





 [arielZarate](#) / [HenryAll](#) Public

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

 [main](#) ▾

...


[HenryAll](#) / [FT-M2-master](#) / [04-Ajax](#) /

 arielZarate ...	25 days ago 
..	
 demo	27 days ago
 homework	25 days ago
 README.json	27 days ago
 README.md	27 days ago


 [README.md](#)







Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

AJAX

Que es AJAX?

AJAX son las siglas de Asynchronous JavaScript and XML. XML es raramente relevante, pero cuando desarrollamos aplicaciones web, usamos Ajax para hacer cosas asincrónicas como actualizar una página, hacer acciones, etc.

En resumen, AJAX se trata de actualizar partes de una página web sin tener que recargar toda la web. Eso es muy útil si tu sitio web es grande, no querrá que tus usuarios tengan que cargar la misma información varias veces.

En que se basa Ajax ?

AJAX se basa en un montón de tecnologías. No tienes que ser un experto en todas ellas pero un poco de conocimiento de fondo será útil. Esto debería darte los antecedentes suficientes para que te hagas una idea de AJAX.

Las tecnologías que forman AJAX son

- XHTML y CSS, para crear una presentación basada en estándares.
- DOM, para la interacción y manipulación dinámica de la presentación.
- XML, XSLT y JSON, para el intercambio y la manipulación de información.
- XMLHttpRequest, para el intercambio asíncrono de información.
- JavaScript, para unir todas las demás tecnologías.

Para que necesitamos Ajax?

Piensa en toda tu aplicación web como un restaurante de comida rápida. Tú eres el cajero, la persona en las primeras líneas. Manejas las **solicitudes** de los cliente



Si miras este diagrama, puedo ver tres trabajos separados que deben hacerse.

1. El cajero debe manejar las solicitudes de los usuarios a un ritmo rápido.
2. Necesita que los cocineros tiren las hamburguesas a la parrilla y cocinen toda la comida.
3. Necesitas un equipo de preparación de comida para empaquetar la comida y ponerla en una bolsa o en una bandeja.

Sin embargo, si no tuvieras AJAX, sólo se te permitiría procesar un pedido a la vez de principio a fin! Tendrías que tomar el pedido... luego cobrar al cliente... luego sentarte ahí sin hacer nada mientras la gente en la cocina cocina la comida... y luego seguir esperando mientras el equipo de preparación de la comida la empaqueta. Sólo podrías tomar el siguiente pedido después de todo eso.



Eso es una mala experiencia para el usuario! Ya no podrías llamarlo "comida rápida". En su lugar, tendrías que llamarlo "comida mediocre"... o algo así.

AJAX permite un **modelo de procesamiento asincrónico**. Eso significa que puedes pedir datos o enviarlos sin cargar la página entera. Esto es como la forma en que funciona un restaurante de comida rápida normal. Como cajero, tomas el pedido del cliente, lo envías al equipo de cocina, y te preparas para tomar el siguiente pedido del cliente.

Los clientes pueden seguir haciendo pedidos, y no es necesario sentarse allí mientras los empleados de la cocina trabajan y hacen esperar a todo el mundo.

Esto ciertamente introduce cierta complejidad. Ahora tienes múltiples especializaciones dentro del restaurante. Además, los pedidos se están manejando a ritmos diferentes. Pero, crea una experiencia de usuario mucho mejor.



Probablemente has visto esto en acción en un restaurante. Una persona está trabajando en la máquina de papas fritas. Una persona está manejando la parrilla. Cuando llega un pedido, el cajero puede comunicarse instantáneamente con ambos y volver a tomar los pedidos.

Como crear una solicitud POST

Pongamos estos conceptos a trabajar. Como cajera, debes enviar las solicitudes de los clientes a la cocina para que el resto de tu equipo pueda preparar la comida. Puedes hacer eso con la solicitud POST.

En tu código real, una solicitud POST envía datos a tu servidor. Eso significa que estás enviando los datos del pedido al back-end, en este caso.

Tiene tres partes principales:

1. **Una URL:** esta es la ruta que seguirá la solicitud. Más en un minuto.
2. **Datos:** cualquier parámetro extra que necesites enviar al servidor.
3. **Callback:** Lo que pasa después de que hayas enviado la solicitud

¿Cuáles son algunas de las cosas comunes que la gente pide en un restaurante de comida rápida? Veamos dos ejemplos:

1. Papas Fritas
2. Un combo de una hamburguesa, papas fritas y una bebida

Estos dos requieren procesos diferentes. Una solicitud de papas fritas podría necesitar sólo una persona para meter algunas papas fritas en una manga. Pero un pedido de comida combinada requerirá el trabajo de varios miembros del equipo. Por lo tanto, estos dos necesitan diferentes URLs.

```
$.post('/comboMeal')$.post('/fries')
```

La URL nos permite usar la misma lógica en el back-end para ciertos tipos de solicitudes. Esa parte está fuera del alcance de este tutorial, por lo que puedes profundizar un poco más en ella cuando mires el back-end.

Lo siguiente es la **data**. Este es un **objeto** que nos dice un poco más sobre la petición. Para la URL de la comida combinada, probablemente necesitamos saber:

1. El tipo de comida principal
2. El tipo de bebida
3. El precio
4. Cualquier petición especial

Por las papas fritas, puede que sólo necesitemos saberlo:

1. El tamaño de las patatas
2. El precio



Veamos un ejemplo de un combo de: una hamburguesa con queso con una Pepsi que cuesta 6 dólares. Esto es lo que parece en JavaScript.

```
let order = {  
  mainMeal: 'cheeseburger',  
  drink: 'Pepsi',  
  price: 6,  
  exceptions: ''  
};$.post('/comboMeal', order);
```

La variable *orden* contiene el contenido del orden. Y luego lo incluimos en el pedido POST para que nuestro personal de cocina sepa qué diablos poner en el combo de comida!

¡Pero no podemos hacer que todo este código se ejecute al azar! Necesitamos un evento de activación que active la solicitud. En este caso, un pedido de un cliente en un restaurante de comida rápida es como una persona que hace clic en un botón de "pedido" en su sitio web. Podemos usar el evento `click()` de jQuery para ejecutar el POST cuando el usuario hace clic en un botón.

```
$('#button').click(function(){  
  let order = {  
    mainMeal: 'cheeseburger',  
    drink: 'Pepsi',  
    price: 6,  

```

```
    exceptions: ''  
  };  
  
  $.post('/comboMeal', order);  
});
```

La última parte. Tenemos que decirle algo al cliente después de que su pedido haya sido enviado. Los cajeros suelen decir "¡El próximo cliente por favor!" ya que este es un restaurante de comida rápida, así que podemos usar eso dentro de la llamada para mostrar que el pedido ha sido enviado.

```
$('#button').click(function(){  
  let order = {  
    mainMeal: 'cheeseburger',  
    drink: 'Pepsi',  
    price: 6,  
    exceptions: ''  
  };  
  $.post('/comboMeal', order, function(){  
    alert('Next customer please!');  
  });  
});
```

Como crear una solicitud GET

Hasta ahora, tenemos la capacidad de presentar una orden. Ahora, necesitamos una forma de entregar ese pedido a nuestro cliente.

Aquí es donde entran las solicitudes GET. GET nos permite solicitar datos del servidor (o de la cocina, esta analogía). Tenga en cuenta: en este momento, nuestra base de datos está llena de pedidos, no la comida en sí. Esta es una distinción importante porque las solicitudes de GET no cambian nuestra base de datos. Sólo entregan esa información al front-end. Las solicitudes POST cambian la información de la base de datos.

Estas son algunas de las preguntas típicas que te pueden hacer antes de recibir tu comida.

1. ¿Te gustaría comer aquí o recibir la comida para llevar?
2. ¿Necesitas algún condimento (como ketchup o mostaza)?
3. ¿Cuál es tu número en el recibo (para verificar que es tu comida)?

Digamos que ordenó tres comidas compuestas para su familia. Quieres comer la comida en el restaurante. Necesitas ketchup. Y el número en su recibo es 191.

Podemos crear una solicitud GET con una URL de '/comboMeal', que corresponde a la solicitud POST junto con la misma URL. Sin embargo, esta vez necesitamos datos diferentes. Es un tipo de solicitud totalmente diferente. El mismo nombre de URL sólo nos permite organizar mejor nuestro código.

```
let meal = {  
  location: 'here',  
  condiments: 'ketchup',  
  receiptID: 191  
};$.get('/comboMeal', meal);
```



También necesitamos un disparador para este. Esta solicitud se activa cuando los clientes responden a tus preguntas como cajero antes de que les entregues la comida. No hay una forma conveniente de representar las preguntas y respuestas con JavaScript. Así que voy a crear otro evento de clic para el botón con la clase "respuesta".

```
$('.answer').click(function(){  
  let meal = {  
    location: 'here',  
    condiments: 'ketchup',  
    idNumber: 191,  
  };$.get('/comboMeal', meal);  
});
```



Este también necesita una función de devolución de llamada, porque vamos a recibir lo que estaba contenido en las tres comidas compuestas en el orden 191. Podemos recibir esos datos a través de un parámetro de *datos* en nuestra llamada de retorno.

Esto nos devolverá lo que sea que la retrollamada estipule para la orden 191. Voy a usar una función llamada *comer* para significar que eventualmente se puede comer la comida, pero ten en cuenta que no hay una función de comer en JavaScript!

```
$('.answer').click(function(){  
  let meal = {  
    location: 'here',  
    condiments: 'ketchup',  
    idNumber: 191,  
  };  
  
  //data contains the data from the server  
  $.get('/comboMeal', meal, function(data){  
    //eat is a made-up function but you get the point  
    eat(data);  
  });  
});
```

El producto final, *datos*, contendría el contenido de las tres comidas combinadas, teóricamente. ¡Depende de cómo esté escrito en el backend!



Eventos en javascript

Como se mencionó anteriormente, los **eventos** son acciones u ocurrencias que suceden en el sistema que está programando — el sistema disparará una señal de algún tipo cuando un evento ocurra y también proporcionará un mecanismo por el cual se puede tomar algún tipo de acción automáticamente (p.e., ejecutando algún código) cuando se produce el evento. Por ejemplo, en un aeropuerto cuando la pista está despejada para que despegue un avión, se comunica una señal al piloto y, como resultado, comienzan a pilotar el avión.

En el caso de la Web, los eventos se desencadenan dentro de la ventana del navegador y tienden a estar unidos a un elemento específico que reside en ella — podría ser un solo elemento, un conjunto de elementos, el documento HTML cargado en la pestaña actual o toda la ventana del navegador. Hay muchos tipos diferentes de eventos que pueden ocurrir, por ejemplo:

- El usuario hace clic con el mouse sobre un elemento determinado o coloca el cursor sobre un elemento determinado.
- El usuario presiona una tecla en el teclado.
- El usuario cambia el tamaño o cierra la ventana del navegador.
- Una página web termina de cargar.
- Un formulario se envía
- Un video se reproduce, pausa o finaliza la reproducción.
- Un error ocurre.

Se deducirá de esto (y echar un vistazo a MDN [Referencia de eventos](#)) que hay **muchos** eventos a los que se puede responder.

Cada evento disponible tiene un **controlador de eventos**, que es un bloque de código (generalmente una función JavaScript definida por el usuario) que se ejecutará cuando se active el evento. Cuando dicho bloque de código se define para ejecutarse en respuesta a un disparo de evento, decimos que estamos **registrando un controlador de eventos**. Tenga en cuenta que los controladores de eventos a veces se llaman **oyentes de eventos** — son bastante intercambiables para nuestros propósitos, aunque estrictamente hablando, trabajan juntos. El oyente escucha si ocurre el evento y el controlador es el código que se ejecuta en respuesta a que ocurra.

Veamos un ejemplo simple para explicar lo que queremos decir aquí. Ya has visto eventos y controladores de eventos en muchos de los ejemplos de este curso, pero vamos a recapitular solo para consolidar nuestro conocimiento. En el siguiente ejemplo, tenemos un solo `<button>`, que cuando se presiona, hará que el fondo cambie a un color aleatorio:

```
<button>Cambiar color</button>
```

El JavaScript se ve así:

```
var btn = document.querySelector('button');

function random(number) {
  return Math.floor(Math.random()*(number+1));
}

btn.onclick = function() {
  var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
  document.body.style.backgroundColor = rndCol;
}
```

En este código, almacenamos una referencia al botón dentro de una variable llamada `btn`, usando la función `Document.querySelector()`. También definimos una función que devuelve un número aleatorio. La tercera parte del código es el controlador de eventos. La variable `btn` apunta a un elemento, y este tipo de objeto tiene una serie de eventos que pueden activarse y, por lo tanto, los controladores de eventos están disponibles. Estamos escuchando el disparo del evento `click`, estableciendo la propiedad del controlador de eventos `onclick` para que sea igual a una función anónima que contiene código que generó un color RGB aleatorio y establece el color de fondo igual a este.

Este código ahora se ejecutará cada vez que se active el evento `click` en el elemento, es decir, cada vez que un usuario haga clic en él.

Event Loop

Antes que nada miremos un dibujo que representa el runtime de v8 (el runtime que usa chrome y node)



Como se puede ver en la imagen, el engine consiste de dos elementos principales

- Memory Heap: es donde se realiza la aloca  n de memoria
- Call Stack: es donde el runtime mantiene un track de las llamadas a las funciones

Solo hablaremos de la call stack, que es la que se relaciona con el Event Loop.

Call Stack

Para los que no sepan un stack (también llamado pila) es una estructura simple, similar a un arreglo en el que solo se puede agregar items al final (push) , y remover el último (pop).

El proceso que realiza el call stack es simple, cuando se está a punto de ejecutar una función, esta es añadida al stack. Si la función llama a su vez, a otra función, es agregada sobre la anterior. Si en algún momento de la ejecución hay un error, este se imprimirá en la consola con un mensaje y el estado del call stack al momento en que ocurrió.

Javascript es un lenguaje single threaded. Esto quiere decir que durante la ejecución de un script existe un solo thread que ejecuta el código. Por lo tanto solo se cuenta con un call stack

Veamos un ejemplo:

```
function multiply (x, y) {  
  return x * y;  
}  
  
function printSquare (x) {  
  var s = multiply(x, x);  
  console.log(s);  
}  
  
printSquare(5);
```

Los estados del call stack serían:



Y que pasa si tenemos una función de esta manera:

```
function foo() {  
  foo();  
}  
  
foo();
```



Lo que sucedería es que en algún momento la cantidad de funciones llamadas excede el tamaño del stack , por lo que el navegador mostrará este error:



Pero qué pasa si llamamos a un timeout o hacemos un request con AJAX a un servidor. Al ser un solo thread, hay un solo call stack y por lo tanto solo se puede ejecutar una cosa a la vez. Es decir el navegador debería congelarse, no podría hacer más nada, no podría renderizar, hasta que la llamada termine de ejecutarse. Sin embargo esto no es así, javascript es asíncrono y no bloqueante. Esto es gracias al Event Loop.

Event Loop

Algo interesante acerca de javascript, o mejor dicho de los runtimes de javascript, es que no cuentan nativamente con cosas como setTimeout, DOM, o HTTP request. Estas son llamadas web apis, que el mismo navegador provee, pero no están dentro del runtime JS.

Por lo tanto este es el gráfico que muestra una visión más abarcativa de javascript. En este se puede ver el runtime, más las Web APIs y el callback queue del cual hablaremos más adelante.



Al haber un solo thread es importante no escribir código bloqueante para la UI no quede bloqueada.

Pero ¿Cómo hacemos para escribir código no bloqueante?

La solución son callbacks asíncronas. Para esto combinamos el uso de callbacks (funciones que pasamos como parámetros a otras funciones) con las WEB API's.

Por ejemplo:

```
console.log("hola");

setTimeout(function timeoutCallback() {

  console.log("mundo");

}, 500);

console.log("Ubykuo, everytime, everywhere");

/*
 * Resultados:
 * => hola
 * => Ubykuo, everytime, everywhere
 * => mundo
 */
```

Como pueden ver la ejecución no se queda bloqueada en `setTimeout()` ya que imprime la instrucción que le sigue primero) ¿Pero entonces cómo es que posible que esto sea así si solo existe un solo thread? ¿Cómo es que la ejecución continua y al mismo tiempo el `setTimeout` hace la cuenta regresiva para ejecutar la función pasada como callback?

Esto es porque, como mencione anteriormente, el `setTimeout` NO es parte del runtime. Sino que es provista por el navegador como WEB APIs (o en el caso de Node por `c++` apis). Los cuales SI se ejecutan en un thread distinto.

¿Como se maneja esto con una única call stack?

Existe otra estructura donde se guardan las funciones que deben ser ejecutadas luego de cierto evento (timeout, click, mouse move), en el caso del código de ejemplo de arriba se guarda que, cuando el timeout termine se debe ejecutar la función `timeoutCallback()`. Tener en cuenta que cuando sucede el evento, esta estructura no es la que la ejecuta y tampoco las agrega al call stack ya que sino podría pasar que la función se ejecutará en medio de otro código. Lo que hace es enviarla a la Callback Queue.

Lo que hace el event loop es fijarse el call stack, y si está vacío (es decir no hay nada ejecutandose) envía la primera función que esté en la callback queue al call stack y comienza a ejecutarse.

Luego de terminar la cuenta regresiva del `setTimeout()` (que no es ejecutada en el runtime de javascript), `timeoutCallback()` será enviada a la callback queue. El event loop chequeara el Call Stack, si este está vacío enviará `timeoutCallback()` al call stack para su ejecución.

El flujo en imágenes de todo este trabalenguas seria:

De esta manera se logra que el código sea no bloqueante, en vez de un `setTimeout` podría ser una llamada a un servidor, en donde habría que esperar que se procese nuestra solicitud y nos envíe una respuesta , el cual sería tiempo ocioso si no contáramos con callbacks asincronicas, de modo que el runtime pueda seguir con otro código. Una vez que la respuesta haya llegado del servidor y Call Stack esté vacío, se podrá procesar la respuesta (mediante la función pasada como callback) y hacer algo con ella , por ejemplo mostrarla al usuario.

Este video explica muy bien el Event Loop <https://www.youtube.com/watch?v=8aGhZQkoFbQ>

¿Por que si bloqueamos el call stack la ui ya no responde más?

Esto se debe a que el navegador intenta realizar un proceso de renderizado cada cierto tiempo. Pero este no puede realizarse si hay código en el stack. El proceso de renderizado es similar a una callback asincrónica, ya que debe esperar a que el stack está vacío, es como una función más en la Callback Queue (aunque con cierta prioridad). Por lo que si hay código bloqueante, el proceso de renderizado tardará más en realizarse y el usuario no podrá hacer nada, no podrá seleccionar texto, no podrá ingresar texto, no podrá apretar un botón.

¿Que pasaria si a un usuario que interactuando con nuestra página le sucediera esto?

Lo más probable es que cierre el navegador y nunca más vuelva a entrar a nuestra página. No es algo que queremos que suceda.

Homework

Completa la tarea descrita en el archivo [README](#)
