**[#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/" \l "async-code-in-a-nutshell)Async Code in a Nutshell**

Entonces, ¿de qué se tratan todas estas cosas? Manejo de código asincrónico! Pero, ¿qué es el código asíncrono (async) para empezar? Considera este ejemplo.

**const getUser = () => {**

**setTimeout(() => {**

**return { name: 'Max' }**

**}, 2000)**

**}**

**const user = getUser() // This doesn't actually fetch the user**

**console.log(user.name) // This won't work**

¡Este código no funcionará! Tenemos una función (flecha ES6) que ejecuta setTimeout () y luego intenta devolver un objeto después de 2 segundos. Eso no funciona porque la función getUser () se ejecuta y su resultado se usa de inmediato. Sin embargo, el valor de retorno inmediato de la función es nulo. Devolver algo después de 2 segundos no funcionará. Lo mismo sería el caso de las funciones en las que se conecta con un servidor web, accede al sistema de archivos o hace algo más que no se ha terminado de inmediato.

La razón por la que no funciona es que el código JavaScript se ejecuta de forma no bloqueante (non-blocking). Eso significa que no esperará a que el código asíncrono se ejecute y devuelva un valor, solo ejecutará línea por línea hasta que termine. Por supuesto, aún podemos trabajar con operaciones asincrónicas, simplemente no funciona como se intentó en el fragmento de código anterior.

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#callbacks-to-the-rescue)Callbacks to the Rescue!

Como el código anterior no funciona así, necesitamos cambiar algo. Las devoluciones de llamada (Callbacks) son la forma más antigua de trabajar con datos una vez que están allí y continuar la ejecución del otro código mientras se ejecuta el código asíncrono.

Aquí está el código revisado.

**const getUser = cb => {**

**setTimeout(() => {**

**cb({ name: 'Max' })**

**}, 2000)**

**}**

**getUser(user => {**

**console.log(user.name) // Prints 'Max' after 2 seconds**

**})**

¿Qué cambió? Ahora no devolvemos nada en getUser (), ¡no habría funcionado de todos modos!

En cambio, ahora pasamos una función callback a getUser (). Dentro de getUser (), esta función callback se recibe como un argumento de función normal y se ejecuta una vez que se completa el temporizador. Lo bueno es que ahora también podemos pasar un argumento a la función callback.

**const getUser = cb => {**

**setTimeout(() => {**

**cb({ name: 'Max' }) // <- Passing an argument to the callback function**

**}, 2000)**

**}**

¿Por qué es útil esto? ¡Nos permite pasar datos al código que se define en otro lugar pero que solo se ejecutará una vez que se complete nuestro temporizador! En el ejemplo anterior, es una función anónima pero, por supuesto, también podría usar una función con nombre.

**// Use an anonymous function**

**getUser(user => {**

**console.log(user.name) // Prints 'Max' after 2 seconds**

**})**

**// Alternatively, use a named function:**

**getUser(handleUser) // <- This also works, just make sure to NOT execute the function here (handleUser() would be wrong!)**

**const handleUser = user => {**

**console.log(user.name) // Prints 'Max' after 2 seconds**

**}**

Tenga en cuenta que la ejecución del código no se detiene mientras espera el temporizador.

**const getUser = cb => {**

**setTimeout(() => {**

**cb({ name: 'Max' })**

**}, 2000)**

**}**

**getUser(user => {**

**console.log(user.name) // Prints 'Max' after 2 seconds**

**})**

**console.log('This prints before "Max" gets printed!') // <- This does indeed print before 'Max'**

Entonces ... las callbacks son bastante impresionantes, ¿verdad? Podemos posponer la ejecución del código y trabajar con los datos una vez que los tengamos. Y ni siquiera bloqueamos la ejecución de nuestro programa principal.

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#welcome-to-callback-hell)Welcome to Callback Hell

Lamentablemente, no es tan fácil. La idea detrás de las callbacks no es mala y hay una razón por la que todavía se está utilizando en toda la web. Pero cuando se trabaja con muchas operaciones asincrónicas dependientes, rápidamente termina en un infierno de callbacks. Considera este ejemplo.

**const checkAuth = cb => {**

**// In reality, you of course don't have a timer but will probably reach out to a server**

**setTimeout(() => {**

**cb({ isAuth: true })**

**}, 2000)**

**}**

**const getUser = (authInfo, cb) => {**

**if (!authInfo.isAuth) {**

**cb(null)**

**return**

**}**

**setTimeout(() => {**

**cb({ name: 'Max' })**

**}, 2000)**

**}**

**checkAuth(authInfo => {**

**getUser(authInfo, user => {**

**console.log(user.name)**

**})**

**})**

Esta parte aquí es bastante mala:

**checkAuth(authInfo => {**

**getUser(authInfo, user => {**

**console.log(user.name)**

**})**

**})**

Podría crecer rápidamente al usar múltiples callbacks y anidarlas entre sí no hará que su código sea más fácil de leer, comprender o mantener.

Este problema se denomina infierno de callback. Realmente puede ser un infierno porque tendrás dificultades para cambiar una de sus muchas capas de callbacks. La depuración puede ser una molestia y ciertamente ya no es legible una vez que alcanzas tres o cuatro niveles de anidamiento.

Tampoco olvides el manejo de errores. No es trivial detectar errores y manejarlos correctamente cuando descansan en el infierno de callbacks. De hecho, profundizará aún más en el infierno de callbacks una vez que comience a manejar diferentes resultados de sus diversas funciones asincrónicas. Lo más probable es que desee ejecutar otro código asincrónico si encuentra un error. ¿Te imaginas dónde terminarás con esto?

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#promising-a-better-future)Promising a Better Future

Las callbacks están bien para operaciones asincrónicas individuales, pero ciertamente no son perfectas, lo sabemos ahora. Afortunadamente, no somos los únicos que descubrimos este problema. ES6 presenta una solución: ¡Promesas! De hecho, con la ayuda de varias bibliotecas de terceros, ya podría y puede usar el concepto de promesas en un código ES5. Así es como funciona una promesa en JavaScript.

**const getUser = () => {**

**return new Promise(resolve => {**

**setTimeout(() => {**

**resolve({ name: 'Max' })**

**}, 2000)**

**})**

**}**

**getUser().then(user => {**

**console.log(user.name)**

**})**

La función getUser () puede parecer confusa, sin embargo, enfóquese en la parte inferior del código. Simplemente llamamos a getUser () allí y luego usamos then () para manejar el valor asincrónico. Dentro de getUser (), creamos una nueva promesa. La función constructora de ese objeto recibe una función como argumento. Esa función se ejecuta automáticamente y también puede recibir dos argumentos: resolver y rechazar (resolve and reject). Ambas son funciones que puede ejecutar dentro de la función pasada al constructor de la promesa.

Al llamar a resolve(data), resolve(= completa) la promesa y devuelve datos a la función ejecutada en el bloque then.

También puede llamar a reject (err) para arrojar un error. Volveré al manejo de errores más tarde.

Puede que no parezca mucho mejor de inmediato que las callbacks, pero tenga en cuenta que aquí solo usamos una operación asincrónica. El verdadero poder de las promesas se puede ver una vez que comenzamos a usar operaciones asíncronas dependientes.

**checkAuth()**

**.then(authStatus => {**

**return getUser(authStatus) // returns a new promise which may use the authStatus we fetched**

**})**

**.then(user => {**

**console.log(user.name) // prints the user name**

**})**

¡Este código es mucho más legible que su alternativa de callback! Y, por supuesto, este argumento se vuelve aún más fuerte una vez que comenzamos a encadenar más operaciones asíncronas. Dentro de then(), simplemente puede devolver el resultado de una llamada a la función. Y el resultado de esa llamada de función se puede usar en el siguiente bloque then() - lo recibirá como argumento allí. Si el resultado devuelto es una promesa, JavaScript esperará su finalización y lo resolverá por usted. ¡Increíble!

## What about errors?

¡Ud. también puede manejar los errores con facilidad! Simplemente agregue un bloque catch() a su cadena y detectará cualquier error arrojado por cualquiera de sus promesas.

**checkAuth()**

**.then(authStatus => {**

**return getUser(authStatus) // returns a new promise which may use the authStatus we fetched**

**})**

**.then(user => {**

**console.log(user.name) // prints the user name**

**})**

**.catch(error => {**

**// handle error here**

**})**

Las promesas ofrecen una mejora real sobre las callbacks y te dan la oportunidad de escapar del infierno, lo que no suena tan mal. ES6 también ofrece algunas otras características agradables que puede usar con promesas; puede echar un vistazo a Promise.all () o Promise.race (), por ejemplo.

## ¿Somos felices?

Las promesas son increíbles, ¿somos felices entonces? Tal vez. Las promesas son bastante buenas y se ven mucho uso en estos días. Realmente puede construir tuberías (pipelines) predecibles y manejables para operaciones asincrónicas con ellos. Sin embargo, tienen una limitación: solo puede manejar una operación asincrónica con cada promesa. Eso está bien para enviar solicitudes HTTP y reaccionar a las respuestas, por ejemplo. Realmente no es una gran solución si desea manejar operaciones asincrónicas que no terminan después de un "valor".

¿Cuál sería un ejemplo para una operación asincrónica que podría ejecutarse varias veces? Piensa en los eventos de los usuarios. Ciertamente son asíncronos ya que no puede bloquear su código para esperar a que ocurran. Sin embargo, realmente no puedes manejarlos con promesas. Puede manejar un evento (por ejemplo, hacer clic en un botón) pero luego su promesa se resuelve y no puede reaccionar a más clics.

**const button = document.querySelector('button')**

**const handleClick = () => {**

**return new Promise(resolve => {**

**button.addEventListener('click', () => {**

**resolve(event)**

**})**

**})**

**}**

**handleClick().then(event => {**

**console.log(event.target)**

**})**

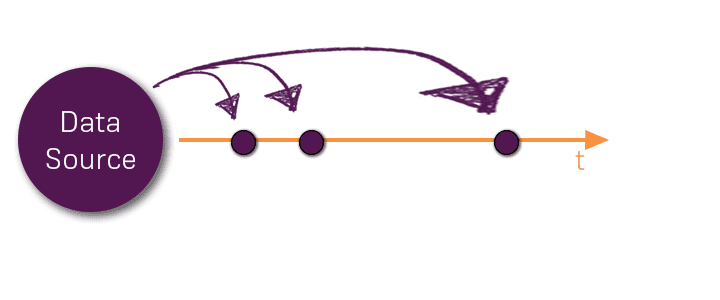
En este ejemplo, estamos agregando un detector de eventos (listener) a un botón en nuestro DOM. Esto sucede dentro de la promesa, pero dado que una promesa solo puede resolverse una vez, solo estamos reaccionando al primer clic. Los clics posteriores irán al vacío. Es esta razón, así como otra ventaja importante, lo que hizo que Observables sea muy popular. ¿Y ahora qué es eso?

[**#**](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#rxjs-observables)**RxJS Observables**

¡Las promesas no son malas, en absoluto! Usted puede muy bien apegarse a ellas y en realidad, volveré a las promesas más adelante en este artículo. Pero aunque es relativamente nuevo en el mundo de JavaScript, RxJS Observables ya ganó bastante terreno. Hay buenas razones para eso. Aquí están los dos argumentos más importantes para usar observables sobre promesas:

* Estás trabajando con flujos de datos en lugar de valores únicos
* Tienes un increíble conjunto de herramientas de operadores a tu disposición para manipular, transformar y trabajar con tus datos asíncronos

Flujos de datos

Comencemos con esa cosa de "flujos de datos". ¿Qué quiero decir exactamente con esa expresión?

La fuente de datos (por ejemplo, al hacer clic en un botón) puede emitir múltiples valores. También podría emitir solo uno, ¡no lo sabemos! O tal vez incluso sabemos que solo recibiremos un objeto de datos (por ejemplo, solicitud HTTP + respuesta); esto aún puede tratarse como una secuencia de eventos y, por lo tanto, podemos usar RxJS Observables. Más adelante habrá más información sobre argumentos que hablan de observables en todo tipo de situaciones.

Por ahora, centrémonos en ese flujo de datos. Como se mencionó anteriormente, no puede manejar esa transmisión de esa manera con promesas. Sin embargo, puedes hacerlo con observables. Así es como podría verse.

**const button = document.querySelector('button')**

**const observable = Rx.Observable.fromEvent(button, 'click')**

En realidad, esto todavía no hace nada (es decir, no reaccionamos a los clics), pero configuramos un observable aquí. Como su nombre lo indica, un observable observa algo. En este caso, observa los eventos de clic en el botón que pasamos a fromEvent (). Se podría decir que los clics en ese botón son nuestra fuente de datos ahora.

Eso es, por supuesto, agradable, pero no tan útil. Queremos reaccionar a estos clics, ¿verdad? No te preocupes, podemos! Tenemos un observable que ahora está viendo clics en ese botón. Con eso, ahora podemos configurar una suscripción en ese observable.

**const button = document.querySelector('button')**

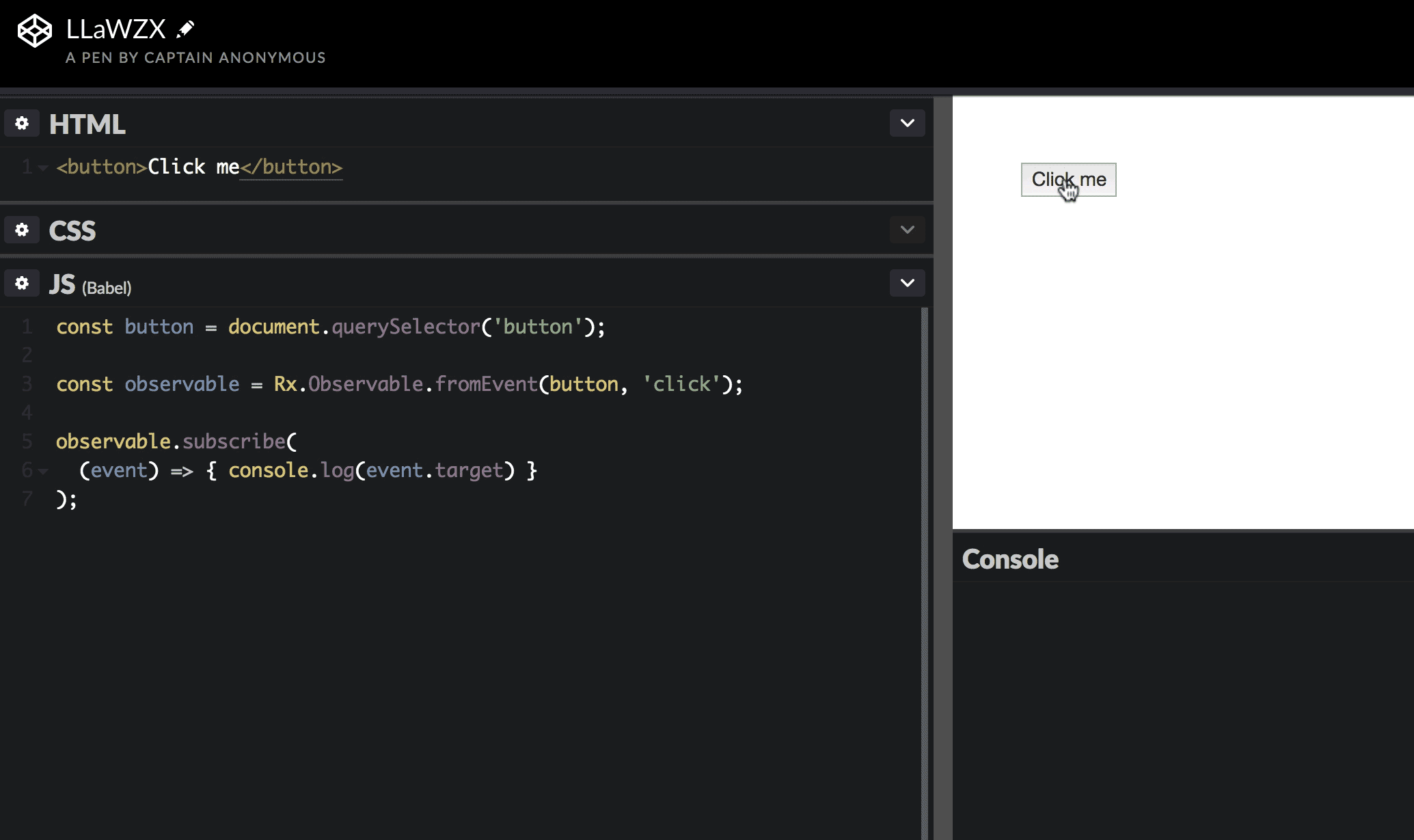
**const observable = Rx.Observable.fromEvent(button, 'click')**

**observable.subscribe(event => {**

**console.log(event.target)**

**})**

Con subscribe(), en realidad nos suscribimos a todas las piezas de datos que el observable reconoce. ¿Recuerda? Tenemos un flujo de piezas de datos. Cada vez que aparece una nueva pieza, nuestra suscripción se informa. Luego reaccionamos pasando una función como el primer argumento para subscripcion(). Podemos pasar otras dos funciones, pero volveré sobre ellas. En la primera función que pasamos, recibimos los datos de cada emisión de datos que reconoce el observable. En otras palabras: la función que pasamos a subscribe() se ejecuta cada vez que se emite un nuevo valor. En nuestro caso, cada vez que se hace clic en el botón.



Poder reaccionar a una cantidad infinita de flujo de datos asíncrono (¡o también sincrónico!) es bastante impresionante. Como puede ver, también tiene una sintaxis muy limpia y sencilla. También puede escuchar / reaccionar a otras dos cosas: errores y finalización.

Errores

Si su fuente de datos observable emite un error, probablemente quiera reaccionar a eso. Un ejemplo sería, por supuesto, una solicitud HTTP con errores. En las promesas tenías catch(), con los observables puede simplemente pasar otra función a subscribe().

**observable.subscribe(**

**event => {**

**console.log(event.target)**

**},**

**error => {**

**console.log(error)**

**} // <- Handle the error here**

**)**

Por supuesto, esto no tiene mucho sentido para un observable observando clics de un botón. Esto no puede arrojar un error. Pero muchas fuentes de datos (por ejemplo, solicitudes HTTP, validación, autenticación) pueden arrojar errores. En tales casos, la segunda función pasada a subscribe() se activa – puedes implementar cualquier lógica que desees para manejar errores dentro de esa función.

## Finalización de un Observable

Algunos, pero no todos, los observables también se completan eventualmente. Nuestro observador de clics no lo hace. Como lo haría? No sabe si el usuario volverá a hacer clic o no. Sin embargo, otros observables se completan. La buena solicitud HTTP anterior lo hace, por ejemplo. En tal caso, la primera función pasada a subscribe() todavía se activa como se describió anteriormente. Puede manejar el valor emitido allí. Pero además, se ejecuta una tercera función, si la proporcionó.

**observable.subscribe(**

**event => {**

**console.log(event.target)**

**},**

**error => {**

**console.log(error)**

**},**

**() => {**

**console.log('Completed!')**

**} // <- Gets executed once the observable completes - it doesn't receive any argument, no data**

**)**

Este tercer argumento no recibe datos, simplemente es un lugar donde puede ejecutar cualquier código que desee ejecutar una vez que sepa que el observable terminó.

Estos son los conceptos básicos sobre los observables y probablemente ya pueda ver por qué podrían ser una alternativa útil a las promesas y callbacks. Pero, ¿qué haces si el observable devuelve un nuevo observable? Al usar promesas, puede simplemente encadenar llamadas () para manejar las promesas resolviendo nuevas promesas. ¿Cómo funciona eso para los observables? ¿Y qué hay de la segunda gran ventaja que describí anteriormente: los muchos operadores?

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#observables-operators)Observables & Operators

Como se mencionó anteriormente, otra gran ventaja de los observables son los muchos operadores que puede usar en ellos. Hasta ahora, no hemos usado ninguno. subscribe() no es realmente un operador especial, sino una herramienta vital para aprovechar los observables.

Volvamos al tema de los observables que devuelven nuevos observables. Echa un vistazo al siguiente código.

**const button = document.querySelector('button');**

**const observable = Rx.Observable.fromEvent(button, 'click');**

**observable.subscribe(**

**(event) => {**

**const secondObservable = Rx.Observable.timer(1000);**

**secondObservable.subscribe(**

**(data) => console.log(data); // <- Back in callback hell?**

**);**

**}**

**);**

Aquí, creamos un nuevo observable con cada clic. Este observable emitirá un (y solo un) valor (0) después de 1 segundo. Funciona bien, pero para mí, se parece mucho al callback hell. ¿Son los observables una forma más elegante de llegar allí? ¡De ningún modo! Solo tenemos que usar uno de los increíbles operadores incluidos en la biblioteca RxJS: switchMap ().

**const button = document.querySelector('button')**

**const observable = Rx.Observable.fromEvent(button, 'click')**

**observable**

**.switchMap(event => Rx.Observable.timer(1000)) // <- use the data of the first observable in the second one (if you want) and return the new observable**

**.subscribe(data => console.log(data))**

¿No es ese un hermoso código? Mapeamos fácilmente el valor de nuestro primer observable (externo) en uno nuevo (interno). Podríamos usar los datos (en nuestro caso, el objeto evento) allí si quisiéramos. En el ejemplo, lo ignoramos y simplemente devolvemos un nuevo observable que se dispara después de 1 segundo y devuelve 0.

Hay muchos más operadores que simplemente switchMap () y son estos operadores los que dan a los observables una ventaja clara sobre las promesas, incluso en los casos en los que realmente no trabajas con un flujo de datos (como la famosa solicitud HTTP). Simplemente puede tratar todo, incluso los datos síncronos, como un flujo y usar los increíbles operadores. Considera este ejemplo.

**const observable = Rx.Observable.of({ name: 'Max' })**

**observable.pluck('name').subscribe(data => console.log(data))**

Este ejemplo no tiene nada de asíncrono: simplemente usamos observables RxJS aquí para recuperar fácilmente un valor de un objeto. Por supuesto, hay formas más fáciles de lograr el mismo resultado, pero tenga en cuenta que es un ejemplo trivial. La conclusión clave es: los observables son impresionantes debido a su naturaleza de flujo de datos y observables. Para datos (async y sync) asíncronos y sincronos.

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/#async-await)Async & Await

Encontramos al claro ganador, ¿no? Los observables poseen todo. Bueno, los observables son increíbles y solo puedo recomendar usarlos. Pero si está lidiando con un problema en el que los operadores no lo ayudan (a pesar de que es difícil de imaginar), es posible que no enfrente una desventaja al usar promesas.

Hay una cosa que puede ser molesta cuando se usan promesas u observables: estás escribiendo el código "en bloque", no como lo haces normalmente. ¿No sería bueno si pudieras escribir código asíncrono como si escribieras uno síncrono? ¿Como eso?

**auth = checkAuth() // <- async operation**

**user = getUser(auth) // <- async operation**

**console.log(user.name)**

Eso sería bueno, pero por supuesto no es posible debido a la forma en que funciona JavaScript. No espera a que finalicen las operaciones asincrónicas, por lo tanto, no puede mezclarlas con su código síncrono de esa manera. Pero con una nueva característica agregada por ES8, ¡de repente puedes!

**async function fetchUser() {**

**const auth = await checkAuth() // <- async operation**

**const user = await getUser(auth) // <- async operation**

**return user**

**}**

**fetchUser().then(user => console.log(user.name))**

En el ejemplo, puede ver que todavía usamos then() para reaccionar a los datos aprobados por nuestra promesa. Pero antes de hacer eso, en realidad usamos dos promesas (tanto checkAuth() como getUser() ¡devuelven una promesa!) para buscar un usuario. Aunque trabajamos con promesas aquí, escribimos código al igual que escribimos código síncrono "normal". ¿Cómo funciona?

La magia ocurre a través de dos palabras claves: async y await. Pones async delante de una función para convertirlo en una función asíncrona. Dicha función al final siempre se resolverá como una promesa – aunque no se parezca a una. Pero podrías reescribir el ejemplo de esta manera.

**function fetchUser() {**

**return checkAuth()**

**.then(auth => {**

**return getUser(auth)**

**})**

**.then(user => {**

**return user**

**})**

**}**

**fetchUser().then(user => console.log(user.name))**

Esto también le da una pista sobre lo que hace await: simplemente espera (increíble, ¿verdad?) una promesa para resolver. Básicamente hace lo mismo que then() hace. Espera a que se resuelva la promesa y luego toma el valor resuelta por la promesa y la almacena en una variable. El código después de await checkAuth() solo se ejecutará una vez que se haya terminado; por lo tanto, es como encadenar llamadas then().

¿Qué pasa con el manejo de errores? Eso también es bastante fácil. Como ahora estamos escribiendo código "sincrónico" de nuevo (no lo estamos), simplemente podemos usar try-catch otra vez.

**async function fetchUser() {**

**try {**

**const auth = await checkAuth() // <- async operation**

**const user = await getUser(auth) // <- async operation**

**return user**

**} catch (error) {**

**return { name: 'Default' }**

**}**

**}**

**fetchUser().then(user => console.log(user.name))**

Async / await le brinda una herramienta poderosa para trabajar con promesas. Sin embargo, no funciona con observables. La razón es que detrás de escena, async / wait realmente solo usa promesas.

# [#](https://academind.com/learn/javascript/callbacks-vs-promises-vs-rxjs-vs-async-awaits/" \l "which-approach-should-you-use)Which Approach Should You Use?

Echamos un vistazo a cuatro enfoques diferentes:

* Callbacks con el peligro de entrar en el infierno de callbacks
* Promesas para escapar del infierno de callback
* Observables para manejar flujos de datos y aplicar la magia de los operadores
* async / await para escribir código "sincrónico" con promesas

## Which approach should you use?

Use Callbacks si no tiene otra opción o solo maneja una operación asincrónica. El código seguirá siendo perfectamente manejable y comprensible. Las funciones Callback no son malas per se: solo existen mejores alternativas en muchos casos.

Uno de estos casos son varias operaciones asincrónicas encadenadas (o dependientes). Entras rápidamente en el infierno de callback cuando intentas usar callbacks en tal situación. Las promesas son una gran herramienta para manejar sus operaciones de una manera estructurada y predecible.

En todos los casos en los que usa promesas, también podría usar observables. No es estrictamente mejor, pero si hay un operador que te hace la vida más fácil o simplemente te encantan los observables, no hay inconveniente en usarlos. Hay un fuerte argumento para utilizar los observables cuando manejas flujos de datos, es decir, operaciones en las que no solo obtendrá un único valor. Las promesas no pueden manejar tales situaciones (fácilmente).

Finalmente, async / await es una herramienta increíble para casos en los que realmente no desea o necesita usar observables pero aún quiere usar promesas. Puede escribir código "sincrónico" con async / wait y manejar sus cadenas de promesa aún más fácilmente.

Finalmente, por supuesto, también se reduce a su gusto y al entorno en el que está trabajando. Para todo menos para las callback, probablemente necesitará un transpiler y / o polyfill. Si solo se dirige a entornos donde ES6 o incluso ES8 son compatibles de forma nativa, eso ya no es cierto. Para los observables, siempre necesitará la biblioteca RxJS; eso, por supuesto, significa que aumentará el código base que envía al final. Puede que no valga la pena si solo manejas casos triviales.

Espero que este artículo pueda arrojar algo de luz sobre estas herramientas y cuándo usarlas.