



*JavaScript: Ajax*

**Definición: Ajax**

A principios del 2000, los/as desarrolladores/as web se encontraban ante el constante desafío de obtener o enviar información sin afectar el estado actual de la página, esto es, sin requerir una recarga completa (refresco de la página).

Si bien existían recursos como la etiqueta [<iframe>](https://developer.mozilla.org/es/docs/Web/HTML/Element/iframe) para cargar contenido anidado en el sitio, es decir, otro HTML dentro de la página actual, la necesidad de comunicar al cliente con el servidor, sin interrumpir el funcionamiento de la aplicación, derivó en la búsqueda de nuevas técnicas de programación.

Para dar respuesta a este problema, surgió [AJAX](https://developer.mozilla.org/es/docs/Web/Guide/AJAX/Getting_Started) (JavaScript Asíncrono y [XML](https://developer.mozilla.org/es/docs/Web/XML/XML_introduction)), un conjunto de técnicas de desarrollo que permiten que las aplicaciones web funcionen de forma asincrónica, pudiendo procesar tareas en segundo plano. Entendemos a un proceso asíncrono como aquel que no genera tiempo de espera, es decir, que no existe pausa en el funcionamiento del programa mientras se aguarda la respuesta del servidor.

En consecuencia, cualquier app o web que emplee AJAX puede enviar y recibir datos sin volver a cargar toda la página, evitando la interrupción de acciones realizadas por el usuario, añadiendo interactividad y dinamismo a nuestra aplicación, **características esenciales del software moderno**.

Para indagar en el funcionamiento de esta tecnología, es prioritario comprender cómo interactúan el cliente y el servidor, centrándonos principalmente en cómo estas dos partes de la aplicación se comunican empleando ciertas reglas.

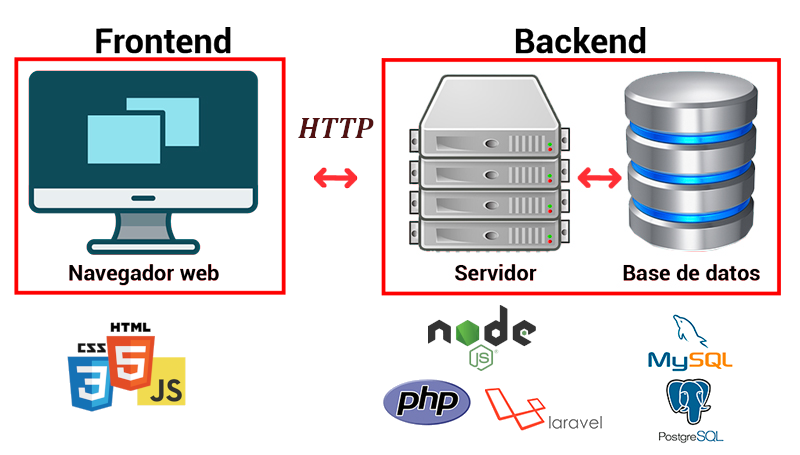
**Comunicación cliente-servidor**

En primer lugar, repasemos el funcionamiento de una apliaciòn web: como vimos en la primera clase, a este tipo de aplicaciones se accede teniendo conexión a internet, empleando un navegador y referenciando la dirección web del sitio (también llamado dominio, por ejemplo https://www.coderhouse.com/). El dominio está asociado a un [servidor](https://youtu.be/4Hcugh7z0cI) , el cual no es más que un ordenador que tiene la aplicación web (se suele decir que “el servidor aloja la aplicación web”). Cuando el navegador del usuario carga el sitio, se puede visualizar la parte frontal de la aplicación (front-end o lado del cliente), con la que se puede interactuar por ejemplo ingresando datos o seleccionado elementos.

Aprendimos que por cada acción importante del usuario, corresponde controlar un evento, el cual utilizamos para capturar entradas, procesarlas y efectuar salidas de interés. No obstante, si necesitamos preservar estos resultados de forma permanente en la aplicación, debemos realizar una solicitud (también llamada petición) a la parte trasera de la aplicación (back-end o parte del servidor), para que procese y almacene los datos necesarios en la [base de datos](https://developer.mozilla.org/es/docs/Web/API/IndexedDB_API/Basic_Concepts_Behind_IndexedDB#database).

Pero tenemos que observar que generalmente existe más de una persona usando una aplicación web, razón por la cual el back-end puede estar recibiendo múltiples peticiones de distintos navegadores (clientes) al mismo tiempo.

Ya que los navegadores (clientes) y servidores necesitan comunicarse constantemente para intercambiar información, existe una serie de reglas que determinan cómo el cliente realiza las peticiones al servidor, las cuales están especificadas en el protocolo de transferencia de hipertexto o HTTP. En el diagrama que veremos a continuación, podemos evidenciar el rol de HTTP como forma de comunicación entre el front-end y el back-end:

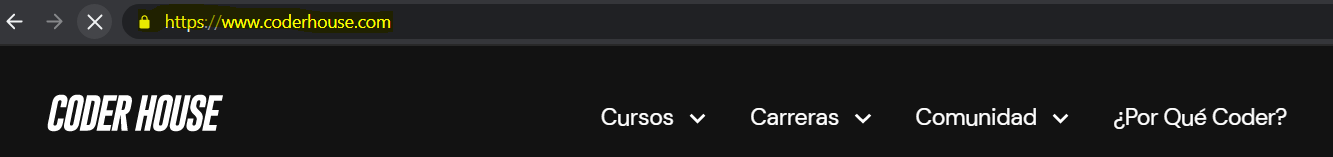


Pero ¿qué necesito para usar HTTP? Nada, tanto el navegador como el servidor conocen estas reglas porque son parte esencial de su funcionamiento. Como programadores/as, emplearemos métodos HTTP (llamados métodos de petición) para acceder o enviar recursos e información al servidor, cuando sea necesario.

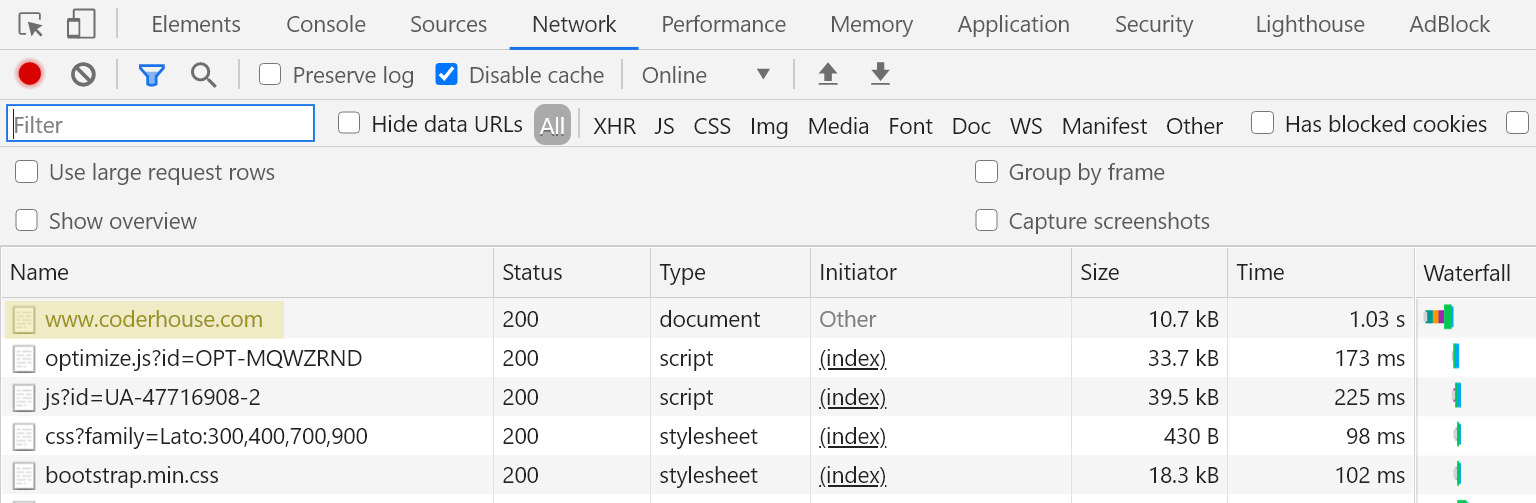
Y ¿cuál es la relación entre AJAX y HTTP? Dijimos que con AJAX procesamos tareas en segundo plano, esto puede incluir realizar petición HTTP sin tener que esperar la respuesta del servidor, pudiendo realizar otras peticiones al mismo tiempo. En los próximos párrafos profundizaremos en dos de los métodos HTTP más empleados.

**Método GET**

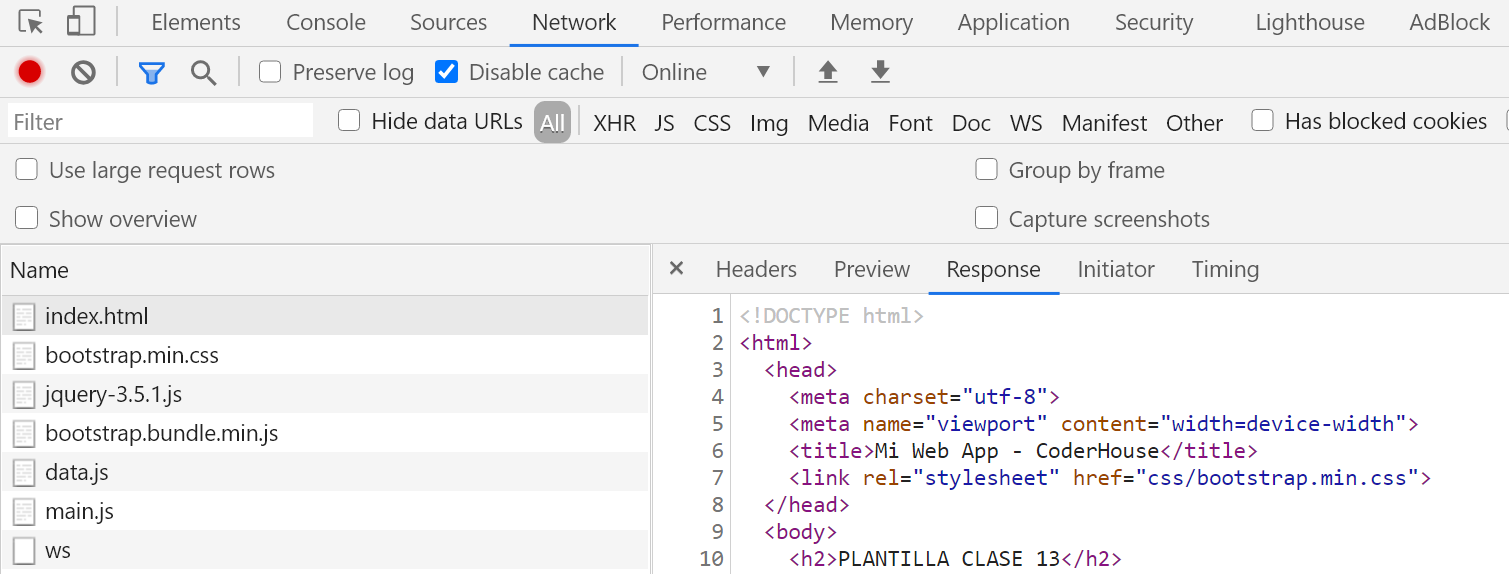
Cada vez que utilizamos el navegador para acceder a una dirección web (por ejemplo <https://www.coderhouse.com/>), estamos usando el método [GET:](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET)



El mencionado método de petición sirve para solicitar un recurso específico del servidor. Las peticiones GET son utilizadas para acceder a datos, páginas o imágenes, entre otros recursos. Si nos dirigimos a la pestaña red (network) de nuestra consola de desarrollo, podremos observar todas las peticiones realizadas por el cliente al momento de solicitar un sitio:



Podemos observar que por cada petición, existe una descripción de estado (status): estos son [códigos de respuesta](https://developer.mozilla.org/es/docs/Web/HTTP/Status) otorgados por el servidor cada vez que el cliente realiza una. El código 200 significa que el recurso solicitado ha sido recuperado, y se transmite en forma de respuesta. Para el caso de una petición GET sobre un documento HTML, el servidor responde la estructura del propio documento, por ejemplo:



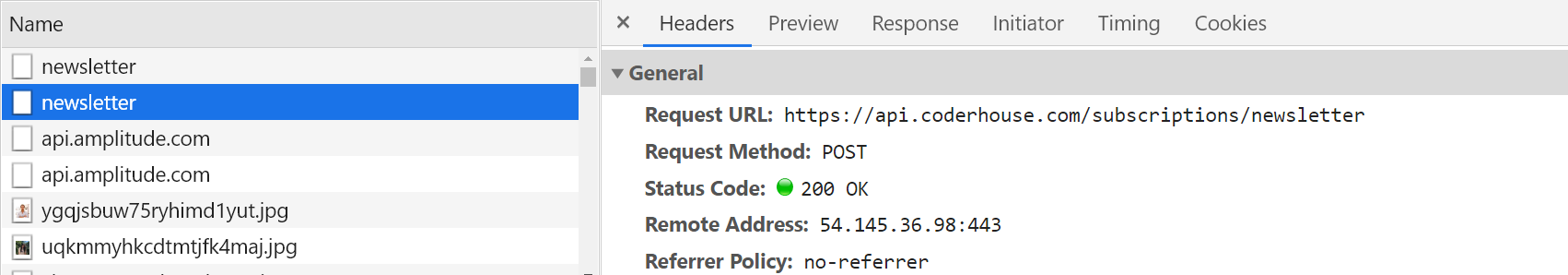
Como programadores/as, empleamos el código de respuesta de la petición GET para saber si el recurso se obtuvo correctamente; de ser así, podemos incluir la respuesta en nuestra aplicación, caso contrario volvemos a solicitar el elemento al servidor, informamos al usuario que el mismo no se cargó correctamente, o no existe.

**Método POST**

Cada vez que enviamos datos usando un formulario, empleamos el método POST, como en el ejemplo, al suscribirse al Newsletter de CoderHouse:

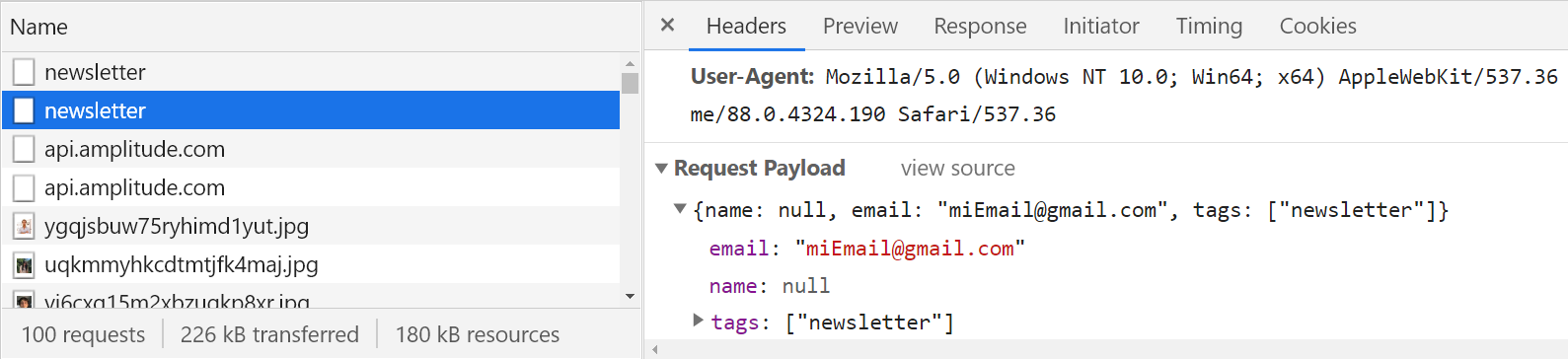


El método POST nos permite enviar datos al servidor, los cuales serán procesados y posiblemente almacenados en una base. Asimismo, para la petición POST contamos con el código de respuesta, el cual será 200 si la información fue procesada correctamente:



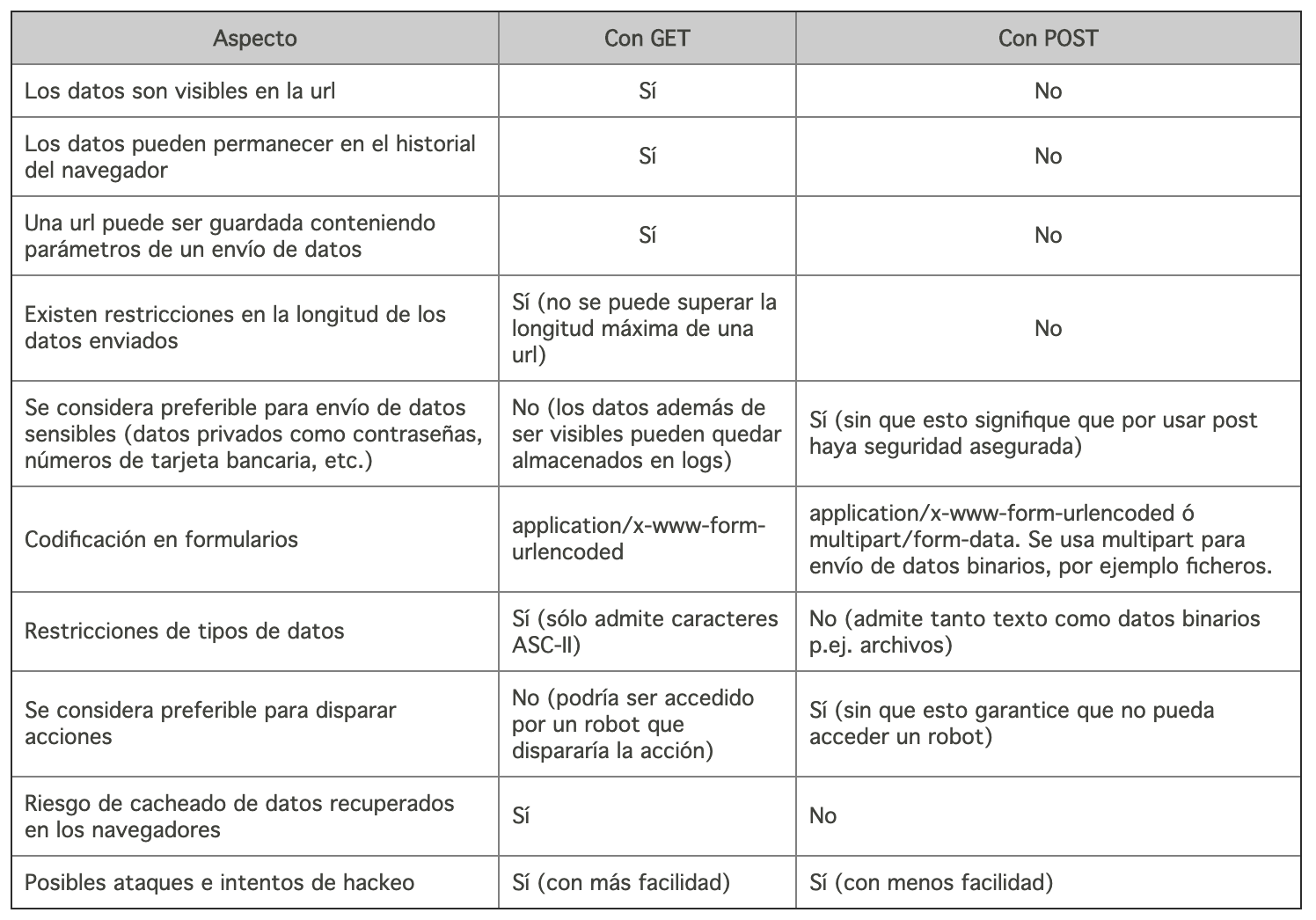
Como programadores/as empleamos el código de respuesta de la petición POST para saber si la informaciòn enviada se procesó correctamente, y en consecuencia efectuar una salida de confirmación de registro.

También es posible visualizar la información enviada durante una petición POST, al igual que el tipo de información (Content-Type), en la parte de la petición HTTP identificada como la cabecera o header:



**Comparación GET y POST**

Si bien existen otros [métodos HTTP](https://developer.mozilla.org/es/docs/Web/HTTP/Methods), GET y POST son los más empleados porque permiten cubrir gran parte de las situaciones de comunicación cliente-servidor. Es decir, solicitar y enviar datos al back-end es una tarea recurrente en nuestra aplicación, por lo cual tenemos la necesidad de conocer cómo funcionan estos métodos, y cuáles son sus diferencias. Para compararlos, podemos tomar la siguiente tabla como referencia:

****

Una de las diferencias más importantes entre ambos es que GET está pensado como método de acceso, razón por la cual las [URLs](https://redfibra.mx/que-es-una-url-y-para-que-sirve-tipos-de-url-y-caracteristicas/) GET pueden ser almacenadas en el historial, o indexadas por buscadores. En cambio las direcciones que emplean POST, al ser este un método reservado al envío de datos en distintos formatos, no pueden utilizarse para navegar.

Cabe aclarar que es posible que una solicitud GET contenga parámetros, lo cual implica enviar cierta información al servidor en la propia dirección (por ejemplo: http://www.domain.com/index.html?parametro1=value1). Generalmente, la información más relevante a almacenar en el servidor se envía usando POST, ya que este método permite determinar el formato de la información a enviar, usando una ruta de acceso no disponible mediante GET,

Por último, en la actualidad existe [HTTPs](https://es.wikipedia.org/wiki/Protocolo_seguro_de_transferencia_de_hipertexto), el cual se encuentra basado en HTTP, siendo un protocolo de comunicación más seguro entre cliente y servidor. En el mismo se encriptan los datos para garantizar la seguridad de las peticiones. No obstante, los métodos de petición son los mismos que HTTP.

*Métodos Ajax con jQuery*

Ahora que contamos con un reconocimiento del funcionamiento de HTTP, es preciso recordar que el objetivo al emplear AJAX es realizar peticiones HTTP al servidor, y procesar las respuestas en segundo plano. Para esto, realizaremos las peticiones usando JavaScript o alguna librería.

En la actualidad existen varias forma de realizar esto, entre las cuales podemos mencionar:

* Objeto [XMLHttpRequest](https://developer.mozilla.org/es/docs/Web/API/XMLHttpRequest) en JavaScript.
* [API fetch](https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch) en JavaScript Vanilla.
* [Mètodos AJAX](http://www.w3bai.com/es/jquery/jquery_ref_ajax.html) en jQuery,
* Librería [Axios](https://github.com/axios/axios) .

Nos centraremos inicialmente en los métodos AJAX con jQuery, ya que son un poco más fáciles de aprender, y comparten características tanto en notación como en funcionamiento con la API fetch, sirviendo también de introducción al concepto de [promesas](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises) en JavaScript.

**¿Cómo usar los métodos AJAX en jQuery?**

Los métodos AJAX más utilizados en jQuery con [$.ajax()](http://www.w3bai.com/es/jquery/ajax_ajax.html), [$.get()](http://www.w3bai.com/es/jquery/ajax_get.html), [$.getJSON()](http://www.w3bai.com/es/jquery/ajax_getjson.html) y [$.post()](http://www.w3bai.com/es/jquery/ajax_post.html). En realidad, $.get(), $.post() y $.getJSON(), son [métodos abreviados](https://api.jquery.com/category/ajax/shorthand-methods/) de casos de uso del método $.ajax(), es decir que se emplean para simplificar peticiones HTTP asíncronas, las cuales podrían realizarse todas con el mismo método.

¿Por qué existen estos métodos? Es sencillo de entender: cuando se repite tanto una configuración, se tiende a simplificar su empleo realizando una función o método para reemplazar instrucciones repetitivas.

Los métodos abreviados de AJAX pueden recibir los siguiente parámetros:

* Una URL (absoluta o relativa) del archivo o dirección que va a procesar la llamada,
* Una función callback para manejar la respuesta de la petición.
* Un array de parámetros (puede no ser necesario en caso de usar GET):

Los datos enviados son adquiridos por el servidor para realizar operaciones, una vez procesados se responde al cliente, desencadenando la llamada a la función callback previamente definida. En el callback de la función podemos definir, por ejemplo, que se efectúe una salida, o se modifique cierta parte de la página, utilizando la información obtenida del servidor. Esto implica la capacidad de actualizar toda o parte de la aplicación, sin la necesidad de recargar esta última o interrumpir las acciones del usuario, determinando un verdadero grado de interactividad.

En cuanto al formato con el que se envían los datos, cabe aclarar que el servidor recibe información en texto plano. Por ende, una forma de estandarizar el envío de datos al backend y recibir como respuesta información detallada es emplear JSON; como ya vimos, es un formato en texto plano que permite representar datos estructurados en la sintaxis de objetos de JavaScript, pudiendo pararlos para operar sobre ellos. En los próximos párrafos presentaremos tres ejemplos de llamadas asíncronas empleando los métodos abreviados de jQuery.

**Llamada Ajax: GET**

Podemos emplear el método $.get() para solicitar un dato puntual del servidor. En el siguiente ejemplo solicitamos cierta informaciòn en formato JSON , disponible en la direccion <https://jsonplaceholder.typicode.com/posts>. Cuando el servidor responde la petición GET, se verifica que el estado es correcto, usando la comparación (estado ==”success” ) en un condicional. De ser así, se efectúa una salida en el DOM usando la información obtenida:

| //Declaramos la url que vamos a usar para el GET  const URLGET = "https://jsonplaceholder.typicode.com/posts"  //Agregamos un botón con jQuery  $("body").prepend('<button id="btn1">GET</button>');  //Escuchamos el evento click del botón agregado  $("#btn1").click(() => {  $.get(URLGET, function (*respuesta*, *estado*) {  if(*estado* === "success"){  let misDatos = *respuesta*;  for (const dato of misDatos) {  $("body").prepend(`<div>  <h3>${dato.title}</h3>  <p> ${dato.body}</p>  </div>`);  }  }  });  });  "font-size": "40px",  "borderLeft": "5px solid #ccc" }); |
| --- |

**Llamada Ajax: POST**

En el caso del método POST, el segundo parámetro comprende un valor o variable que podemos enviar al servidor al momento de realizar la llamada. Para verificar el funcionamiento de post, empleamos la URL <https://jsonplaceholder.typicode.com/posts>, dirección que nos responde con los mismos datos enviados, si bien se supone que al realizar POST existe un procesamiento y almacenamiento de la información enviada. Así, sólo podemos simular el post enviando cierta información y manejando la respuesta con la función callback, ya que no contamos con nuestro propio back-end de aplicación. En consecuencia, nuestro manejo de respuesta del POST se limita a informar al usuario que se guardó correctamente la información enviada con una salida modificando el DOM:

| //Declaramos la url que vamos a usar para el GET  const URLGET = "https://jsonplaceholder.typicode.com/posts"  //Declaramos la información a enviar  const infoPost = { nombre: "Ana", profesion: "Programadora" }  //Agregamos un botón con jQuery  $("body").prepend('<button id="btn1">POST</button>');  //Escuchamos el evento click del botón agregado  $("#btn1").click(() => {  $.post(URLGET, infoPost ,(*respuesta*, *estado*) => {  if(*estado* === "success"){  $("body").prepend(`<div>  Guardado:${*respuesta*.nombre}  </div>`);  }  });  }); |
| --- |

**Llamada Ajax: JSON estático**

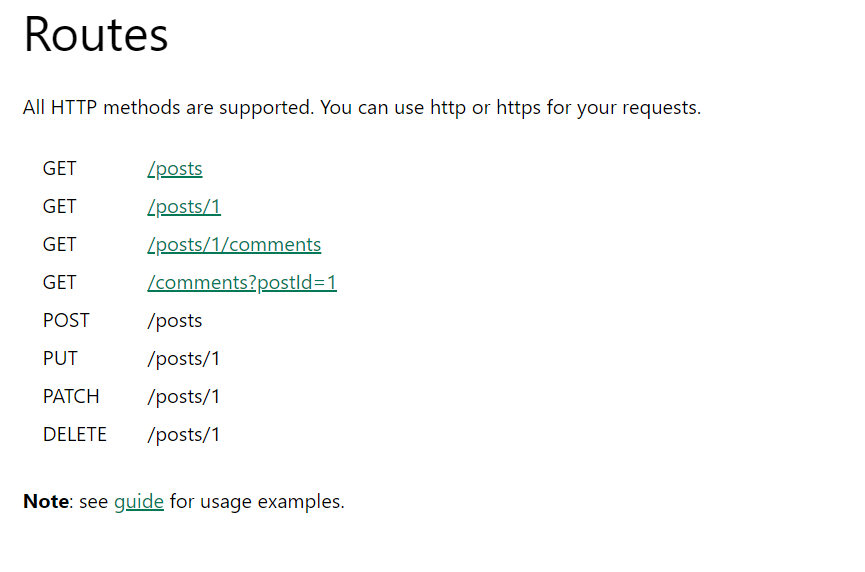
Dado que a veces contamos con archivos .json en nuestro proyecto, estos podrían cargarse empleando el método getJSON. El método funciona de forma similar a get, pero es preferible si se opta por cargar archivos con este formato de forma local en el proyecto.

De la misma forma que en las otras peticiones, al culminar la carga del archivo local por el servidor se dispara el método callback, realizando un modificacion en el DOM si la peticion fue exitosa:

| //Declaramos la url del archivo JSON local  const URLJSON = "data/datos.json"  //Agregamos un botón con jQuery  $("body").prepend('<button id="btn1">JSON</button>');  //Escuchamos el evento click del botón agregado  $("#btn1").click(() => {  $.getJSON(URLJSON, function (*respuesta*, *estado*) {  if(*estado* === "success"){  let misDatos = *respuesta*;  for (const dato of misDatos) {  $("body").prepend(`<div>  <h3>${dato.name}</h3>  <p> ${dato.email}</p>  </div>`)  }  }  });  }); |
| --- |

*API*

Una API (Application Programming Interfaces) es una aplicación web construida en base a la arquitectura [API REST](https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional), a la cual podemos solicitar y enviar información desde el cliente. Si tomamos por ejemplo la API de [JSONPlaceholder](https://jsonplaceholder.typicode.com/), podremos notar que existen distintas rutas que podemos usar:



Cada una de estas direcciones puede ser empleada teniendo en cuenta el tipo de método HTTP soportado, es decir que la llamada debe usar el método señalando: si es GET la llamada asíncrona con jQuery es $.get(); en cambio, si la ruta es POST, el método de llamada es $.post().

Pueden existir direcciones que pueden ser llamadas con màs de un método HTTP (como es el caso de /post en [JSONPlaceholder](https://jsonplaceholder.typicode.com/)) pero esta configuración depende del servidor, en específico depende de cómo el backend del api está construido y que métodos HTTP admite en cada ruta.

En una aplicación moderna, es común requerir que el frontend interactúe con una o más APIs. Ya que las APis permiten conectar sistemas o partes de sistemas web entre si. Por ejemplo: una aplicación e-commerce que permite pagar con MercadoPago, usarà el [API de Mercado Pago](https://www.mercadopago.com.ar/developers/es/guides/online-payments/checkout-api/introduction) para registrar la compra del producto.

Hay que tener en cuenta que cada API está construida de forma distinta, teniendo distintas reglas de acceso. Si bien siguen un convención de construcción conocida como [API REST](https://es.wikipedia.org/wiki/Transferencia_de_Estado_Representacional), es necesario que el programador/a determine qué APIs necesita emplear en el sistema, por qué y cómo emplear las llamadas asíncronas para enviar y obtener datos de ellas correctamente.