

[Get started](#)[Open in app](#)

## Traña Michael

16 Followers · [About](#) [Follow](#)

# Asincronismo en JavaScript

Entendiendo los callback's, promises & async/await



[Traña Michael](#) Feb 13 · 8 min read



Hoy en día **JavaScript** es uno de los lenguajes mas utilizados en el mundo del desarrollo de software. Esto es debido a dos motivos: hasta “hace poco” era el único lenguaje de programación capaz de correr en los navegadores web y a su versatilidad para correr en múltiples plataformas. Por este motivo no es mala idea aprenderlo en este 2020. Así que he decidido hacer una serie de post's acerca de esta herramienta.

Uno de los conceptos con mayor relevancia en JavaScript, y en casi todos los lenguajes de programación, es el **asincronismo**. Sería una “herejía” pensar que, en pleno 2020, sigamos desarrollando aplicaciones sin asincronismo. **Pero ¿qué es asincronismo?** Estas son tareas, cuyas operaciones son resultas fuera de nuestra aplicación, que al ser ejecutadas pueden ser completadas en el momento o en el futuro y que su comportamiento es no bloqueante. Pensemos en lo siguiente: si nuestra app solicita el catálogo de productos, esta envía una orden a través de una API, pero es el servidor quién procesa dicha solicitud y en el momento que este listo devolverá el listado de datos. En este punto nuestra app desconoce el tiempo que se toma el servidor en responder la petición, por lo que sí utilizáramos una función síncrona se estaría todo ese tiempo esperando los datos. Bueno, una función asíncrona enviará la petición a la API, nuestra aplicación continuará con la ejecución de otras tareas y cuando el servidor responda se notificará que los datos están listos para ser procesados.

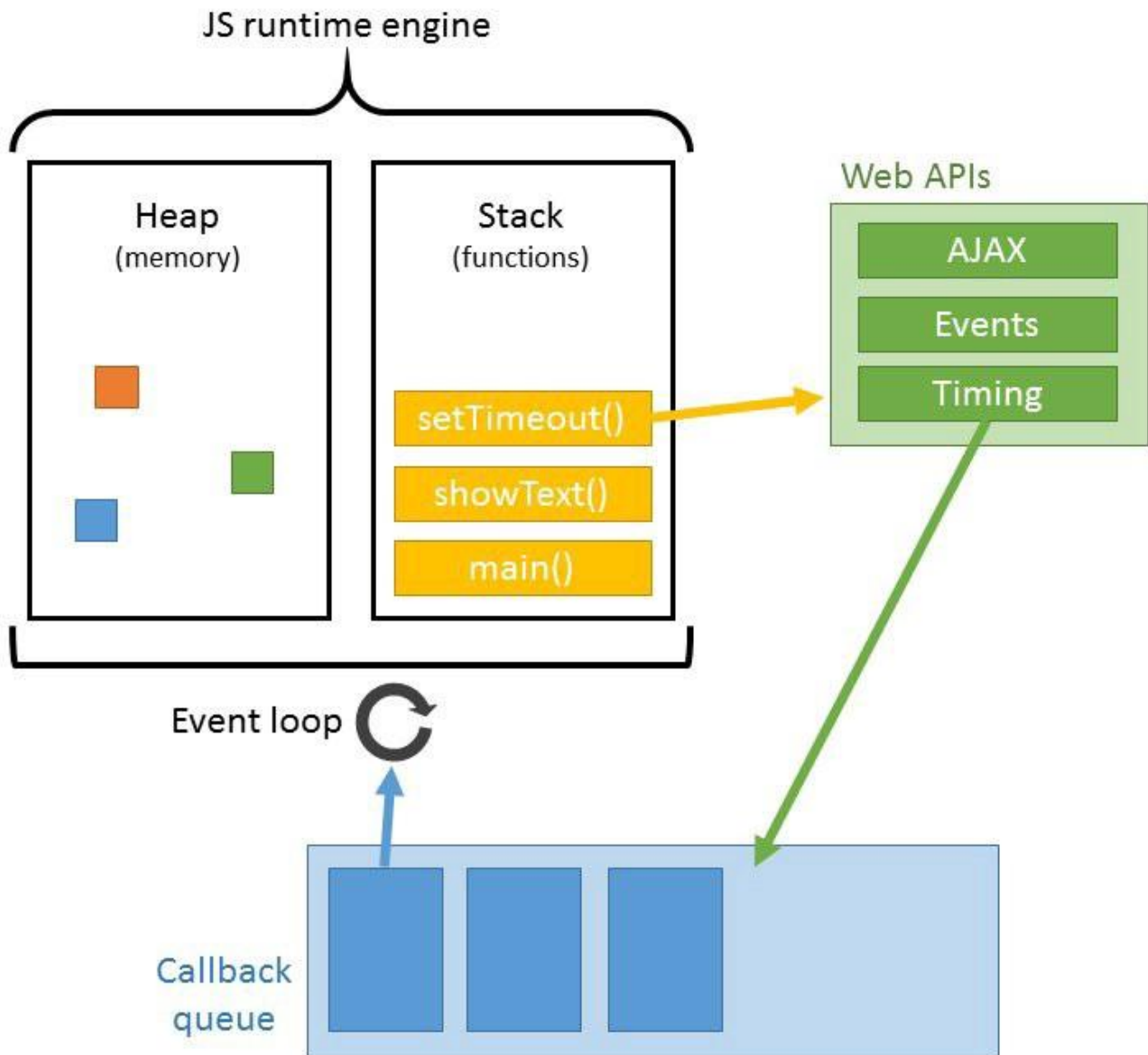
Para entender este concepto primero tenemos que tener claro que JavaScript no es capaz de ejecutar más de una tarea a la vez, sin importar la cantidad core's que tenga tu computadora o servidor. A pesar de que JavaScript no es un lenguaje multi-tarea, o con paralelismo, es altamente concurrente sin importar que tú equipo solo cuente con un procesador de un núcleo. Quizás usted, como programador, se pregunte ¿como es posible emplear concurrencia en un único core? A pesar de lo que se suele creer, la respuesta es que no es necesario contar con más de un núcleo para poder emplearlo. Por ejemplo, una técnica bastante usada es el entrelazado en el que dos tareas grandes se dividen en varias pequeñas y se ejecutan una del primer proceso seguida de otra del siguiente proceso, finalizándolas casi en simultáneo.

## Internamente ¿Cómo funciona el asincronismo en JavaScript?

Técnicamente definiremos JavaScript como un lenguaje de programación con un modelo de concurrencia asíncrono y no-bloqueante implementado con un **Event Loop** en un único hilo o thread de ejecución. A pesar de que JavaScript no es multitareas este puede delegar la ejecución de ciertas funciones a otros procesos y cuya llamada será devuelta inmediatamente evitando así el bloqueo de la aplicación, cuando dicho proceso halla sido capaz de resolver la tarea enviará un mensaje de notificación y procesará la respuesta mediante un **callback, promise o evento**.

Para comprender su funcionamiento debemos añadir dos componentes más: **call stack** y **callback queue**. La **pila de llamadas** o **call stack** es donde se apilan las llamadas a funciones según el orden en el que llegan, ejecutándose primero las que se encuentran encima; por ejemplo: si una función hace un llamado a otra función, esta última estará

en la parte superior y será la que se ejecute primero. Anteriormente habíamos hablado que podíamos delegar tareas a otros procesos que no bloquearan la aplicación y cuyos resultados nos fueran notificados mediante mensajes para procesar las respuestas con **callback's**, pues dichos callback's serán las funciones que se encolarán en el **callback queue** y en el momento que se hallan procesado todas las funciones de la pila de llamadas será el momento en el que las funciones que están en el **callback queue** se envíen al call stack para ser ejecutadas.



Interacción de los diferentes componentes del **Runtime Engine** de JavaScript

## Formas de emplear asincronismo en JavaScript

Existen 3 maneras con las que podemos implementar asincronismo, estas son:  
Utilizando callback's, promises y con `async/await`.

### Callback's

Esta es la forma mas antigua para emplear asincronismo en JavaScript. Y consiste en una función que se pasará como parámetro de otra función, en donde la función principal será la encargada de hacer el llamado de nuestro **callback** (la función parámetro). Cuando esta función principal es no-bloqueante, utilizará este callback para dar respuesta hasta el momento en que los resultados estén listos, hallan sido satisfactorios o no.

Por ejemplo, si queremos consultar datos en Internet lo normal sería que consumamos una API, el resultado estará listo en un tiempo indeterminado por lo que sería lógico emplear asincronismo en este caso, nuestra función no bloqueará el hilo principal de nuestra aplicación y cuando tengamos la respuesta de la API llamaremos al callback para procesar el resultado; ya sea que obtuvimos los datos o un error.

Veamos el siguiente ejemplo:

```
1  function funcionPrincipal(url, callback) {
2      let solicitud = new XMLHttpRequest();
3      solicitud.open('GET', url, true);
4      solicitud.onreadystatechange = (event) => {
5          if(solicitud.readyState === 4) {
6              if (solicitud.status >= 200 && solicitud.status < 400) {
7                  let respuesta = solicitud.responseText;
8                  let datos = JSON.parse(respuesta);
9                  callback(datos, null);
10             }
11             else {
12                 error(null, 'Ha ocurrido un error');
13             }
14         }
15     }
16     solicitud.send();
17 }

18
19 function callback(datos, error) {
20     if(error) {
21         console.log(error);
22     }
23     else {
24         datos.results.forEach(item => {
25             console.log(item.name);
26         });
27     }
28 }
29
```

```
30  funcionPrincipal('https://rickandmortyapi.com/api/character/?page=19', callback);
```

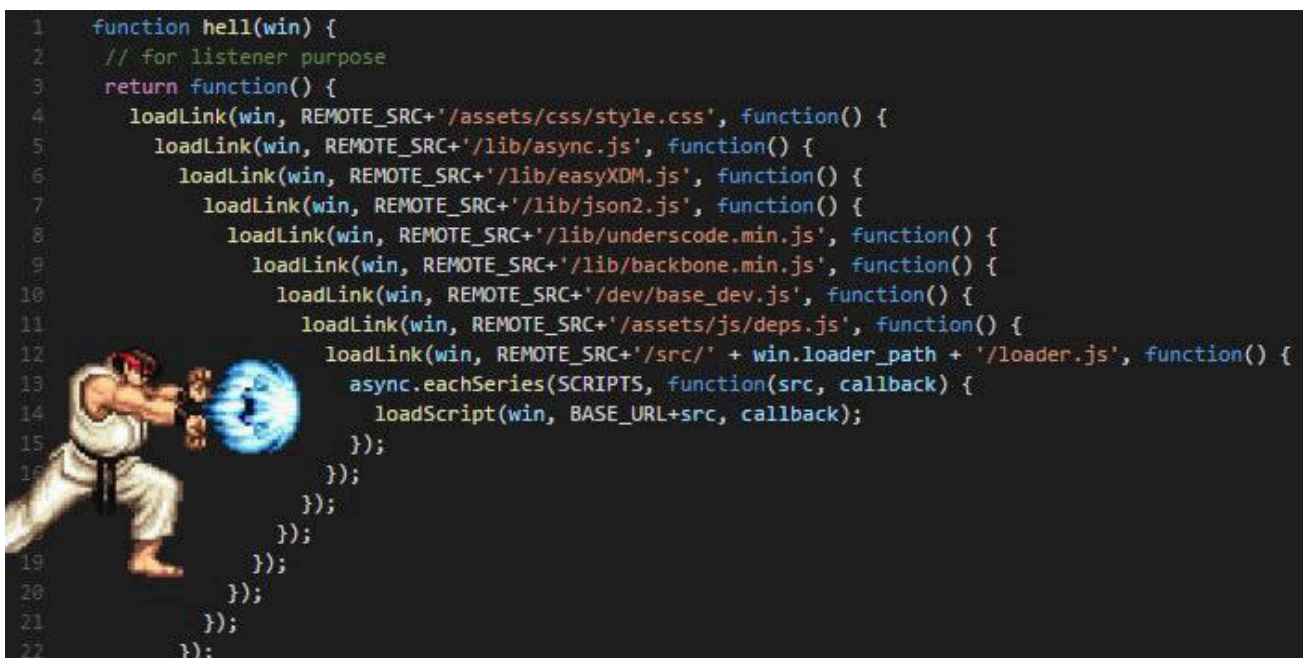
Consumiendo una API REST con asincronismo basado en callbacks

Acá podemos apreciar como emplear asincronismo con callback's, tenemos una función denominada **funcionPrincipal**, esta función recibe 2 parámetros la URL de una API en internet y el callback. Luego tenemos otra función llamada **callback** la cual recibe dos parámetros: los datos si el resultado es satisfactorio y el error en caso de que surja un problema. Por último tenemos una llamada a la función principal pasándole la URL y el callback.

Por último, para comprobar que nuestro método **funcionPrincipal** es asíncrona podemos añadir unas líneas más de código síncrono en donde apreciaremos como no se ven bloqueadas por la llamada de esta función.

*Nota: esta característica cuenta con una desventaja y es que en casi todos los sistemas medianamente complejos tendremos tareas en las que necesitaremos combinar múltiples operaciones asíncronas (Ejemplo: al guardar una venta es muy probable que tengamos que guardar los datos de la venta, los datos de los productos que se vendieron, actualizar el stock de productos, etc.), en donde la siguiente dependa de éxito de la anterior. Cada vez que llamemos una función asíncrona habrá que pasar el callback, comprobar que la respuesta fue satisfactoria, hacer la siguiente llamada asíncrona y así sucesivamente. Esto se conoce como **callback hell** y es debido a que nuestro código se puede volver caótico sino controlamos la cantidad de encadenamientos que realicemos.*

En la siguiente imagen se ilustra el problema.



```

1  function hell(win) {
2    // for listener purpose
3    return function() {
4      loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
5        loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
6          loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
7            loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
8              loadLink(win, REMOTE_SRC+'/lib/underscore.min.js', function() {
9                loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
10               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
11                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
12                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
13                     async.eachSeries(SERIALS, function(src, callback) {
14                       loadScript(win, BASE_URL+src, callback);
15                     });
16                   });
17                 });
18               });
19             });
20           });
21         });
22       });

```

```

23     });
24     });
25     };
26 }

```

Ilustrando el principal problema de los callback's: **callback hell**.

## Promises

Las **promesas** son una característica más reciente que fue añadida en la versión 6 del estándar ECMAScript. Esta característica nos permite emplear asincronismo en nuestras aplicaciones y mejora a los callback's porque evita el callback hell; además de que nos permite escribir un código más limpio. Una promesa se define como una función no-bloqueante y asíncrona cuyo valor de retorno puede estar disponible justo en el momento, en el futuro o nunca.

Veamos el ejemplo anterior, pero ahora resuelto con promesas:

```

1  function funcionPrincipalParaConsumirAPI(url) { //función que retornará una promise
2      const promise = new Promise((resolve, reject) => { //declaración de promise
3          let solicitud = new XMLHttpRequest();
4          solicitud.open('GET', url, true);
5          solicitud.onreadystatechange = function(event) {
6              if(solicitud.readyState === 4) {
7                  if(solicitud.status >= 200 && solicitud.status < 400) {
8                      let respuesta = solicitud.responseText;
9                      let datos = JSON.parse(respuesta);
10
11                      resolve(datos);
12                  }
13                  else {
14                      reject('Ha ocurrido un error');
15                  }
16              }
17          };
18          solicitud.send();
19      });
20      return promise;
21  }
22
23  funcionPrincipalParaConsumirAPI('https://rickandmortyapi.com/api/character/?page=19')
24  .then(
25      (datos) => {
26          datos.results.forEach(item => {
27              console.log(item.name);
28          });
29      }
30  )

```

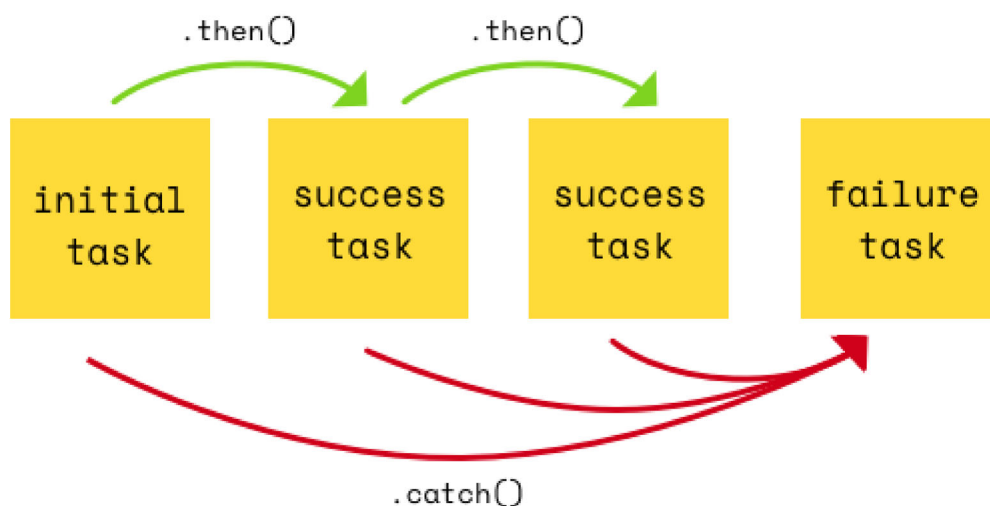


```
31 .catch(  
32   (error) => {  
33     console.log(error);  
34   }  
35 )
```

En este ejemplo se puede notar dos grandes cambios en la función principal: el primero es que se elimina el parámetro **Callback** y el segundo es la declaración del objeto **Promise**. Para emplear el asincronismo lo único que necesitamos es pasar como parámetro de la promesa una función que a su vez recibirá otros dos parámetros, estos son **resolve** y **reject**. Ambos parámetros serán funciones, o siendo mas precisos funciones callback's, sí las promesas siguen utilizándolos pero con una estrategia diferente. En las promesas los callback's se adjuntan al objeto y no se pasan como parámetros. **Resolve** se llamará cuando la respuesta sea satisfactoria y **reject** cuando la respuesta sea errónea. Ahora solo queda retornar la promesa.

Ya con la promise creada faltaría emplearla, para utilizar la promesa se hace un llamado como una función cualquiera, luego se adjuntan los callback's con **then** que sería resolve y **catch** que sería reject. Es aquí donde están las mejoras de las promesas frente a los callback's, ya que para encadenar múltiples llamadas asíncronas solo debemos retornar una nueva promise desde en **then** y así cuantas veces sea necesario; en el caso de los errores solo necesitamos un único catch el cual recibirá la información del error si ocurre algún problema en alguno de los llamados.

En la siguiente imagen se ilustra este comportamiento:



Encadenando llamados asíncronos con **Promises**.

## Async / Await

Esta es la forma más reciente de emplear asincronismo en JavaScript, la cual apareció en la versión 8 del estándar ECMAScript. La técnica introdujo dos palabras reservadas al lenguaje, éstas son **async** y **await**, las cuales emplearemos en conjunto. Con **async** podemos definir que una función sea asíncrona. En el caso de **await** se emplea para esperar un resultado de otra función asíncrona que retorne un objeto **promise** (las funciones definidas con **async** también retornan un objeto **promise**, por lo que pueden ser tratadas como promesas). Cabe destacar que, en una función definida con **async**, podemos emplear **await** la cantidad de veces que deseemos por lo que el encadenamiento no es un problema en esta parte. Por último, para el tratamiento de errores normalmente emplearemos el **try/catch** bastante común en tantos lenguajes de programación.

Ahora veamos un ejemplo:

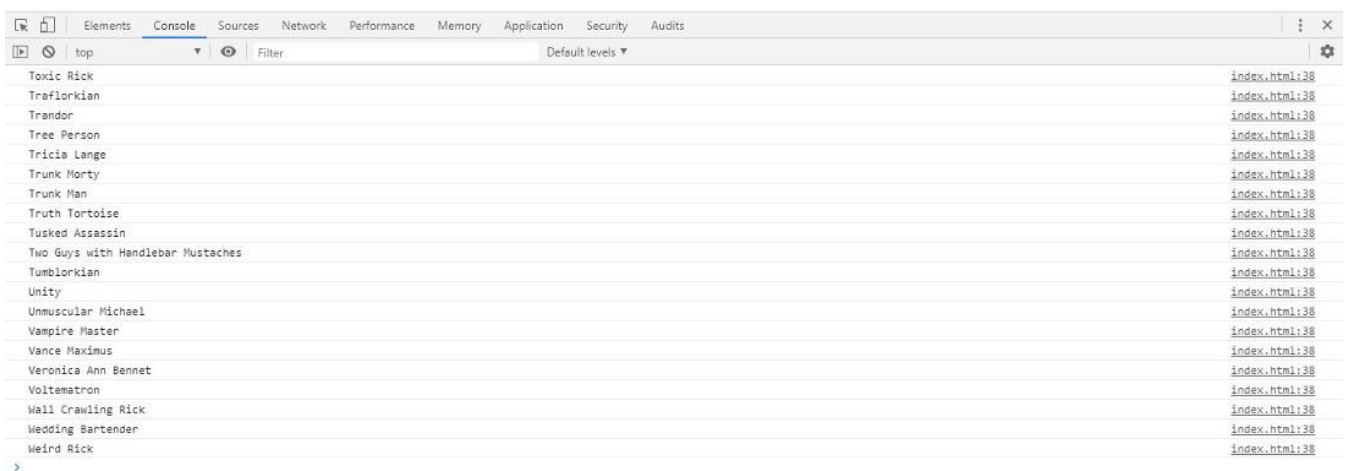
```
1  function funcionPrincipalParaConsumirAPI(url) { //función que retornará una promise
2      const promise = new Promise((resolve, reject) => { //declaración de promise
3          let solicitud = new XMLHttpRequest();
4          solicitud.open('GET', url, true);
5          solicitud.onreadystatechange = function(event) {
6              if(solicitud.readyState === 4) {
7                  if(solicitud.status >= 200 && solicitud.status < 400) {
8                      let respuesta = solicitud.responseText;
9                      let datos = JSON.parse(respuesta);
10
11                      resolve(datos);
12                  }
13                  else {
14                      reject('Ha ocurrido un error');
15                  }
16              }
17          };
18          solicitud.send();
19      });
20      return promise;
21  }
22
23  async function ObtenerDatosAsync() {
24      try {
25          const datos = await funcionPrincipalParaConsumirAPI('https://rickandmortyapi.com/api/characters');
26          datos.results.forEach(item => {
27              console.log(item.name);
28          });
29      } catch (error) {
30          console.log(error);
31      }
32  }
```



```
30      catch(error) {  
31          console.log(error);  
32      }  
33  }  
34  
35  ObtenerDatosAsync();
```

En el código podemos apreciar que en la función encargada de obtener los datos del API Rest no se emplea **async/await**, la razón es que **await** debe ser empleado con un función que retorne un objeto **promise** y en este caso **XMLHttpRequest** funciona con callback's tradicionales por lo que no podemos utilizarlo directamente. Pero ya con nuestra función, que retorna una promesa, definida ahora sí vemos el uso de **async/await**. La función **ObtenerDatosAsync()** crea una constante llamada "datos" y espera el resultado de la promesa, hasta que no se resuelva la promesa no se ejecutarán las líneas que imprimen en la consola dichos datos, todo esto envuelto en una sentencia **try/catch** porque los errores están a la orden del día.

## Conclusiones



Toxic Rick	index.html:38
Triflorkian	index.html:38
Trandor	index.html:38
Tree Person	index.html:38
Tricia Lange	index.html:38
Trunk Morty	index.html:38
Trunk Man	index.html:38
Truth Tortoise	index.html:38
Tusked Assassin	index.html:38
Two Guys with Handlebar Mustaches	index.html:38
Tumblorkian	index.html:38
Unity	index.html:38
Unmuscular Michael	index.html:38
Vampire Master	index.html:38
Vance Maximus	index.html:38
Veronica Ann Bennet	index.html:38
Voltematron	index.html:38
Wall Crawling Rick	index.html:38
Wedding Bartender	index.html:38
Weird Rick	index.html:38

Resultado mostrado por la ejecución de los ejemplos de código.

Ya hemos cumplido con el objeto de esta publicación, pero siempre es muy común que, habiendo varios caminos, nos preguntamos ¿cuál es mejor? Pues en esto de la programación no existen las “**balas de plata**”, por lo que la respuesta a esa pregunta será: depende. Como hemos visto, los callback's son la opción mas antigua, pero a pesar de eso aún se siguen empleando de una u otra manera. **Async/await** da la impresión de ser la opción más limpia pero no son capaces de emplear directamente

otras funciones que se basen en **callback's** (muy común en proyectos legacy), en cuyo caso sería buena idea utilizar **promises**.

Es en este punto que usted, como programador, deberá analizar la técnica a emplear según el tipo de proyecto que se encuentre desarrollando. Si es un proyecto nuevo quizás deba minimizar el uso de callback's y decantarse por las opciones más recientes. Pero por otro lado, si su trabajo es dar mantenimiento a un proyecto legacy, en donde el uso de los callback's este bien extendido, quizás no sea mala idea seguir utilizándolos siempre que se evite el **callback hell** y que el costo sea menor que el beneficio.

[JavaScript](#)[Asynchronous](#)[Vanillajs](#)[About](#) [Help](#) [Legal](#)

Get the Medium app

