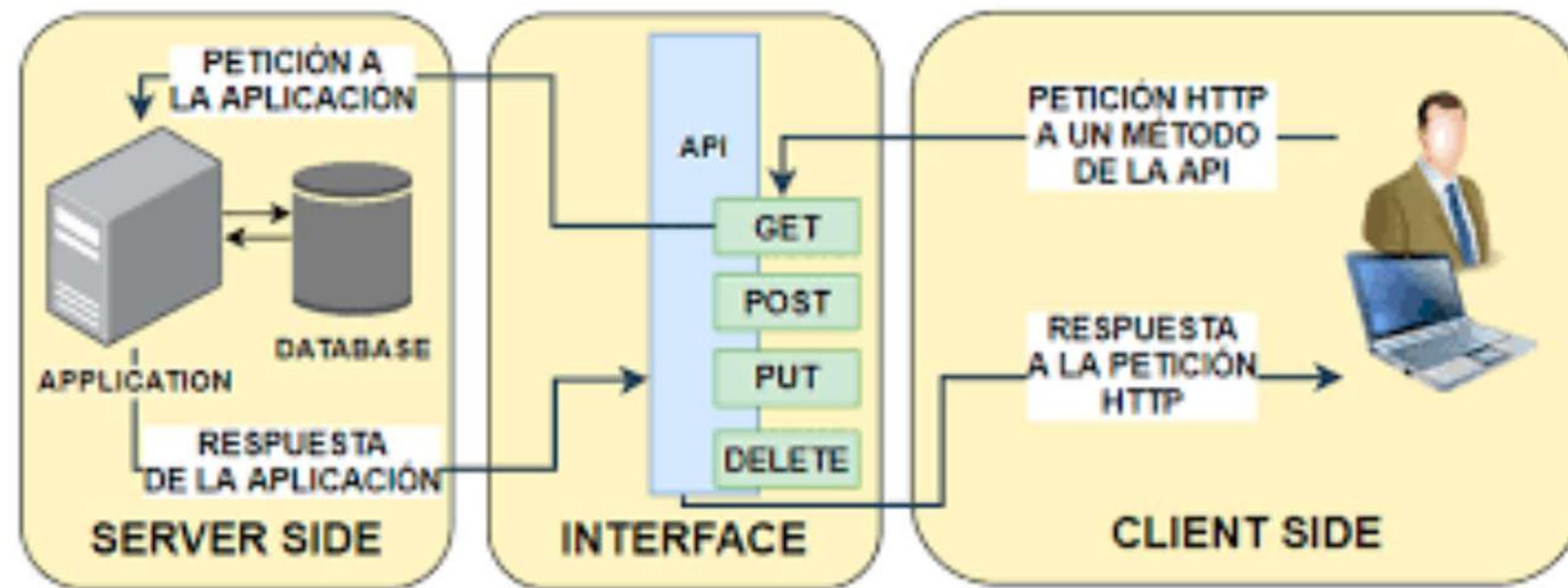
# CONCEPTOS BÁSICOS COMUNICAR CLIENTE/SERVIDOR.

## ¿Y cómo es la comunicación cliente/servidor?

En términos muy simples, el cliente es el navegador, usando un motor como puede ser el v8 de Chrome. Este cliente envía una petición a una api, la cual se encuentra alojada normalmente en un servidor y que expone su contenido a través de recursos expresados en una ruta. Esto se entiende como una petición.

Una petición tiene una estructura para realizarse y lo que nos da el servidor es una respuesta. Dicha respuesta normalmente trae aparejado también un código de respuesta y el contenido de la misma es el que nosotros procesamos en el cliente para exponerla a un usuario.

# CONCEPTOS BÁSICOS COMUNICAR CLIENTE/SERVIDOR.



# CONCEPTOS BÁSICOS COMUNICAR CLIENTE/SERVIDOR.

## ¿Qué es REST?

La transferencia de estado representacional (REST) es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API. En un principio, REST se creó como una guía para administrar la comunicación en una red compleja como Internet. Es posible utilizar una arquitectura basada en REST para admitir comunicaciones confiables y de alto rendimiento a escala. Puede implementarla y modificarla fácilmente, lo que brinda visibilidad y portabilidad entre plataformas a cualquier sistema de API.

Los desarrolladores de API pueden diseñar API por medio de varias arquitecturas diferentes. Las API que siguen el estilo arquitectónico de REST se llaman API REST. Los servicios web que implementan una arquitectura de REST son llamados servicios web RESTful. El término API RESTful suele referirse a las API web RESTful. Sin embargo, los términos API REST y API RESTful se pueden utilizar de forma intercambiable.

Antes vimos como comenzar a armar una petición de fetch a un servidor, pero antes de avanzar con ello es importante especificar que es el HEADEr o encabezado, el BODY o cuerpo.

El header o encabezado define los parámetros operativos o metadatos de una solicitud REST. Por lo general un par de encabezados típicos son:

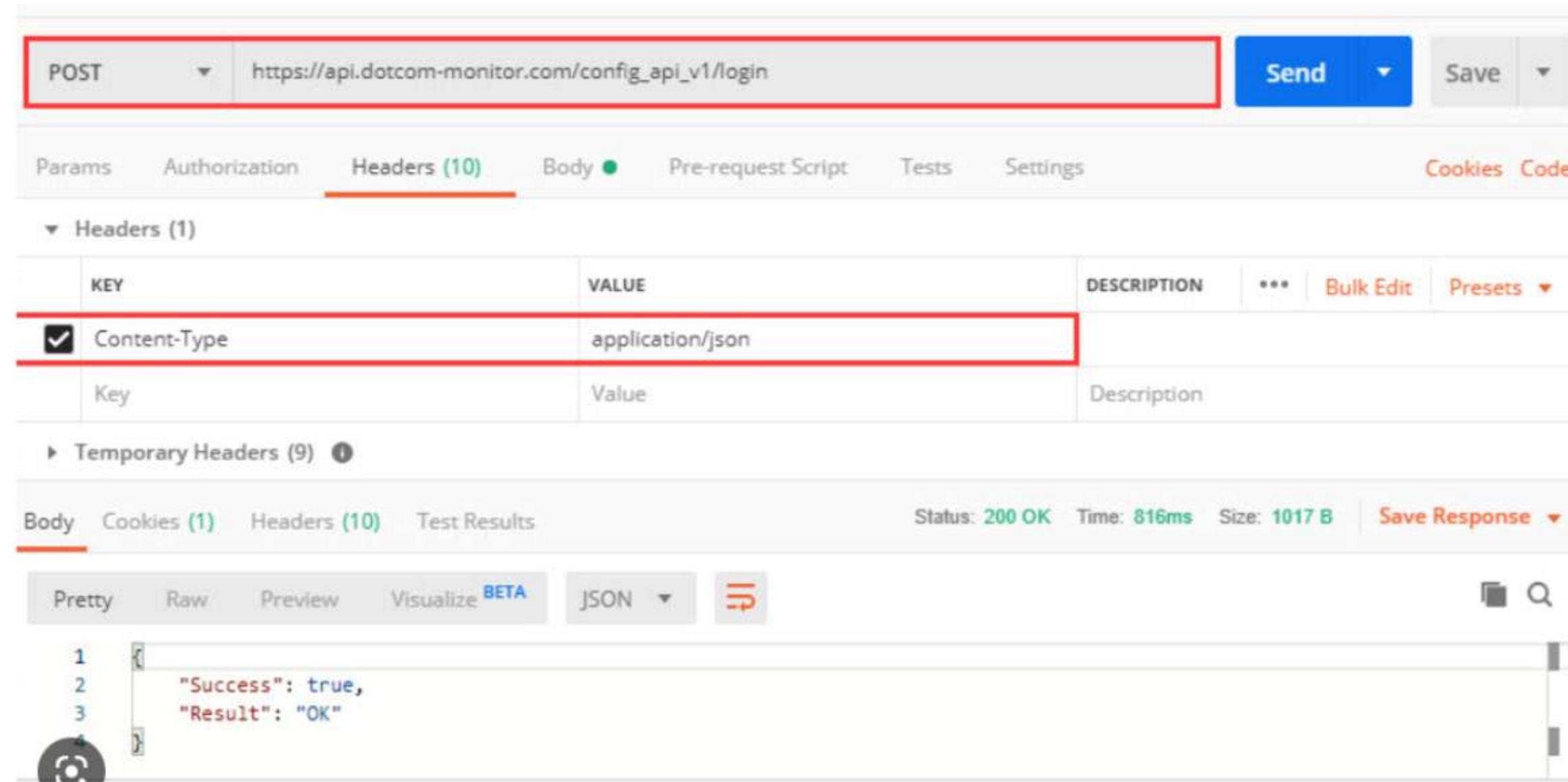
Content-Type	El tipo de medio de los datos en la solicitud. Al incluir un cuerpo en la solicitud de REST, debe especificar el tipo de medio del cuerpo en el campo de encabezado Content-Type. Incluya el campo de encabezado Content-Type en las solicitudes PUT y POST.
Accept	El tipo de medio de los datos de la respuesta. Para especificar el formato de la solicitud, utilice <code>application/&lt;json/xml&gt;</code> en el encabezado o agregue <code>.json</code> o <code>.xml</code> a la URL. El valor predeterminado es XML.

Otro encabezado típico corresponde a la Autentificación, la cual puede llevar datos de acceso usuario/contraseña o bien un token generado por JWT.

Luego del encabezado, tenemos el BODY o cuerpo de la solicitud, donde se envían datos, usualmente en solicitudes POST o PUT. El cuerpo de los datos se escribe después de las líneas de encabezado. Si el mensaje de la solicitud incluye un cuerpo, utilice el campo de encabezado Content-Type para especificar el formato del cuerpo en el encabezado de la solicitud.

El formato más utilizados a nivel de máquina es XML y a nivel de frontend es el JSON

Veremos JSON más a fondo más adelante, pero un ejemplo típico de una solicitud con header y body en POSTMAN es la siguiente:



The screenshot shows the POSTMAN interface with the following details:

- URL Y VERBO:** POST https://api.dotcom-monitor.com/config\_api\_v1/login
- Headers (10):** Content-Type: application/json
- BODY:**

```
1 {  
2   "Success": true,  
3   "Result": "OK"  
4 }
```

URL Y VERBO

HEADER

BODY

El STATUS es un código de respuesta y un mensaje de la api rest, la cual de forma sencilla y mediante convenciones, nos dice que tipo de respuesta nos ha entregado el servidor, ya que nuestra petición puede ser realizada de forma correcta o bien arrojar errores.

Las convenciones típicas (recordar son convenciones y pueden variar) y asociadas a colores de los status son las siguientes:

## HTTP Status Codes



Finalmente, CRUD es un acrónimo para 4 acciones, CREATE, READ, UPDATE y DELETE, las cuales son las 4 acciones típicas que se realizan con datos. Create implica una operación de escritura en una base de datos, read a una operación de lectura o conocido también como traerse datos, update a actualizar la información de una entrada ya existente y finalmente delete que implica eliminar datos desde un backend.

Es importante señalar que aunque el crud se orienta generalmente a una base de datos, esto no es siempre así, puede orientarse a operaciones que solo se manejen en la memoria de la aplicación o incluso en el state en una aplicación cliente.

# WEB SERVICES REST Y VERBOS BÁSICOS

Como vimos anteriormente, el protocolo rest es una forma convencional de comunicarse con una api. Es importante establecer antes de seguir, que suele existir una confusión entre rest, restful y webservice, ya que son conceptos entrelazados y a veces se suele nombrar a unos con el nombre de otros, así que antes de seguir aclarar:

Web service es un programa, diseñado para el intercambio de información máquina a máquina, sobre una red. Entonces esto hace que una computadora o programa pueda solicitar y recibir información de otra computadora o programa. A quien solicita la información se le llama cliente y a quien envía la información se le llama servidor.

# WEB SERVICES REST Y VERBOS BÁSICOS

¿Qué es *rest*? , es una arquitectura para aplicaciones basadas en redes (como Internet), sus siglas significan **R**Epresentational **S**tate **T**ransfer y por otro lado RESTful web service o RESTful api, son programas basados en REST. Pero muchas veces se usan como sinónimos (REST y RESTful).

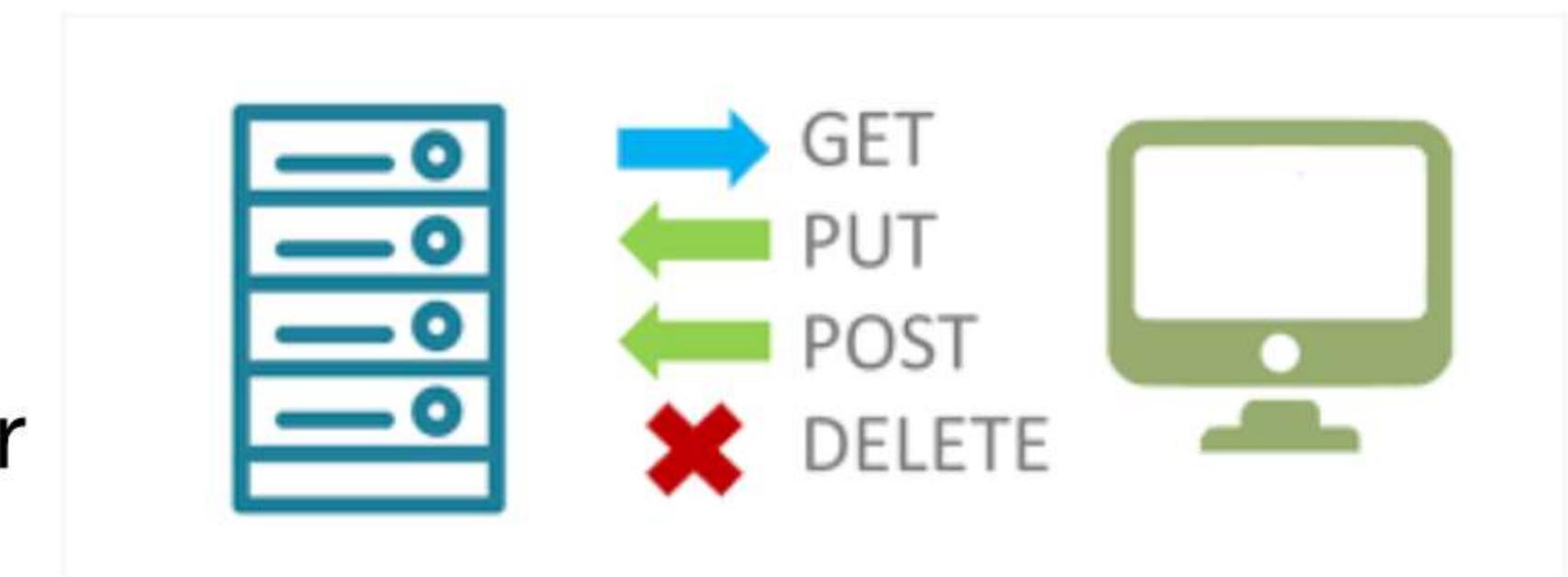
En resumen, un webservice puede o no ser restful y eso depende de si sigue la arquitectura rest. Y la verdad es que suele llamarse rest para abreviar a todo esto, aunque es importante que sepas la diferencia.

Respecto a los verbos, esta es la forma en que nosotros nos comunicamos via rest, existen 4 verbos básicos para las principales operaciones y estos son:

- Listar y leer: Usan el método **GET**
- Crear: Usan el método **POST**
- Actualizar: Usan el método **PATCH** para actualizar y **PUT** para reemplazar.
- Borrar: Usan el método **DELETE**

## Listar y leer: Usan el método **GET**

El método GET solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.



## Crear: Usan el método **POST**

El método POST se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.



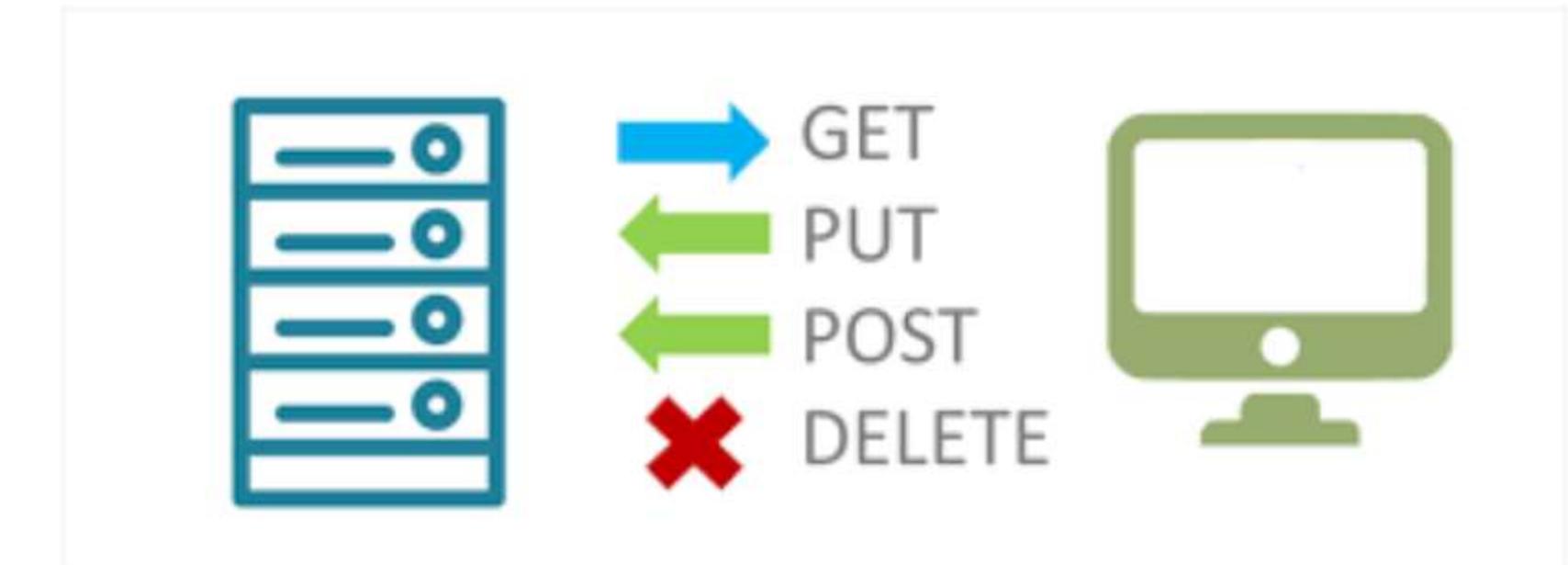
Actualizar: Usan el método **PATCH** para actualizar y **PUT** para reemplazar.

El modo **PUT** reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición.



## Borrar: Usan el método **DELETE**

El método **DELETE** borra un recurso en específico.



Ya con toda la información que tenemos podemos interactuar con una api. Ya podemos hacer una solicitud enviando cualquiera de nuestros cuatro verbos, pero antes ir con fetch a nuestra aplicación vue, podemos usar clientes para interactuar con ellas y además usar servicios online que nos proveen rutas para probar cosas.

En este caso utilizaremos postman y reqres. El primero como cliente y el segundo para tener puntos para probar.

# CONOCIENDO POSTMAN Y REQRES

## SITIO WEB DE POSTMAN

Product ▾ Pricing Enterprise ▾ Resources and Support ▾ Explore

[Sign In](#) [Sign Up for Free](#)

**Build 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.



**API Tools**

A comprehensive set of tools that help accelerate the API Lifecycle - from design, testing, documentation, and mocking to discovery.



**API Repository**

Easily store, iterate and collaborate around all your API artifacts on one central platform used across teams.



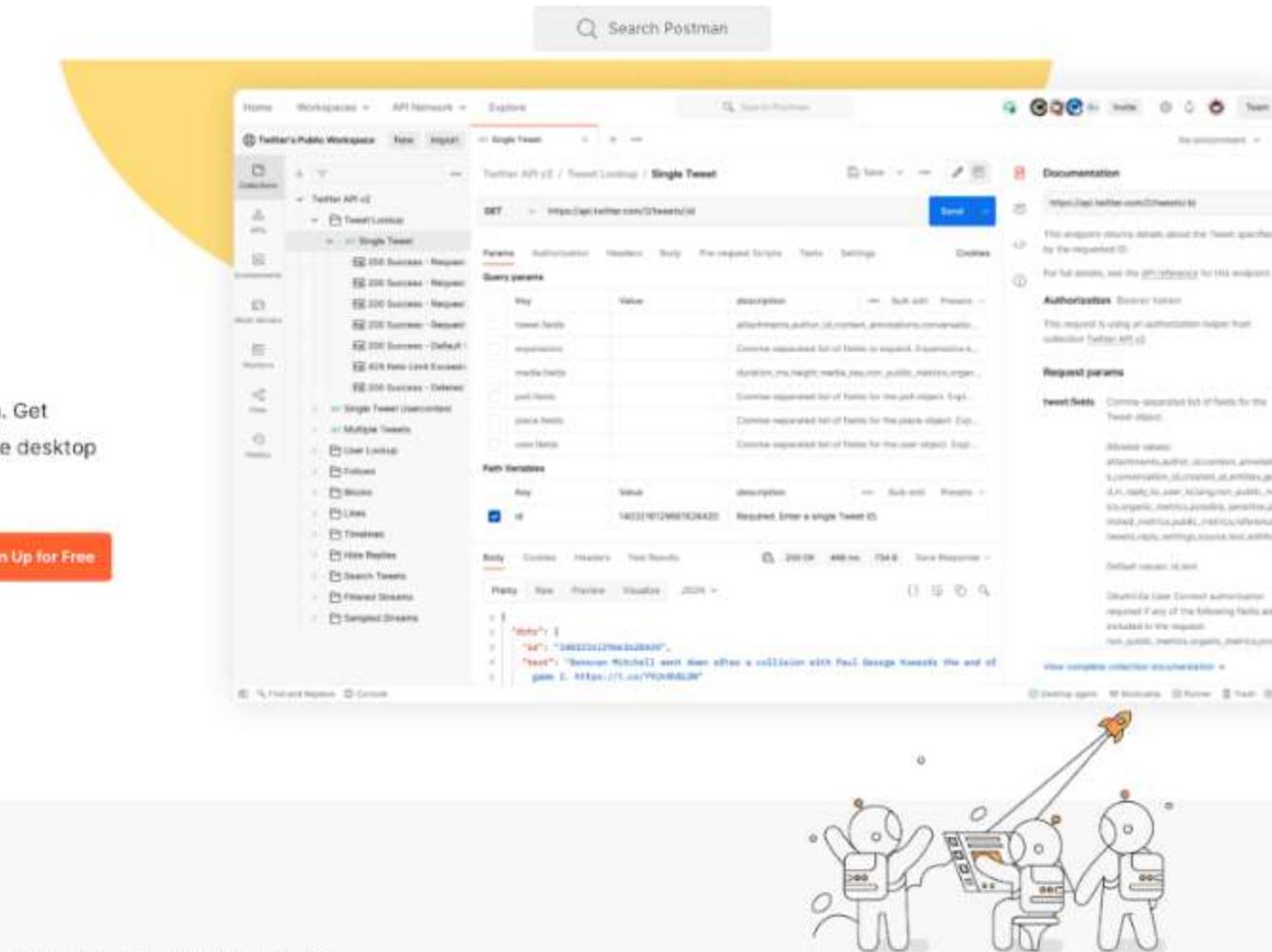
**Workspaces**

Organize your API work and collaborate with teammates across your organization or stakeholders across the world.



**Governance**

Improve the quality of APIs with governance rules that ensure APIs are designed, built, tested, and distributed meeting organizational standards.



The screenshot shows the Postman web application interface. At the top, there's a navigation bar with links for Product, Pricing, Enterprise, Resources and Support, and Explore. On the right, there are 'Sign In' and 'Sign Up for Free' buttons. The main area displays a collection named 'Twitter API v2 / Tweet Lookup / Single Tweet'. It shows a list of responses for various tweet IDs. The 'Documentation' tab is selected, providing details about the endpoint, such as the URL (`https://api.twitter.com/2/tweets/lookup`), description ('This endpoint returns details about the Tweet specified by the required ID.'), and examples. Below the documentation, there are sections for 'Request params' (with a 'tweet.fields' example) and 'Path variables' (with an 'id' example). The bottom part of the screenshot shows a preview of the response body, which includes fields like 'id', 'text', and 'created\_at'.



# CONOCIENDO POSTMAN Y REQRES

Para utilizar postma, hay que bajar su cliente, el cual está en :

<https://www.postman.com/downloads/>

## The Postman app

Download the app to get started with the Postman API Platform.

 Mac Intel Chip

 Mac Apple Chip

By downloading and using Postman, I agree to the [Privacy Policy](#) and [Terms](#).

[Release Notes](#) · [Product Roadmap](#)

Not your OS? Download for Windows ([x64](#)) or Linux ([x64](#))

También puedes usar su cliente web registrándote gratis.

## Postman on the web

Access the Postman API Platform through your web browser. Create a free account, and you're in.

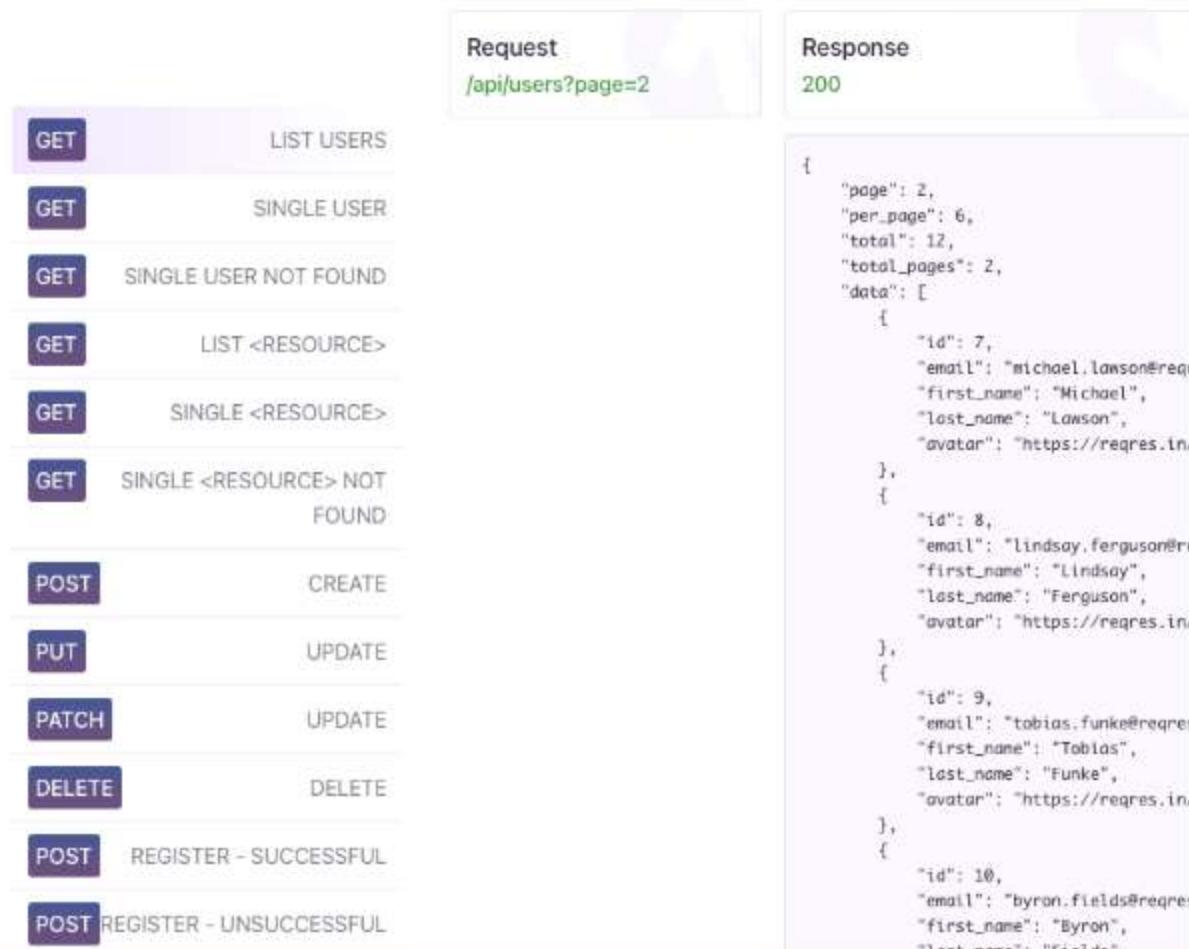
[Try the Web Version](#)

# CONOCIENDO POSTMAN Y REQRES

<https://reqres.in> es una web sencilla al ingresar ya puedes seleccionar algunas rutas y ver el contenido que vas a utilizar.

Give it a try

SUPPORT REQRES



The screenshot shows the Reqres API documentation. On the left, a sidebar lists various API endpoints with their methods and descriptions:

- GET /api/users?page=2 (LIST USERS)
- GET /api/users/:id (SINGLE USER)
- GET /api/users/:id (SINGLE USER NOT FOUND)
- GET /api/users (LIST <RESOURCE>)
- GET /api/users/:id (SINGLE <RESOURCE>)
- GET /api/users/:id (SINGLE <RESOURCE> NOT FOUND)
- POST /api/users (CREATE)
- PUT /api/users/:id (UPDATE)
- PATCH /api/users/:id (UPDATE)
- DELETE /api/users/:id (DELETE)
- POST /api/register (REGISTER - SUCCESSFUL)
- POST /api/register (REGISTER - UNSUCCESSFUL)

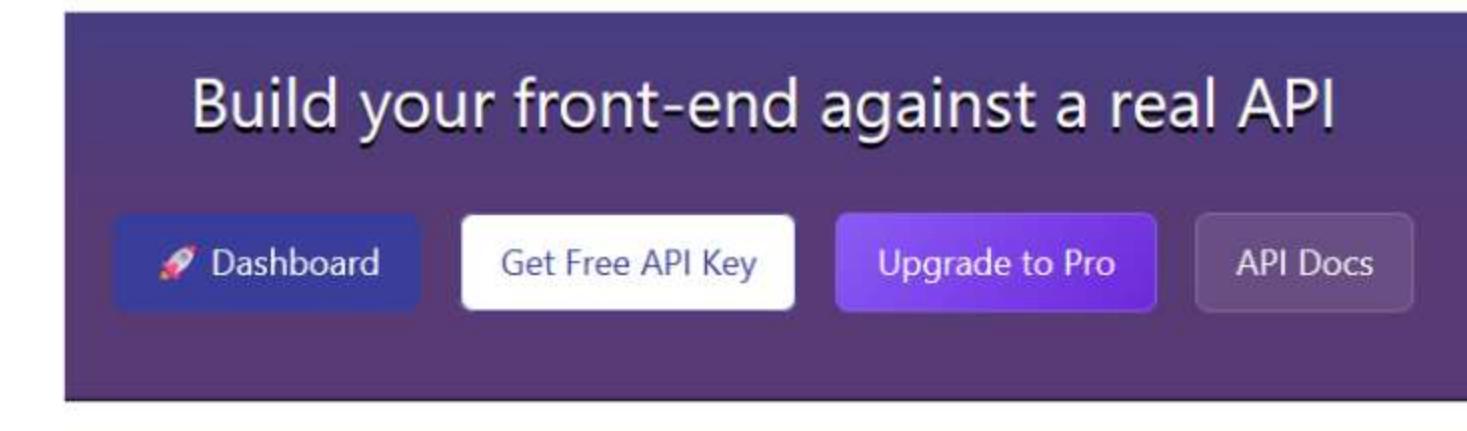
On the right, a specific endpoint is selected: `/api/users?page=2`. The "Request" tab shows a GET method. The "Response" tab shows a 200 status code with a JSON response body:

```
{  
  "page": 2,  
  "per_page": 6,  
  "total": 12,  
  "total_pages": 2,  
  "data": [  
    {  
      "id": 7,  
      "email": "michael.lawson@reqres.in",  
      "first_name": "Michael",  
      "last_name": "Lawson",  
      "avatar": "https://reqres.in/img/faces/7.jpg"  
    },  
    {  
      "id": 8,  
      "email": "lindsay.ferguson@reqres.in",  
      "first_name": "Lindsay",  
      "last_name": "Ferguson",  
      "avatar": "https://reqres.in/img/faces/8.jpg"  
    },  
    {  
      "id": 9,  
      "email": "tobias.funke@reqres.in",  
      "first_name": "Tobias",  
      "last_name": "Funke",  
      "avatar": "https://reqres.in/img/faces/9.jpg"  
    },  
    {  
      "id": 10,  
      "email": "byron.fields@reqres.in",  
      "first_name": "Byron",  
      "last_name": "Fields",  
      "avatar": "https://reqres.in/img/faces/10.jpg"  
    }  
  ]  
}
```

Actualmente, reqres necesita una API key que podemos agregar en la pestaña Headers de Postman:

Key: x-api-key

Value: reqres-free-v1



The screenshot shows the Reqres API landing page. At the top, a banner reads "Build your front-end against a real API". Below the banner are four buttons: "Dashboard" (blue), "Get Free API Key" (white), "Upgrade to Pro" (purple), and "API Docs" (gray). The main content area features a large button labeled "Your Free API Key" with the value "reqres-free-v1". Below this, a section says "Add this header to your API requests:" followed by the header "x-api-key: reqres-free-v1". At the bottom, there's a call-to-action "Hitting 100+ requests/day? Go Pro →".

Your Free API Key

reqres-free-v1

Add this header to your API requests:

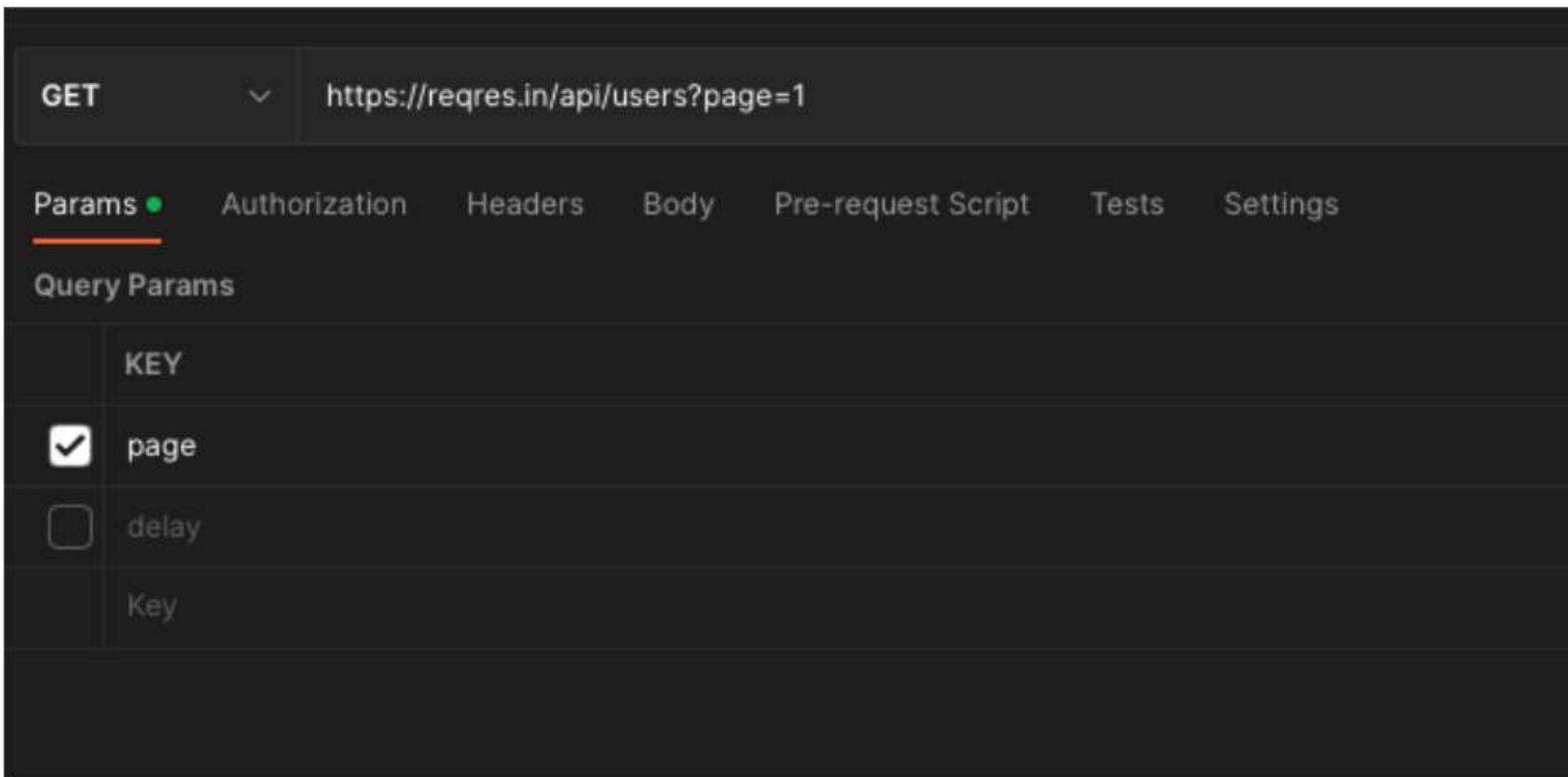
x-api-key: reqres-free-v1

Hitting 100+ requests/day? [Go Pro →](#)

# CONOCIENDO POSTMAN Y REQRES

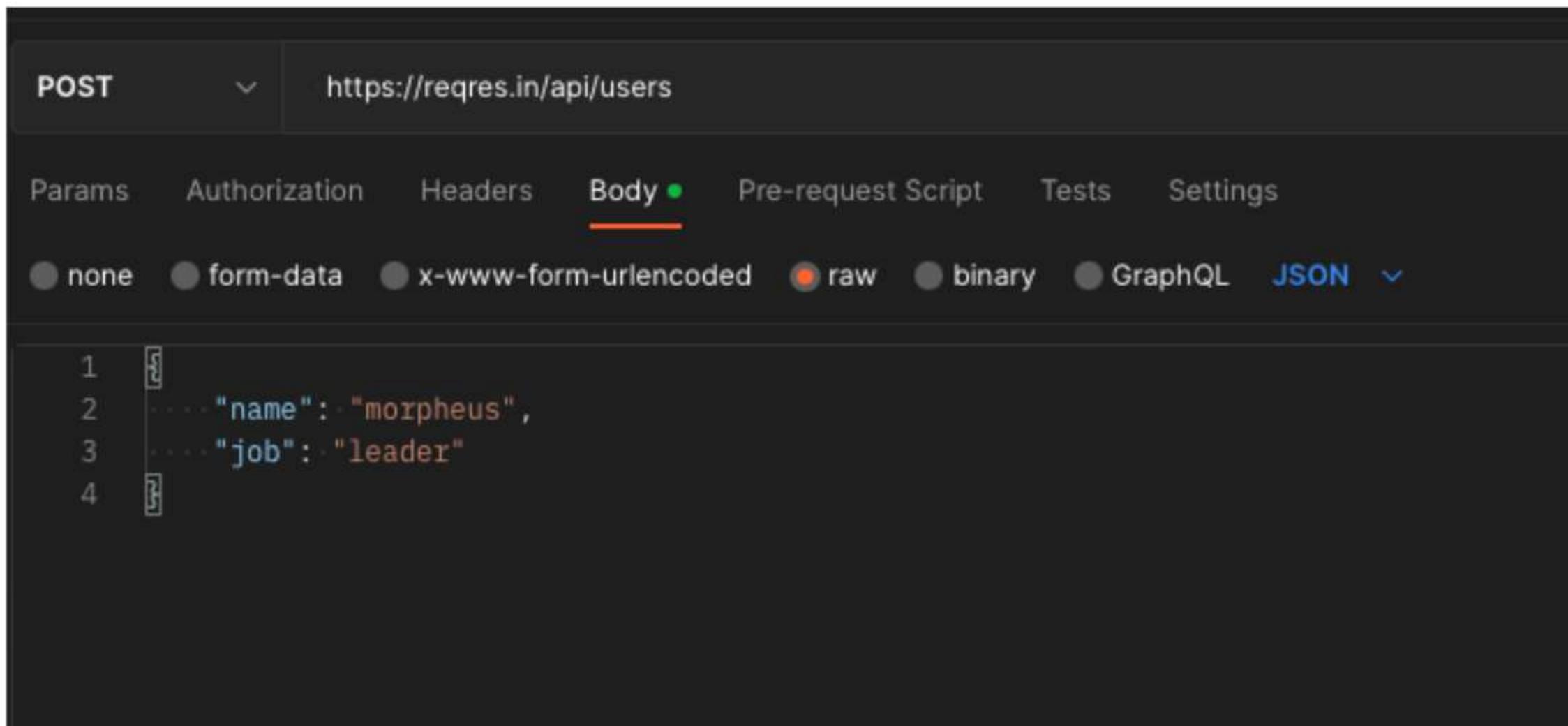
Una vez instalado postman, veremos algunas pruebas con reqresin usando todos los verbos.

Partimos con una petición GET para pedir datos, una lista de usuarios:

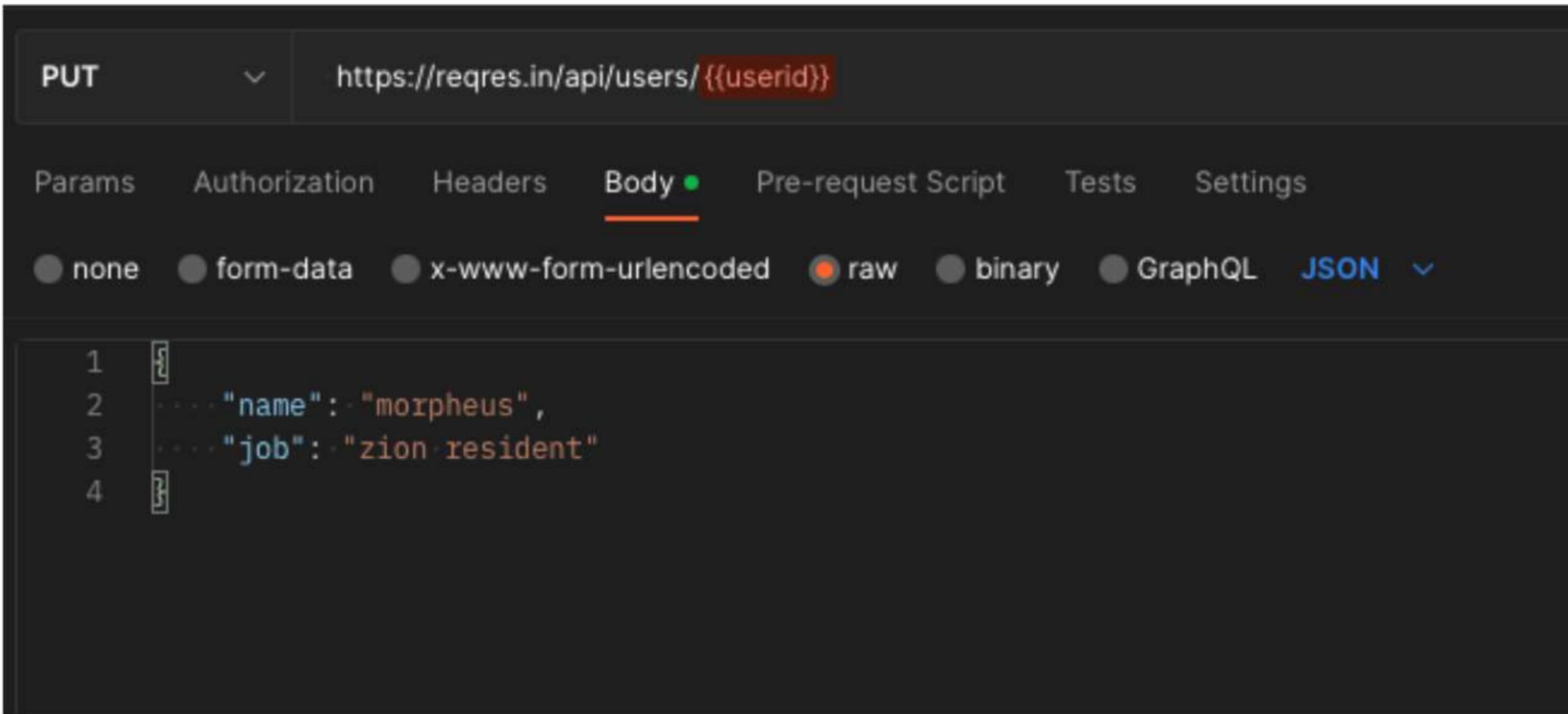


# CONOCIENDO POSTMAN Y REQRES

Un ejemplo de una petición POST, en este caso colocamos la información en body y le indicamos que los datos irán como raw



En este caso usaremos PUT para actualizar, reemplazando {{userId}} con la id del usuario. Esta id la podemos ver en la respuesta de el post que usamos anteriormente o viendo la lista de usuarios.

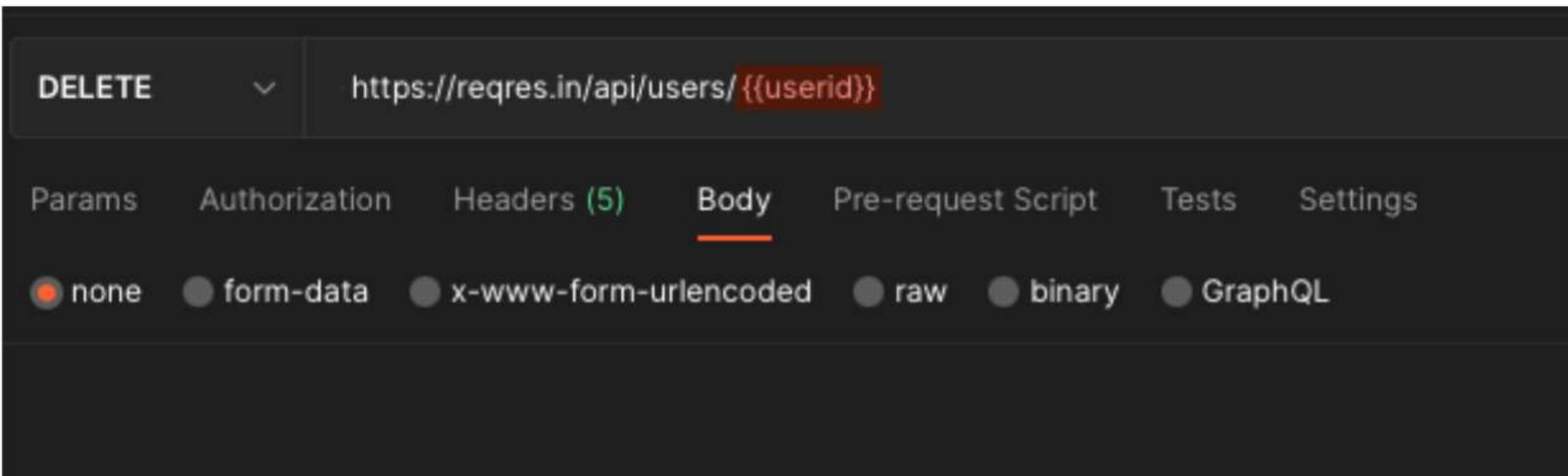


The screenshot shows the Postman application interface. At the top, it displays a 'PUT' method and the URL `https://reqres.in/api/users/{{userid}}`. Below the URL, there are tabs for 'Params', 'Authorization', 'Headers', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Body' tab is currently selected, indicated by an orange underline. Under the 'Body' tab, there are several options: 'none', 'form-data', 'x-www-form-urlencoded', 'raw', 'binary', 'GraphQL', and 'JSON'. The 'JSON' option is selected, as indicated by the blue text and a dropdown arrow. In the main body area, there is a code editor containing the following JSON data:

```
1 [ {  
2   "name": "morpheus",  
3   "job": "zion resident"  
4 } ]
```

Finalmente usamos DELETE para eliminar un usuario, indicando el id del usuario a eliminar en {{userId}}.

Estos valores se conocen como parámetros .



La información en el body se envió mediante JSON. JSON es SON es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato independiente del lenguaje.

Nos quedamos con esa idea, json es una forma más fácil de enviar datos a una api, ya que antes lo que se usaba antes era XML y la diferencia es más menos esta:

## XML

VS.

## JSON

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <endereco>
3   <cep>31270901</cep>
4   <city>Belo Horizonte</city>
5   <neighborhood>Pampulha</neighborhood>
6   <service>correios</service>
7   <state>MG</state>
8   <street>Av. Presidente Antônio Carlos, 6627</street>
9 </endereco>
```

```
1 { 
2   "endereco": {
3     "cep": "31270901",
4     "city": "Belo Horizonte",
5     "neighborhood": "Pampulha",
6     "service": "correios",
7     "state": "MG",
8     "street": "Av. Presidente Antônio Carlos, 6627"
9   }
10 }
```

## ¿Cómo funciona?

Una de las características más significativas de JSON, al ser un formato independiente de los lenguajes de programación, es que los servicios que comparten información por este método no necesitan hablar el mismo idioma. Es decir que el emisor y el receptor pueden ser totalmente distintos, por ejemplo, Java y Python. Esto es así porque cada uno tiene su propia librería de codificación y decodificación para cadenas en este formato.

Es decir que JSON es un formato común para serializar y deserializar objetos en la mayoría de los idiomas. Por eso se ha adoptado ampliamente en el mundo de la programación como una alternativa a XML que suele ser un poco más difícil de utilizar.

Su funcionamiento se basa en la estructuración de una colección de pares con nombre y valor que contienen:

- **Una clave:** que corresponde al identificador del contenido.
- **Un valor:** que representa el contenido correspondiente.

## ¿Cómo surgió JSON?

A comienzos de la década de los 90 surgió el problema de que las máquinas con diferentes sistemas operativos y lenguajes pudieran entenderse entre sí. Así fue, de hecho, que nació XML como una solución estándar.

Sin embargo, XML presentaba problemas, sobre todo cuando se trataba de trabajar con grandes volúmenes de datos, puesto que el procesamiento se volvía lento.

En su momento comenzaron a aparecer intentos para definir formatos que fueran más ligeros y rápidos para el intercambio de información. Uno de ellos fue JSON, promovido y popularizado 10 años después de la aparición de XML.

Desde entonces, JSON se caracteriza por reducir el tamaño de los archivos y el volumen de datos que es necesario transmitir. Estos factores hicieron que JSON fuera adquiriendo popularidad hasta convertirse en una eminencia.

Aun así, esto no significa que XML haya dejado de utilizarse. Actualmente, ambos se usan para el intercambio de datos.

Al usar APIs, ya sea por seguridad o para limitar recursos o bien para tener claro quien revisa que datos y si tiene la autorización para solicitar los datos es que se ha desarrollado una seguridad de los mismos.

Una de las formas más comunes de autenticar y autorizar solicitudes de peticiones a apis es a través de JWT o JSON WEB TOKENS. JWT es SON Web Token es un estándar abierto basado en JSON propuesto por IETF para la creación de tokens de acceso que permiten la propagación de identidad y privilegios o claims en inglés.

En una api normalmente se realiza un primer paso de autenticación con usuario y contraseña a través de una ruta generada para este propósito y luego se genera un token que se añade como encabezado a las peticiones que requieren autenticarse.

Un ejemplo podría ser el de reqres.in, donde si vemos las rutas podemos ver una ruta de registro:

Podemos ver que al registrar con una petición post y esos datos, nos arroja una id y un token

## Request

/api/register

```
{  
  "email": "eve.holt@reqres.in",  
  "password": "pistol"  
}
```

## Response

200

```
{  
  "id": 4,  
  "token": "QpwL5tke4Pnpja7X4"  
}
```

Esto lo podemos usar para login

## Request

/api/login

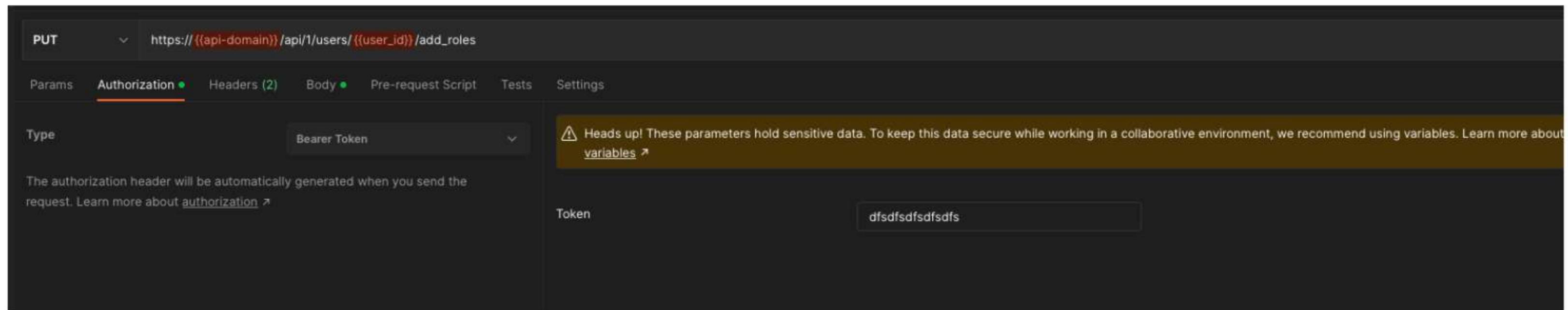
```
{  
  "email": "eve.holt@reqres.in",  
  "password": "cityslicka"  
}
```

## Response

200

```
{  
  "token": "QpwL5tke4Pnpja7X4"  
}
```

Y en el caso de que sea un Bearer Token, lo agregamos a postman de esta forma para revisar esas rutas.



The screenshot shows a Postman request configuration for a PUT method. The URL is `https://{{api-domain}}/api/1/users/{{user_id}}/add_roles`. The 'Authorization' tab is selected, showing a dropdown set to 'Bearer Token'. A warning message states: '⚠ Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables. Learn more about variables ↗'. Below the dropdown, it says: 'The authorization header will be automatically generated when you send the request. Learn more about authorization ↗'. The 'Token' field contains the value 'dfsdfsdfsdfsdf'.

En la respuesta vemos los datos de registro

#### Response body

```
{  
  "data": {  
    "id": 1,  
    "email": "george.bluth@reqres.in",  
    "first_name": "George",  
    "last_name": "Bluth",  
    "avatar": "https://reqres.in/img/faces/1-image.jpg"  
  },  
  "support": {  
    "url": "https://reqres.in/#support-heading",  
    "text": "To keep ReqRes free, contributions towards server costs are appreciated!"  
  }  
}
```



Download

#### Response headers

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Es importante destacar que para agregar una capa más de seguridad los datos del usuario también pueden ser encriptados y desencriptados desde el lado del cliente.

Razones para hacer esto son varias, una de las más importantes es no exponer información sensible de los clientes a los servidores, los cuales recordemos, también pueden ser vulnerables.

Si las contraseñas se guardan en el servidor de forma encriptada, una vulneración de estos no redundaría en revelar una clave de usuario, que podría usar en otros servicios.

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Javascript posee una interface para estas labores llamada crypto.

Un ejemplo simple de encriptación de claves en javascript usando crypto ( documentación en: <https://developer.mozilla.org/en-US/docs/Web/API/Crypto> )

Es la siguiente, creando primero listeners y variables:

```
document.addEventListener("DOMContentLoaded", async () => {
    const $contraseñaEncriptar = document.querySelector("#contraseñaEncriptar"),
        $informacionEncriptar = document.querySelector("#informacionEncriptar"),
        $resultadoEncriptacion = document.querySelector("#resultadoEncriptacion"),
        $botonEncriptar = document.querySelector("#botonEncriptar"),
        $contraseñaDesencriptar = document.querySelector("#contraseñaDesencriptar"),
        $informacionDesencriptar = document.querySelector("#informacionDesencriptar"),
        $resultadoDesencriptacion = document.querySelector("#resultadoDesencriptacion"),
        $botonDesencriptar = document.querySelector("#botonDesencriptar");

    const bufferABase64 = buffer => btoa(String.fromCharCode(...new Uint8Array(buffer)));
    const base64ABuffer = buffer => Uint8Array.from(atob(buffer), c => c.charCodeAt(0));
    const LONGITUD_SAL = 16;
    const LONGITUD_VECTOR_INICIALIZACION = LONGITUD_SAL;
```

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Luego una función para las claves:

```
const derivacionDeClaveBasadaEnContraseña = async (contraseña, sal, iteraciones, longitud, hash, algoritmo = 'AES-CBC') => {
    const encoder = new TextEncoder();
    let keyMaterial = await window.crypto.subtle.importKey(
        'raw',
        encoder.encode(contraseña),
        { name: 'PBKDF2' },
        false,
        ['deriveKey']
    );
    return await window.crypto.subtle.deriveKey(
        {
            name: 'PBKDF2',
            salt: encoder.encode(sal),
            iterations: iteraciones,
            hash
        },
        keyMaterial,
        { name: algoritmo, length: longitud },
        false,
        ['encrypt', 'decrypt']
    );
}
```

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Una función para encriptar:

```
const encriptar = async (contraseña, textoPlano) => {
    const encoder = new TextEncoder();
    const sal = window.crypto.getRandomValues(new Uint8Array(LONGITUD_SAL));
    const vectorInicializacion = window.crypto.getRandomValues(new Uint8Array(LONGITUD_VECTOR_INICIALIZACION));
    const bufferTextoPlano = encoder.encode(textoPlano);
    const clave = await derivacionDeClaveBasadaEnContraseña(contraseña, sal, 100000, 256, 'SHA-256');
    const encrypted = await window.crypto.subtle.encrypt(
        { name: "AES-CBC", iv: vectorInicializacion },
        clave,
        bufferTextoPlano
    );
    return bufferABase64([
        ...sal,
        ...vectorInicializacion,
        ...new Uint8Array(encrypted)
    ]);
};
```

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Otra para desencriptar:

```
const desencriptar = async (contraseña, encriptadoEnBase64) => {
    const decoder = new TextDecoder();
    const datosEncriptados = base64ABuffer(encriptadoEnBase64);
    const sal = datosEncriptados.slice(0, LONGITUD_SAL);
    const vectorInicializacion = datosEncriptados.slice(0 + LONGITUD_SAL, LONGITUD_SAL + LONGITUD_VECTOR_INICIALIZACION);
    const clave = await derivacionDeClaveBasadaEnContraseña(contraseña, sal, 100000, 256, 'SHA-256');
    const datosDesencriptadosComoBuffer = await window.crypto.subtle.decrypt(
        { name: "AES-CBC", iv: vectorInicializacion },
        clave,
        datosEncriptados.slice(LONGITUD_SAL + LONGITUD_VECTOR_INICIALIZACION)
    );
    return decoder.decode(datosDesencriptadosComoBuffer);
}
```

# ENCRYPTACIÓN DE DATOS CON EL CLIENTE

Una opción más sencilla es ocupar un composable o un paquete de vue. Dejo acá link con un paquete que hace el trabajo más fácil:

Paquete de npm

<https://www.npmjs.com/package/vue-cryptojs>

Si bien todo puede ser encriptado, es mejor dejarlo solo para información sensible.

# CONSUMIENDO UNA API CON AXIOS

Y hablando de paquetes que hacen la vida más simple, existe un paquete para hacer fetch que ayuda mucho en el día a día y este es axios.

<https://www.npmjs.com/package/axios>

Axios permite realizar el trabajo con peticiones de formas más sencillas, resolviendo a priori muchos problemas que ocurren con fetch, de forma liviana.

Una de las ventajas principales de axios es que automáticamente transforma la data recibida (parse json) y que además es más sencilla de ocupar.

# CONSUMIENDO UNA API CON AXIOS

¿Qué tan sencillo es? Dejo una petición de ejemplo para comparar:

```
axios.post('/user', {
  firstName: 'Fred',
  lastName: 'Flintstone'
})
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
});
```

```
// Example POST method implementation:
async function postData(url = '', data = {}) {
  // Default options are marked with *
  const response = await fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
      'Content-Type': 'application/json'
      // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, origin,
origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin,
unsafe-url
    body: JSON.stringify(data) // body data type must match "Content-Type" header
  });
  return response.json(); // parses JSON response into native JavaScript objects
}

postData('https://example.com/answer', { answer: 42 })
.then((data) => {
  console.log(data); // JSON data parsed by `data.json()` call
});
```

# CONSUMIENDO UNA API CON AXIOS

¿Qué tan sencillo es? Dejo una petición de ejemplo para comparar:

```
axios.post('/user', {  
  firstName: 'Fred',  
  lastName: 'Flintstone'  
})  
.then(function (response) {  
  console.log(response);  
})  
.catch(function (error) {  
  console.log(error);  
});
```

No lleva la url completa porque se configura en baseurl

El body se coloca como un objeto simple

No necesita parsearse la petición y si hay cabeceras, estas se configuran fácilmente.

# CONSUMIENDO UN SERVICIO

Consumiendo un servicio, manejo de promesas y callbacks.

Un servicio por definición es asíncrono y por lo mismo, el consumirlo se debe hacer observando esa asincronicidad. Al principio esto se hacía mediante callbacks, donde una función llamaba a otra y se iba manejando la asincronicidad, pero era verboso y difícil de debuggear.

Es por eso que surgen las promesas y después de estas `async/await`, azúcar de tipo sintáctico para hacer más fácil trabajar con estas.

En vue puedes trabajar con script setup, `async/await` de la siguiente forma:

# CONSUMIENDO UN SERVICIO

Consumiendo un servicio, manejo de promesas y callbacks.

```
vue
<script setup>
const res = await fetch('...')

const posts = await res.json()

</script>

<template>
  {{ posts }}
</template>
```

# CONSUMIENDO UN SERVICIO

Y aunque escapa del scope del curso, es importante hacer notar que se puede usar Suspense con estos componentes en el template, un ejemplo de esto sería el siguiente:

```
template
<Suspense>
  <!-- component with nested async dependencies -->
  <Dashboard />

  <!-- loading state via #fallback slot -->
  <template #fallback>
    Loading...
  </template>
</Suspense>
```

# MANEJO DE ERRORES

Y el manejo de errores, si bien no se realiza mediante Suspense en si, se puede realizar usando errorCaptured o el hook onErrorCaptured()

```
<template>
  <div v-if="error">
    {{ error }}
  </div>
  <Suspense v-else>
    <template #default>
      <UserProfile />
    </template>
    <template #fallback>
      <div>Loading...</div>
    </template>
  </Suspense>
</template>
```

```
<script>
import { onErrorCaptured } from 'vue'

setup () {
  const error = ref(null)
  onErrorCaptured(e => {
    error.value = e
    return true
  })
  return { error }
</script>
```

Finalmente, así como tenemos la capa de servidor, también tenemos el backend, que nos presta lógica de negocios y datos para nuestro Frontend.

Es vital para un programador comprender las bases y como se maneja un backend.

El backend es la parte del desarrollo web que se encarga de que toda la lógica de una página web funcione. Se trata del conjunto de acciones que pasan en una web pero que no vemos como, por ejemplo, la comunicación con el servidor.

## Tipos de Backend.

Es un poco extraño preguntar por tipos de backend. En general uno suele clasificarlos e identificarlos según los lenguajes de programación que manejan (backends en javascript, go, php, java, c++, etc) o bien en la infraestructura en que están alojados (cloud, on premise, function, Edge, etc).

Lo cierto, es que lo importante no es tanto en que lenguaje está programado o en que infraestructura está montado el backend, ya que nosotros nos comunicaremos con ellos usualmente mediante rutas.

Ahora, hay excepciones a esto, puesto que hay backends, que ellos programáticamente generan el html que consumiremos, esto es lo que se conoce como server side rendering, existiendo frameworks en vue que lo hacen, como es vite-ssr o nuxt.

Es importante tener claro como frontenders como nos comunicaremos con el backend y en este caso, mostraremos uno en particular que se puede adaptar a vue y que además es cloud, hablamos de firebase.

# FIREBASE COMO BACKEND

Firebase adopta la forma de una plataforma en la nube para desarrollar, administrar y ejecutar aplicaciones. Ofrece una amplia gama de herramientas para realizar el desarrollo de aplicaciones y las asignaciones de alojamiento.

Aunque cuenta con algunos problemas, ejecuta y administra aplicaciones de manera eficiente y también capta a los usuarios de la aplicación. En lugar de desarrollar estas herramientas por sí mismos, los desarrolladores pueden acceder a todos estos poderosos programas listos para usar con Firebase para que puedan concentrarse en la tarea principal de desarrollar aplicaciones funcionales para sus usuarios.

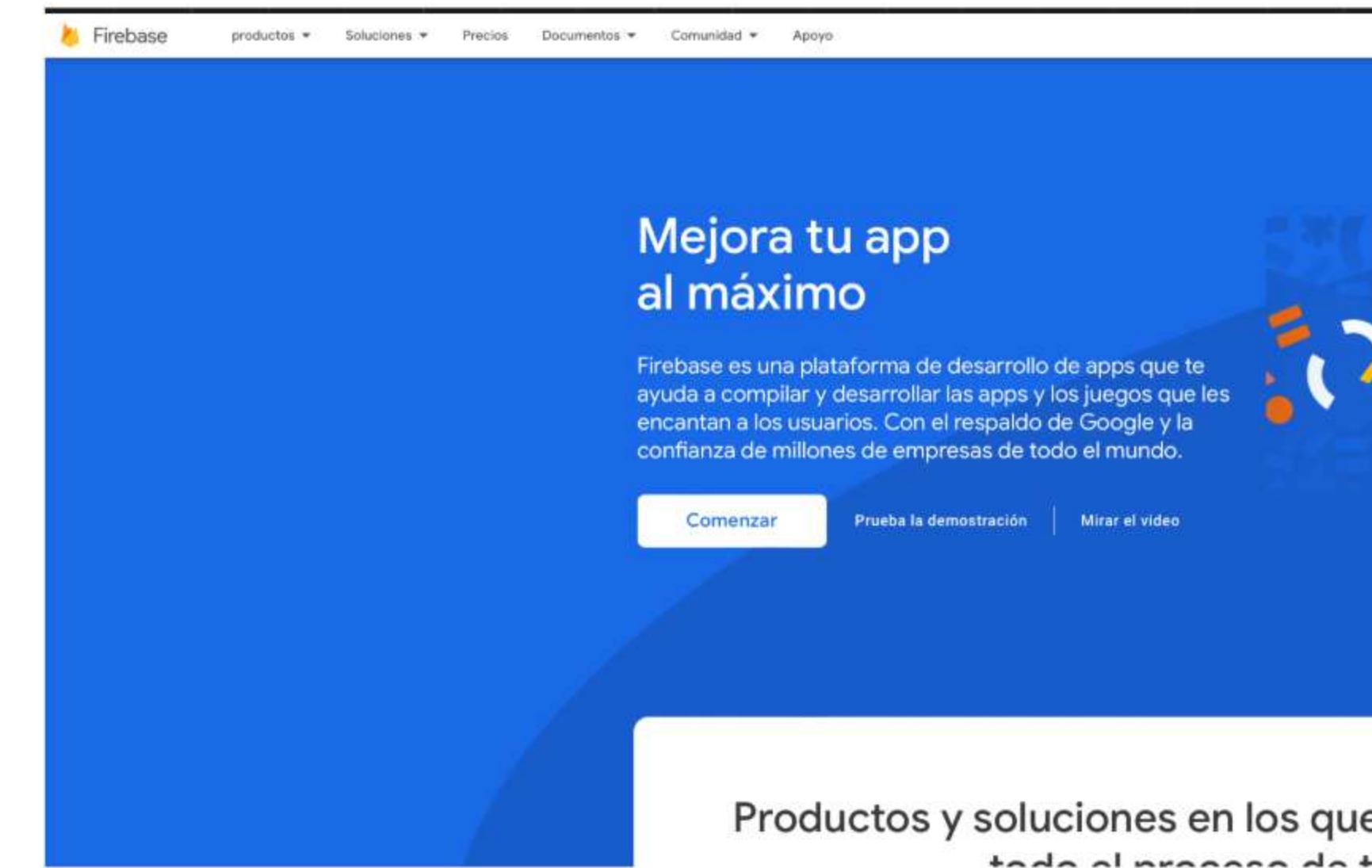
Algunas características excelentes de Firebase incluyen mensajería push, análisis de servidor, almacenamiento de backend, autenticación de usuario y mucho más.

Firebase está alojado en la nube, lo que permite a los desarrolladores escalar sus aplicaciones sin estrés. Todas estas características hacen de Firebase una de las soluciones de desarrollo de aplicaciones líderes en el mundo en la actualidad.

# FIREBASE COMO BACKEND

Es fácil partir con firebase, se necesita solo una cuenta de Google y seguir 3 pasos:

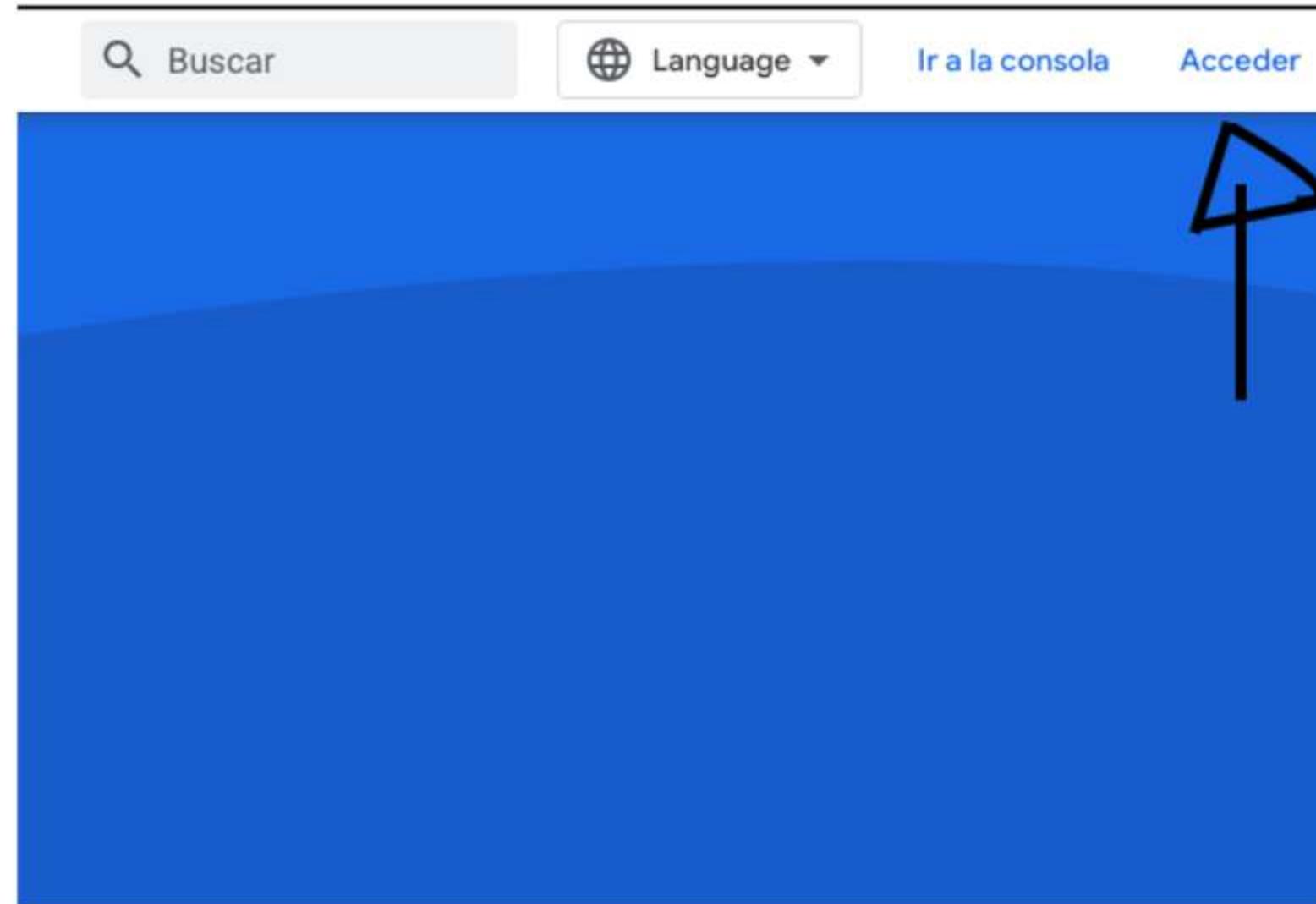
1. Ingresar en este enlace: <https://firebase.google.com/>



# FIREBASE COMO BACKEND

Es fácil partir con firebase, se necesita solo una cuenta de Google y seguir 3 pasos:

2. Debes loguearte a través de una cuenta Gmail / Gsuite.



# FIREBASE COMO BACKEND

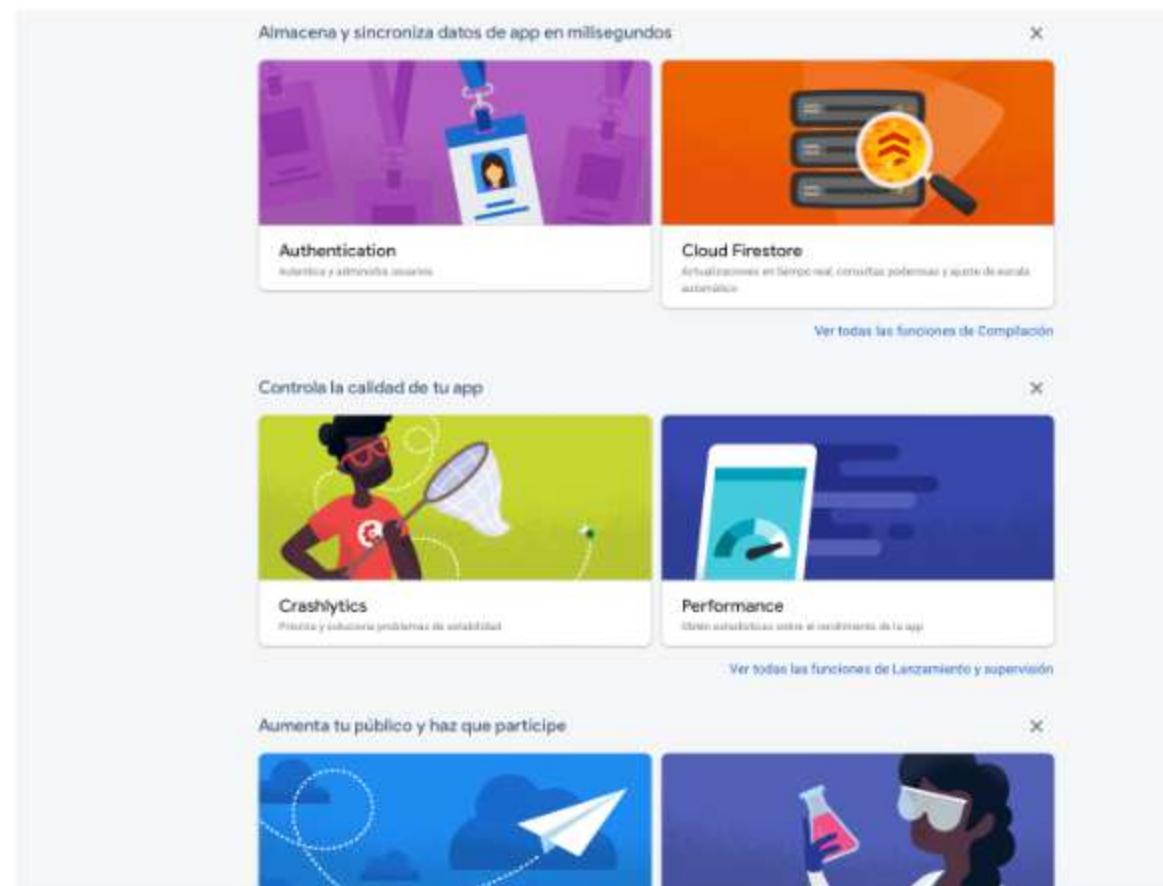
Es fácil partir con firebase, se necesita solo una cuenta de Google y seguir 3 pasos:

3. crear un proyecto



# FIREBASE COMO BACKEND

Cada proyecto puede tener uno o más servicios de firebase asociados y esos servicios, los veremos en las siguientes diapositivas:



Desde este panel puedes ver  
Los servicios de firebase  
E incorporarlos a tu proyecto

# CARACTERÍSTICAS DE FIREBASE

## REALTIME DATABASE

Una de las herramientas más destacadas y esenciales de Firebase son las bases de datos en tiempo real. Estas se alojan en la nube, son No SQL y almacenan los datos como JSON. Permiten alojar y disponer de los datos e información de la aplicación en tiempo real, manteniéndolos actualizados aunque el usuario no realice ninguna acción. Firebase envía automáticamente eventos a las aplicaciones cuando los datos cambian, almacenando los datos nuevos en el disco. Aunque no hubiera conexión por parte de un usuario, sus datos estarían disponibles para el resto y los cambios realizados se sincronizarían una vez restablecida la conexión.

## AUTENTICACIÓN DE USUARIOS

La identificación de los usuarios de una app es necesaria en la mayoría de los casos si estos quieren acceder a todas sus características.

Firebase ofrece un sistema de autenticación que permite tanto el registro propiamente dicho (mediante email y contraseña) como el acceso utilizando perfiles de otras plataformas externas (por ejemplo, de Facebook, Google o Twitter), una alternativa muy cómoda para usuarios reacios a completar el proceso.

Así, este tipo de tareas se ven simplificadas, considerando también que desde aquí se gestionan los accesos y se consigue una mayor seguridad y protección de los datos.

Se debe mencionar que Firebase puede guardar en la nube los datos de inicio de sesión con total seguridad, evitando que una persona tenga que identificarse cada vez que abra la aplicación.

# CARACTERÍSTICAS DE FIREBASE

## ALMACENAMIENTO EN LA NUBE

Firebase cuenta con un sistema de almacenamiento, donde los desarrolladores pueden guardar los **ficheros de sus aplicaciones** (y vinculándolos con referencias a un árbol de ficheros para mejorar el rendimiento de la app) y sincronizarlos. Al igual que la mayoría de herramientas de Firebase, es personalizable mediante determinadas reglas.

Este almacenamiento es de gran ayuda para **tratar archivos de los usuarios** (por ejemplo, fotografías que hayan subido), que se pueden servir de forma más rápida y fácil. También hace la descarga de referencias a ficheros más segura.

# CARACTERISTICAS DE FIREBASE

## HOSTING

Firebase también ofrece un **servidor para alojar las apps de manera rápida y sencilla**, esto es, un hosting estático y seguro. Proporciona certificados de seguridad **SSL** y **HTTP2** de forma automática y gratuita para cada dominio, reafirmando la seguridad en la navegación. Funciona situándolas en el **CDN** (Content Delivery Network) de Firebase, una red que recibe los archivos subidos y permite entregar el contenido.

## CLOUD MESSAGING

Su utilidad es el **envío de notificaciones y mensajes** a diversos usuarios en tiempo real y a través de varias plataformas.

# INTEGRANDO FIREBASE A VUE

La forma más sencilla de integrar firebase y su sdk a vue es usando el plugin vuefire.

Para esto primero instalamos vuefire y firebase, dos paquetes:

```
yarn add vuefire firebase  
# or  
npm install vuefire firebase
```

# INTEGRANDO FIREBASE A VUE

Instalamos Vuefire  
En nuestra instancia  
De Vue:

```
ts

import { createApp } from 'vue'
import { VueFire, VueFireAuth } from 'vuefire'
import App from './App.vue'
// the file we created above with `database`, `firestore` and other exports
import { firebaseApp } from './firebase'

const app = createApp(App)
app
  .use(VueFire, {
    // imported above but could also just be created here
    firebaseApp,
    modules: [
      // we will see other modules later on
      VueFireAuth(),
    ],
  })
  .mount('#app')
```

# INTEGRANDO FIREBASE A VUE

Finalmente usamos composable  
Para usar los servicios  
En aquellas partes  
Donde nos interese llamarlos:

```
<script setup>
import { useFirestore } from 'vuefire'

const db = useFirestore()

</script>

<template>
  <div>
    ...
  </div>
</template>
```

# INTEGRANDO FIREBASE A VUE

Un ejemplo sencillo  
Usando la firestore  
Para traer una colección de datos  
Y pintarla usando un v-for:

```
<script setup>
import { useCollection } from 'vuefire'
import { collection } from 'firebase/firestore'

const todos = useCollection(collection(db, 'todos'))
</script>

<template>
  <ul>
    <li v-for="todo in todos" :key="todo.id">
      <span>{{ todo.text }}</span>
    </li>
  </ul>
</template>
```



*sustantiva*  
C O N • S E N T I D O