Promises – Using Promises

Using promises

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is an object representing the eventual completion or failure of an asynchronous operation. Since most people are consumers of already-created promises, this guide will explain consumption of returned promises before explaining how to create them.

Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function. Imagine a function, createAudioFileAsync(), which asynchronously generates a sound file given a configuration record and two callback functions: one called if the audio file is successfully created, and the other called if an error occurs.

Here's some code that uses createAudioFileAsync():

JS

function successCallback(result) {

console.log(`Audio file ready at URL: ${result}`);

}

function failureCallback(error) {

console.error(`Error generating audio file: ${error}`);

}

createAudioFileAsync(audioSettings, successCallback, failureCallback);

If createAudioFileAsync() were rewritten to return a promise, you would attach your callbacks to it instead:

JS

createAudioFileAsync(audioSettings).then(successCallback, failureCallback);

This convention has several advantages. We will explore each one.

[Chaining](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining)

A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. In the old days, doing several asynchronous operations in a row would lead to the classic callback pyramid of doom:

JS

doSomething(function (result) {

doSomethingElse(result, function (newResult) {

doThirdThing(newResult, function (finalResult) {

console.log(`Got the final result: ${finalResult}`);

}, failureCallback);

}, failureCallback);

}, failureCallback);

With promises, we accomplish this by creating a promise chain. The API design of promises makes this great, because callbacks are attached to the returned promise object, instead of being passed into a function.

Here's the magic: the then() function returns a **new promise**, different from the original:

JS

const promise = doSomething();

const promise2 = promise.then(successCallback, failureCallback);

This second promise (promise2) represents the completion not just of doSomething(), but also of the successCallback or failureCallback you passed in — which can be other asynchronous functions returning a promise. When that's the case, any callbacks added to promise2 get queued behind the promise returned by either successCallback or failureCallback.

With this pattern, you can create longer chains of processing, where each promise represents the completion of one asynchronous step in the chain. In addition, the arguments to then are optional, and catch(failureCallback) is short for then(null, failureCallback) — so if your error handling code is the same for all steps, you can attach it to the end of the chain:

JS

doSomething()

.then(function (result) {

return doSomethingElse(result);

})

.then(function (newResult) {

return doThirdThing(newResult);

})

.then(function (finalResult) {

console.log(`Got the final result: ${finalResult}`);

})

.catch(failureCallback);

You might see this expressed with [arrow functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) instead:

JSCopy to Clipboard

doSomething()

.then((result) => doSomethingElse(result))

.then((newResult) => doThirdThing(newResult))

.then((finalResult) => {

console.log(`Got the final result: ${finalResult}`);

})

.catch(failureCallback);

**Important:** Always return results, otherwise callbacks won't catch the result of a previous promise (with arrow functions, () => x is short for () => { return x; }). If the previous handler started a promise but did not return it, there's no way to track its settlement anymore, and the promise is said to be "floating".

Wrong way

JS

doSomething()

.then((url) => {

// I forgot to return this

fetch(url);

})

.then((result) => {

// result is undefined, because nothing is returned from

// the previous handler.

// There's no way to know the return value of the fetch()

// call anymore, or whether it succeeded at all.

});

This may be worse if you have race conditions — if the promise from the last handler is not returned, the next then handler will be called early, and any value it reads may be incomplete.

JS

const listOfIngredients = [];

doSomething()

.then((url) => {

// I forgot to return this

fetch(url)

.then((res) => res.json())

.then((data) => {

listOfIngredients.push(data);

});

})

.then(() => {

console.log(listOfIngredients);

// Always [], because the fetch request hasn't completed yet.

});

Therefore, as a rule of thumb, whenever your operation encounters a promise, return it and defer its handling to the next then handler.

Wright way

JS

const listOfIngredients = [];

doSomething()

.then((url) =>

fetch(url)

.then((res) => res.json())

.then((data) => {

listOfIngredients.push(data);

}),

)

.then(() => {

console.log(listOfIngredients);

});

// OR

doSomething()

.then((url) => fetch(url))

.then((res) => res.json())

.then((data) => {

listOfIngredients.push(data);

})

.then(() => {

console.log(listOfIngredients);

});

[Nesting](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#nesting)

In the two examples above, the first one has one promise chain nested in the return value of another then() handler, while the second one uses an entirely flat chain. Simple promise chains are best kept flat without nesting, as nesting can be a result of careless composition. See [common mistakes](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#common_mistakes).

Nesting is a control structure to limit the scope of catch statements. Specifically, a nested catch only catches failures in its scope and below, not errors higher up in the chain outside the nested scope. When used correctly, this gives greater precision in error recovery:

JS

doSomethingCritical()

.then((result) =>

doSomethingOptional(result)

.then((optionalResult) => doSomethingExtraNice(optionalResult))

.catch((e) => {}),

) // Ignore if optional stuff fails; proceed.

.then(() => moreCriticalStuff())

.catch((e) => console.error(`Critical failure: ${e.message}`));

Note that the optional steps here are nested — with the nesting caused not by the indentation, but by the placement of the outer ( and ) parentheses around the steps.

The inner error-silencing catch handler only catches failures from doSomethingOptional() and doSomethingExtraNice(), after which the code resumes with moreCriticalStuff(). Importantly, if doSomethingCritical() fails, its error is caught by the final (outer) catch only, and does not get swallowed by the inner catch handler.

[Chaining after a catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#chaining_after_a_catch)

It's possible to chain *after* a failure, i.e. a catch, which is useful to accomplish new actions even after an action failed in the chain. Read the following example:

JS

new Promise((resolve, reject) => {

console.log("Initial");

resolve();

})

.then(() => {

throw new Error("Something failed");

console.log("Do this");

})

.catch(() => {

console.error("Do that");

})

.then(() => {

console.log("Do this, no matter what happened before");

});

This will output the following text:

Initial

Do that

Do this, no matter what happened before

**Note:** The text "Do this" is not displayed because the "Something failed" error caused a rejection.

[**Common mistakes**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#common_mistakes)

Here are some common mistakes to watch out for when composing promise chains. Several of these mistakes manifest in the following example:

JSCopy to Clipboard

// Bad example! Spot 3 mistakes!

doSomething()

.then(function (result) {

// Forgot to return promise from inner chain + unnecessary nesting

doSomethingElse(result).then((newResult) => doThirdThing(newResult));

})

.then(() => doFourthThing());

// Forgot to terminate chain with a catch!

The first mistake is to not chain things together properly. This happens when we create a new promise but forget to return it. As a consequence, the chain is broken — or rather, we have two independent chains racing. This means doFourthThing() won't wait for doSomethingElse() or doThirdThing() to finish, and will run concurrently with them — which is likely unintended. Separate chains also have separate error handling, leading to uncaught errors.

The second mistake is to nest unnecessarily, enabling the first mistake. Nesting also limits the scope of inner error handlers, which—if unintended—can lead to uncaught errors. A variant of this is the [promise constructor anti-pattern](https://stackoverflow.com/questions/23803743/what-is-the-explicit-promise-construction-antipattern-and-how-do-i-avoid-it), which combines nesting with redundant use of the promise constructor to wrap code that already uses promises.

The third mistake is forgetting to terminate chains with catch. Unterminated promise chains lead to uncaught promise rejections in most browsers. See [error handling](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#error_handling) below.

A good rule of thumb is to always either return or terminate promise chains, and as soon as you get a new promise, return it immediately, to flatten things:

JS

Copy to Clipboard

doSomething()

.then(function (result) {

// If using a full function expression: return the promise

return doSomethingElse(result);

})

// If using arrow functions: omit the braces and implicitly return the result

.then((newResult) => doThirdThing(newResult))

// Even if the previous chained promise returns a result, the next one

// doesn't necessarily have to use it. You can pass a handler that doesn't

// consume any result.

.then((/\* result ignored \*/) => doFourthThing())

// Always end the promise chain with a catch handler to avoid any

// unhandled rejections!

.catch((error) => console.error(error));

Note that () => x is short for () => { return x; }.

Now we have a single deterministic chain with proper error handling.

Using [async/await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) addresses most, if not all of these problems — the tradeoff being that it may be easy to forget the [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) keyword.

[**Error handling**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#error_handling)

You might recall seeing failureCallback three times in the pyramid of doom earlier, compared to only once at the end of the promise chain:

JSCopy to Clipboard

doSomething()

.then((result) => doSomethingElse(result))

.then((newResult) => doThirdThing(newResult))

.then((finalResult) => console.log(`Got the final result: ${finalResult}`))

.catch(failureCallback);

If there's an exception, the browser will look down the chain for .catch() handlers or onRejected. This is very much modeled after how synchronous code works:

JSCopy to Clipboard

try {

const result = syncDoSomething();

const newResult = syncDoSomethingElse(result);

const finalResult = syncDoThirdThing(newResult);

console.log(`Got the final result: ${finalResult}`);

} catch (error) {

failureCallback(error);

}

This symmetry with asynchronous code culminates in the [async/await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) syntax:

JSCopy to Clipboard

async function foo() {

try {

const result = await doSomething();

const newResult = await doSomethingElse(result);

const finalResult = await doThirdThing(newResult);

console.log(`Got the final result: ${finalResult}`);

} catch (error) {

failureCallback(error);

}

}

It builds on promises — for example, doSomething() is the same function as before, so there's minimal refactoring needed to change from promises to async/await. You can read more about the async/await syntax in the [async functions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function) and [await](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await) references.

Promises solve a fundamental flaw with the callback pyramid of doom, by catching all errors, even thrown exceptions and programming errors. This is essential for functional composition of asynchronous operations.

[**Promise rejection events**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#promise_rejection_events)

If a promise rejection event is not handled by any handler, it bubbles to the top of the call stack, and the host needs to surface it. On the web, whenever a promise is rejected, one of two events is sent to the global scope (generally, this is either the [window](https://developer.mozilla.org/en-US/docs/Web/API/Window) or, if being used in a web worker, it's the [Worker](https://developer.mozilla.org/en-US/docs/Web/API/Worker) or other worker-based interface). The two events are:

[unhandledrejection](https://developer.mozilla.org/en-US/docs/Web/API/Window/unhandledrejection_event)

Sent when a promise is rejected but there is no rejection handler available.

[rejectionhandled](https://developer.mozilla.org/en-US/docs/Web/API/Window/rejectionhandled_event)

Sent when a handler is attached to a rejected promise that has already caused an unhandledrejection event.

In both cases, the event (of type [PromiseRejectionEvent](https://developer.mozilla.org/en-US/docs/Web/API/PromiseRejectionEvent)) has as members a [promise](https://developer.mozilla.org/en-US/docs/Web/API/PromiseRejectionEvent/promise) property indicating the promise that was rejected, and a [reason](https://developer.mozilla.org/en-US/docs/Web/API/PromiseRejectionEvent/reason) property that provides the reason given for the promise to be rejected.

These make it possible to offer fallback error handling for promises, as well as to help debug issues with your promise management. These handlers are global per context, so all errors will go to the same event handlers, regardless of source.

In [Node.js](https://developer.mozilla.org/en-US/docs/Glossary/Node.js), handling promise rejection is slightly different. You capture unhandled rejections by adding a handler for the Node.js unhandledRejection event (notice the difference in capitalization of the name), like this:

JSCopy to Clipboard

process.on("unhandledRejection", (reason, promise) => {

// Add code here to examine the "promise" and "reason" values

});

For Node.js, to prevent the error from being logged to the console (the default action that would otherwise occur), adding that process.on() listener is all that's necessary; there's no need for an equivalent of the browser runtime's [preventDefault()](https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault) method.

However, if you add that process.on listener but don't also have code within it to handle rejected promises, they will just be dropped on the floor and silently ignored. So ideally, you should add code within that listener to examine each rejected promise and make sure it was not caused by an actual code bug.

[**Composition**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#composition)

There are four [composition tools](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise#promise_concurrency) for running asynchronous operations concurrently: [Promise.all()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/all), [Promise.allSettled()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled), [Promise.any()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/any), and [Promise.race()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/race).

We can start operations at the same time and wait for them all to finish like this:

JSCopy to Clipboard

Promise.all([func1(), func2(), func3()]).then(([result1, result2, result3]) => {

// use result1, result2 and result3

});

If one of the promises in the array rejects, Promise.all() immediately rejects the returned promise and aborts the other operations. This may cause unexpected state or behavior. [Promise.allSettled()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/allSettled) is another composition tool that ensures all operations are complete before resolving.

These methods all run promises concurrently — a sequence of promises are started simultaneously and do not wait for each other. Sequential composition is possible using some clever JavaScript:

JSCopy to Clipboard

[func1, func2, func3]

.reduce((p, f) => p.then(f), Promise.resolve())

.then((result3) => {

/\* use result3 \*/

});

In this example, we [reduce](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce) an array of asynchronous functions down to a promise chain. The code above is equivalent to:

JSCopy to Clipboard

Promise.resolve()

.then(func1)

.then(func2)

.then(func3)

.then((result3) => {

/\* use result3 \*/

});

This can be made into a reusable compose function, which is common in functional programming:

JSCopy to Clipboard

const applyAsync = (acc, val) => acc.then(val);

const composeAsync =

(...funcs) =>

(x) =>

funcs.reduce(applyAsync, Promise.resolve(x));

The composeAsync() function accepts any number of functions as arguments and returns a new function that accepts an initial value to be passed through the composition pipeline:

JSCopy to Clipboard

const transformData = composeAsync(func1, func2, func3);

const result3 = transformData(data);

Sequential composition can also be done more succinctly with async/await:

JSCopy to Clipboard

let result;

for (const f of [func1, func2, func3]) {

result = await f(result);

}

/\* use last result (i.e. result3) \*/

However, before you compose promises sequentially, consider if it's really necessary — it's always better to run promises concurrently so that they don't unnecessarily block each other unless one promise's execution depends on another's result.

[**Creating a Promise around an old callback API**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#creating_a_promise_around_an_old_callback_api)

A [Promise](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) can be created from scratch using its [constructor](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise). This should be needed only to wrap old APIs.

In an ideal world, all asynchronous functions would already return promises. Unfortunately, some APIs still expect success and/or failure callbacks to be passed in the old way. The most obvious example is the [setTimeout()](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout) function:

JSCopy to Clipboard

setTimeout(() => saySomething("10 seconds passed"), 10 \* 1000);

Mixing old-style callbacks and promises is problematic. If saySomething() fails or contains a programming error, nothing catches it. This is intrinsic to the design of setTimeout.

Luckily we can wrap setTimeout in a promise. The best practice is to wrap the callback-accepting functions at the lowest possible level, and then never call them directly again:

JSCopy to Clipboard

const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(10 \* 1000)

.then(() => saySomething("10 seconds"))

.catch(failureCallback);

The promise constructor takes an executor function that lets us resolve or reject a promise manually. Since setTimeout() doesn't really fail, we left out reject in this case. For more information on how the executor function works, see the [Promise()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/Promise) reference.

[**Timing**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#timing)

Lastly, we will look into the more technical details, about when the registered callbacks get called.

[**Guarantees**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#guarantees)

In the callback-based API, when and how the callback gets called depends on the API implementor. For example, the callback may be called synchronously or asynchronously:

JSCopy to Clipboard

function doSomething(callback) {

if (Math.random() > 0.5) {

callback();

} else {

setTimeout(() => callback(), 1000);

}

}

The above design is strongly discouraged because it leads to the so-called "state of Zalgo". In the context of designing asynchronous APIs, this means a callback is called synchronously in some cases but asynchronously in other cases, creating ambiguity for the caller. For further background, see the article [Designing APIs for Asynchrony](https://blog.izs.me/2013/08/designing-apis-for-asynchrony/), where the term was first formally presented. This API design makes side effects hard to analyze:

JSCopy to Clipboard

let value = 1;

doSomething(() => {

value = 2;

});

console.log(value); // 1 or 2?

On the other hand, promises are a form of [inversion of control](https://en.wikipedia.org/wiki/Inversion_of_control) — the API implementor does not control when the callback gets called. Instead, the job of maintaining the callback queue and deciding when to call the callbacks is delegated to the promise implementation, and both the API user and API developer automatically gets strong semantic guarantees, including:

* Callbacks added with [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) will never be invoked before the [completion of the current run](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop#run-to-completion) of the JavaScript event loop.
* These callbacks will be invoked even if they were added *after* the success or failure of the asynchronous operation that the promise represents.
* Multiple callbacks may be added by calling [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) several times. They will be invoked one after another, in the order in which they were inserted.

To avoid surprises, functions passed to [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) will never be called synchronously, even with an already-resolved promise:

JSCopy to Clipboard

Promise.resolve().then(() => console.log(2));

console.log(1);

// Logs: 1, 2

Instead of running immediately, the passed-in function is put on a microtask queue, which means it runs later (only after the function which created it exits, and when the JavaScript execution stack is empty), just before control is returned to the event loop; i.e. pretty soon:

JSCopy to Clipboard

const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(0).then(() => console.log(4));

Promise.resolve()

.then(() => console.log(2))

.then(() => console.log(3));

console.log(1); // 1, 2, 3, 4

[**Task queues vs. microtasks**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#task_queues_vs._microtasks)

Promise callbacks are handled as a [microtask](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide) whereas [setTimeout()](https://developer.mozilla.org/en-US/docs/Web/API/setTimeout) callbacks are handled as task queues.

JSCopy to Clipboard

const promise = new Promise((resolve, reject) => {

console.log("Promise callback");

resolve();

}).then((result) => {

console.log("Promise callback (.then)");

});

setTimeout(() => {

console.log("event-loop cycle: Promise (fulfilled)", promise);

}, 0);

console.log("Promise (pending)", promise);

The code above will output:

Promise callback

Promise (pending) Promise {<pending>}

Promise callback (.then)

event-loop cycle: Promise (fulfilled) Promise {<fulfilled>}

For more details, refer to [Tasks vs. microtasks](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide/In_depth#tasks_vs_microtasks).

[**When promises and tasks collide**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises#when_promises_and_tasks_collide)

If you run into situations in which you have promises and tasks (such as events or callbacks) which are firing in unpredictable orders, it's possible you may benefit from using a microtask to check status or balance out your promises when promises are created conditionally.

If you think microtasks may help solve this problem, see the [microtask guide](https://developer.mozilla.org/en-US/docs/Web/API/HTML_DOM_API/Microtask_guide) to learn more about how to use [queueMicrotask()](https://developer.mozilla.org/en-US/docs/Web/API/queueMicrotask) to enqueue a function as a microtask.