Javascript fundamentals

ASYNCHRONOUS PROGRAMMING PATTERNS IN JAVASCRIPT

How to Use Async/Await and Promises to Solve Programming Problems

Tamás Sallai

Table of contents

Introduction

Asynchronous programming is everywhere in Javascript. This is the result of the fundamental choices that define how it works. In other languages, you can use multiple threads and that allows synchronous waiting, a crucial feature missing from Javascript. It is by design single-threaded and a wait operation stops everything, just think of the case where a long calculation freezes the UI.

Without a way to wait for a later result synchronously, Javascript needs to use callbacks. Even simple things like waiting for a given duration requires a function that will be run when the time is up:

```
setTimeout(() => {
  console.log("1 second passed!");
}, 1000);
```

You can find this pattern everywhere, as most of the things are asynchronous in nature. Using `fetch` to make an HTTP call to a server is an async operation. Just like getting information about the available cameras and microphones with the `getUserMedia` call, as it needs to get permission from the user. Same with reading and writing files. While these have synchronous versions for now, they are terrible for performance. Or want to do some web scraping with Puppeteer? Every single instruction is asynchronous as all of them communicate with a remote process. Or for backend applications, reading or writing data to a database is also inherently async.

And not only that some functions are async but all the other functions that call them need to be async too. A piece of functionality that requires making a network call, for example, is asynchronous, no matter how insignificant that call is compared to what other things the function is doing. Because of this, almost all Javascript applications consist of mostly asynchronous operations.

Over the years, the language got a lot of features that make writing async code easier. Gone are the days of the so-called callback hell where a series of callback-based async calls made the program's structure very hard to understand and easy to inadvertently silence errors.

First, Promises gained widespread support, flattening the structure of callbacks. Then async/await became the mainstream way of async programming, hiding a lot of the complexities of asynchronicity behind only two keywords: `async` and `await`.

This improved readability and made it a lot easier for new programmers to write asynchronous code. Modern Javascript is still using the same single-threaded no-sync-wait event loop, but the program structure reflects that of a modern language.

But asynchronous programming is inherently hard. While the language helps with the syntax to make understanding and writing code easier, asynchronicity introduces a lot of potential errors, most of them so subtle they only occur in special circumstances.

Even though I've been working for many years with asynchronous code, some of the problems in this book took me a week to reach a solution I'm happy with. My goal with this book is that you'll have an easier time when you encounter similar problems by knowing what are the hard parts. This way you won't need to start from zero but you'll have a good idea of what are the roadblocks and the best practices.

You'll notice that error handling is a recurring topic in this book. This is because it is an often overlooked concept and that leads to code that easily breaks. By knowing how errors in async functions and Promises work you'll write safer programs.

Structure

This book is divided into two parts.

The first chapter, Getting started with async/await, is an introduction to async/await and Promises and how each piece of the async puzzle fit together. The primary focus is async functions as they are the mainstream way to program asynchronously in Javascript. But async/await is a kind of magic without knowing about Promises, so you'll learn about them too.

By the end of the first chapter, you'll have a good understanding of how to use async functions and how they work under the hood.

The second part of the book consists of several common programming tasks and problems. You'll learn when that particular problem is important, how to solve it, and what are the edge cases. This gives you a complete picture so that when you encounter a similar problem you'll know how to approach it.

This book is not meant to be read from cover to cover but to be used as a reference guide. With the patterns described in this book, my hope is that you'll see the underlying difficulty with async programming so when you work on your own solutions you'll know the pitfalls and best practices so you'll end up with more reliable code.

Getting started with async/await

In this chapter, we start with an introduction to asynchronous programming in Javascript. The main focus is async functions, as you'll likely use them more often, but you'll also learn how Promises work.

Async functions

Normal functions return a result with the `return` keyword:

```
const fn = () => {
  return 5;
}

fn();
// 5
```

Async functions, in contrast, return a `Promise`:

```
const asyncFn = async () => {
  return 5;
}

asyncFn();
// Promise
```

You can make any function async with the `async` keyword, and there are a lot of functions that return a Promise instead of a result.

For example, to read a file from the filesystem, you can use `fs.promises`, a variant of the `fs` functions returning Promises:

```
const fs = require("fs");

fs.promises.readFile("test.js");
// Promise
```

Or convert an image to jpeg with the sharp library, which also returns a Promise:

```
const sharp = require("sharp");

sharp("image.png")
  .jpeg()
  .toFile("image.jpg");
// Promise
```

Or make a network request with `fetch`:

```
fetch("https://advancedweb.hu");
// Promise
```

How to use the Promise

An async function still has a return value, and the Promise holds this result. To get access to the value, **attach a callback** using the `then()` function. This callback will be called with the result of the function.

To get the file contents after the `readFile`:

```
const fs = require("fs");

fs.promises.readFile("test.js").then((result) => {
  console.log(result);
  // Buffer
});
```

Similarly, to get the result of our simple async function, use `then()`:

```
const asyncFn = async () => {
  return 5;
}

asyncFn().then((res) => {
  console.log(res);
  // 5
});
```

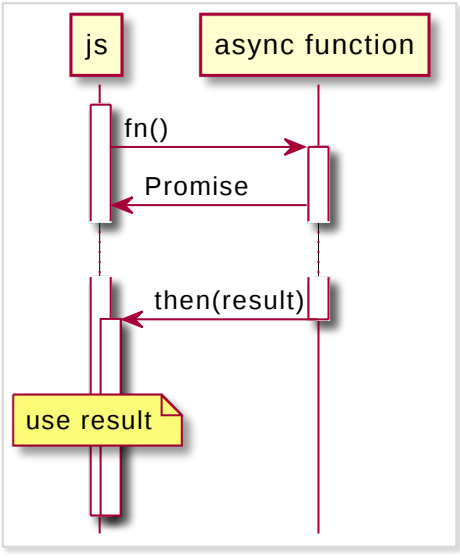
The benefits of Promises

But why complicate a function call with Promises? A normal function just returns a value that can be used on the next line, without the need for any callbacks.

The benefit of returning a Promise instead of a value is that the **result might not be ready by the time the function returns**. The callback can be called much later, but the function must return immediately. This extends what a function can do.

```
// sync
fn();
// result is ready

// async
asyncFn().then(() => {
  // result is ready
})
// result is pending
```

In many cases, a synchronous result is not possible. For example, making a network request takes forever compared to a function call. With Promises, it does not matter if something takes a lot of time or produces a result immediately. In both cases, the `then()` function will be called when the result is ready.



An async function returns a Promise

Using standardized Promises also allows other constructs to build on it. As we've seen, the `async` keyword makes a function return a Promise instead of a value.

> Recap
>
> Async functions return Promises which are values that are available sometime in the future.

The await keyword

Using Promises with callbacks requires changes to the code structure and it makes the code harder to read.

Instead of a flat structure of synchronous function calls:

```
const user = getUser();
const permissions = getPermissions();
const hasAccess = checkPermissions(user, permissions);
if (hasAccess) {
  // handle request
}
```

Promises need callbacks:

```
getUser().then((user) => {
  getPermissions().then((permissions) => {
    const hasAccess = checkPermissions(user, permissions);
    if (hasAccess) {
      // handle request
    }
  });
});
```

> Note
>
> Promises support a flat structure when they are invoked sequentially, this is their main selling point over traditional callbacks. For example, a series of async calls can process an object:
>
> ```
> getUser()
>   .then(getPermissionsForUser)
>   .then(checkPermission)
>   .then((allowed) => {
>     // handle allowed or not
>   });
> ```
>
> This is an almost flat structure which we'll detail in the Chaining Promises chapter, but a callback is still needed at the end. The `await` keyword eliminates the need for that.

To solve this without losing the advantages of Promises, async functions can use the `await` keyword. It **stops the function until the Promise is ready** and returns the result value.

```
const asyncFn = async () => {
  return 5;
}

await asyncFn();
// 5
```

The above code that uses callbacks can use `await` instead which leads to a more familiar structure:

```
const user = await getUser();
const permissions = await getPermissions();
const hasAccess = checkPermissions(user, permissions);
if (hasAccess) {
  // handle request
}
```

The `await` keyword that **waits for async results** makes the code look almost like it's synchronous, but with all the benefits of Promises.

Note that `await` stops the execution of the function, which seems like something that can not happen in Javascript. But under the hood, it still uses the `then()` callbacks and since async functions return Promises they don't need to provide a result immediately. This allows halting the function without major changes to how the language works.

Async function examples

Async functions with `await` are powerful. They make a complicated and asynchronous workflow seem easy and familiar, hiding all the complexities of results arriving later.

Browser automation is a prime example. The Puppeteer project allows starting Chromium and driving it with the DevTools protocol.

Taking a screenshot of a webpage is only a few lines:

```
const browser = await puppeteer.launch(options);
const page = await browser.newPage();
const response = await page.goto(url);
const img = await page.screenshot();
await browser.close();
```

This code hides a lot of complexity. It starts a browser, then sends commands to it, all asynchronous since the browser is a separate process. But the end result is an image buffer containing the screenshot.

> Note
>
> The above code leaves the browser running if there is an error during execution. You'll learn how to handle closing resources in the The async disposer pattern chapter.

Another example is to interface with databases. A remote service always requires network calls and that means asynchronous results.

This code, used in an AWS Lambda function, updates a user's avatar image:

```
// move object out of the pending directory
await s3.copyObject({/*...*/}).promise();
await s3.deleteObject({/*...*/}).promise();

// get the current avatar image
const oldAvatar = (await dynamodb.getItem({/*...*/}).promise())
  .Item.Avatar.S;

// update the user's avatar
await dynamodb.updateItem({/*...*/}).promise();
// delete the old image
await s3.deleteObject({/*...*/}).promise();

return {
  statusCode: 200,
};
```

It makes calls to the AWS S3 service to move objects, and to the DynamoDB database to read and modify data. Both of these are remote services, but all the complexities are hidden behind the `await`s.

> Recap
>
> The `await` keyword stops the function until the future result becomes available.

Chaining Promises

We've seen that when an async function returns a value it will be wrapped in a Promise and the `await` keyword extracts the value from it. But what happens when an async function returns a Promise? Would that mean you need to use two `await`s?

Consider this code:

```
const f1 = async () => {
  return 2;
};

const f2 = async () => {
  return f1();
}

const result = await f2();
```

Strictly following the process described in the previous chapters, `f1()` returns `Promise<2>`, and `f2()` returns `Promise<Promise<2>>`, so the value of `result` will be `Promise<2>` instead of `2`.

But this is not what happens. When an async function returns a Promise, it returns it without adding another layer. It does not matter if it returns a value or a Promise, it will always be a Promise and it will always resolve with the final value and not another Promise.

This works the same for Promise chains too. The `.then()` callback is also wrapped in a Promise if it's not one already so you can chain them easily:

```
getUser()
  .then(async function getPermissionsForUser(user) {
    const permissions = await // ...;
    return permissions;
  })
  .then(async function checkPermission(permissions) {
    const allowed = await // ...;
    return allowed;
  })
  .then((allowed) => {
    // handle allowed or not
  });
```

The `getUser` function returns a `Promise<User>`, then the `getPermissionsForUser` gets the user object (the resolved value), then returns the permission set. The next call, `checkPermission` gets the permissions, and so on.

A useful analog is how the `flatMap` function for an array works. It does not matter if it returns a value or an array, the end result will always be an array with values. It is a `map`, followed by a `flat`.

```
[1].flatMap((a) => 5) // [5]
[1].flatMap((a) => [5]) // [5]

[1].flat() // [1]
[[1]].flat() // [1]
```

When I'm not sure what a Promise chain returns, I mentally translate Promises to arrays where every async function return a flattened array with its result and an `await` is getting the first element:

```
const f1 = () => {
  return [2].flat();
};

const f2 = () => {
  return [f1()].flat();
}

const result = f2()[0]; // 2
```

A Promise chain then becomes a series of `flatMap`s:

```
const getUser = () => ["user"];

getUser()
  .flatMap(function getPermissionsForUser(user) {
    // user = "user"
    const permissions = "permissions";
    return permissions;
  })
  .flatMap(function checkPermission(permissions) {
    // permissions = "permissions"
    const allowed = true;
    return allowed;
  })
  .flatMap((allowed) => {
    // allowed = true
    // handle allowed or not
  });
```

This eliminates most of the complexities of asynchronicity and is a lot easier to reason about.

Promises

So far we've discussed how to write Promise-producing async functions and how to wait for a Promise to have a value. But how to create Promises in cases where there are no existing Promises to build on?

To know this, we need to discuss how asynchronicity works in Javascript and how to construct Promises from callbacks.

Callbacks

Async results come in the form of callbacks. These are like the parameter of the `then()` function in Promises but are called when a particular event happens. For example, the simplest callback is the `setTimeout` call that waits a given number of milliseconds then invokes the argument function:

```
setTimeout(() => {
  // 1s later
}, 1000);
```

This pattern of using callbacks is everywhere in Javascript.

The gapi, the library to use Google services, needs a callback when it loads a client library:

```
gapi.load("client:auth2", () => {
  // auth2 client loaded
});
```

Similarly, event listeners, such as a click handler, calls a function when the event happened:

```
const button = document.querySelector("#button");

button.addEventListener("click", () => {
  // button is clicked
}, {once: true})
```

The problem with callbacks is that we are back to square one as instead of having a flat structure we have nesting again.

Fortunately, there is a simple way to convert callbacks to Promises and then use them with `await`.

> ### Event listeners and Promises
>
> Eagle-eyed readers might have spotted the `{once: true}` part in the last example. Event listeners are inherently different than async functions as they represent *multiple events* instead of a single result value. Because of this, you can not replace them with Promises.
>
> The exception is when there is exactly one event, as in the above example. In a sense, it works similarly to the timeout or the gapi load function, as the callback is invoked when the event happens.

The Promise constructor

Async functions return Promises but you can only use `await` to wait for other Promises and not for callbacks. To make a Promise, you need to use its constructor:

```
new Promise((res) => {
  // call res when the Promise is ready
});
```

Calling the `res` callback signals the Promise that the result is ready. This is the same as when an async function returns.

To convert the `setTimeout` to a Promise, wrap it in the constructor and use `res` as the callback:

```
await new Promise((res) => {
  setTimeout(res, 1000);
});
// 1s later

// same as
await new Promise((res) => {
  setTimeout(() => {
    res();
  }, 1000);
});
```

Now that it is a Promise, `await` works just like for other async functions.

Similarly, the two other examples are also easy to promisify:

```
await new Promise((res) => {
  gapi.load("client:auth2", res);
});
// auth2 client loaded
```

```
const button = document.querySelector("#button");

await new Promise((res) => {
  button.addEventListener("click", res, {once: true});
});
// button is clicked
```

With Promises instead of callbacks it's easy to use `await` and make a sequence of async operations. To load the auth2 client when the button is clicked:

```
await new Promise((res) => {
  button.addEventListener("click", res, {once: true});
});
// the button is clicked
await new Promise((res) => {
  gapi.load("client:auth2", res);
});
// the gapi library is loaded
```

> `Converting between callbacks and Promises`
>
> You'll learn more about this in the Convert between Promises and callbacks chapter.

Promise states

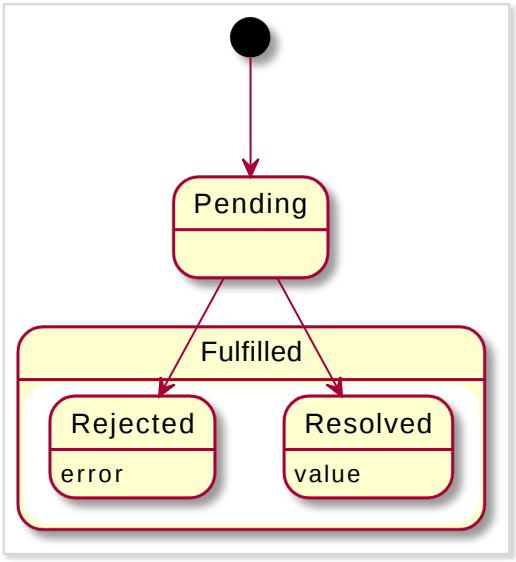
As a Promise holds a future value it can be in several states.

When you create a Promise, it starts in the **Pending** state. It is before the resolve/reject callback is called or the async function finished.

When there is a value, the Promise transitions into the **Resolved** state. It is when the `await` in unblocked and the async function continues to run.

If there is an error, the Promise goes into the **Rejected** state. It is when the `await` throws an exception.

The Resolved and the Rejected states are collectively called **Fulfilled** (or **Settled**). When a Promise is in this state it's already finished and produced a value or an error.



Promise states

Result value

The examples above all used the Promise `res` function to signal when an async operation is ready and did not produce any value. To also return a value, **pass it to the `res` call**.

```
const value = new Promise((res) => {
  res(5);
});
// 5
```

A more complicated example, the next Promise shows the Google Drive file picker and resolves with the selected file. This seems complicated, but the underlying idea is the same: create a new Promise, then call the `res` method when the result is ready.

```
const folder = await new Promise((res) => {
  const picker = new google.picker.PickerBuilder()
    // ... configuration ...
    .setCallback((data) => {
      if (/* selected action*/) {
        // a folder is selected
        const selectedFolder = ...

        // resolve the Promise
        res(selectedFolder);
      }
    })
    .build();
  picker.setVisible(true);
});
// folder is the selected folder
```

> Note
>
> You can return only one value with the `res()` call. You'll learn about ways to return multiple in the Returning multiple values chapter.

Error handling

So far we've been discussing async functions when everything is going fine. No database timeout, no validation errors, no sudden loss of network. But these things happen, so we need to prepare for them too. Without proper error propagation, failures stall Promises and async functions, stopping them forever.

With synchronous functions, errors happen when something `throw`s them, and they bubble up until there is a `try-catch`:

```
const fn = () => {
  throw new Error("Something bad happened");
}

try {
  fn();
}catch(e) {
  // handle error
}
```

Async functions work similarly so that when there is an error thrown it will go up until a `try-catch`. When there is an asynchronous error (we'll look into how they work in the next chapter), you can handle it with the same familiar structure:

```
const fn = async () => {
  throw new Error("Something bad happened");
}

try {
  await fn();
}catch(e) {
  // handle error
}
```

> Note
>
> Errors are thrown during the `await` and not when the function is called:
>
> ```
> const asyncFn = async () => {
>   throw new Error("Something bad happened");
> }
>
> const res = asyncFn(); // no error here
>
> try {
>   await res; // error
> }catch(e) {
>   // handle error
> }
> ```

Error callback in the Promise constructor

But when you use the Promise constructor, you need to pay attention to propagate the errors.

Let's revisit our previous examples! The `gapi.load` loads a client library that can be used with Google services and it needs a callback to notify when it's finished. This is easy to turn into a Promise with the Promise constructor:

```
await new Promise((res) => {
  gapi.load("client:auth2", res);
});
// auth2 loaded
```

But what happens when there is an error? Maybe the network is down and the client library can not be loaded. In this case, the callback function is never called and the Promise is never finished. An `await` waiting for it will wait forever.

```
await new Promise((res) => {
  gapi.load("client:auth2", res); // error
});
// never happens
```

To implement proper error propagation, we need two things. First, a way to signal the Promise that an error happened, and second, a way to detect that error.

Rejecting a Promise

The Promise constructor provides a second callback that can signal that an error happened:

```
new Promise((res, rej) => {
  // call res() when everything is fine
  // call rej() when there is an error
});
```

When the `rej` function is called it behaves similarly to an async function throwing an error:

```
try {
  await new Promise((res, rej) => {
    rej(); // signal error
  });
}catch(e) {
  // handle error
}
```

Similar to the `res` function, you can pass the error object which will be the rejection reason:

```
try {
  await new Promise((res, rej) => {
    rej(new Error("Something bad happened"));
  });
}catch(e) {
  console.log(e.message) // Something bad happened
}
```

What is an error

What constitutes an error is up to the implementation. For example, `fetch` throws an error when the connection can not be established, but it does not when the response code is outside the 2xx range.

```
try {
  const response = await fetch(options);
  if (response.ok) {
    // request successful
  }else {
    // error response
  }
} catch (e) {
  // error while sending the request
}
```

Exceptions thrown in the constructor also reject the Promise:

```
new Promise(() => {
  throw Error("something bad happened");
});
// rejected with error
```

This behavior makes most programming errors to propagate correctly. But unlike the `reject` callback, this only handles synchronous errors.

Detecting errors

Now that we have a way to signal an error to the Promise, let's see how to adapt the gapi example to take advantage of this construct!

In this specific case, the callback can also be an object with `callback` and `onerror` handlers:

```
await new Promise((res, rej) => {
  gapi.load("client:auth2", {callback: res, onerror: rej});
});
```

Now if there is an error during the load process the `rej` function is called and the `await` throws an error.

> **Note**
>
> How the underlying function signals results and errors can vary from library-to-library. You'll learn more about it in the Callback styles chapter.

Error propagation in Promise chains

While async functions use the same `try..catch` structure as normal functions, Promise chains work differently. The `then` callback gets an additional callback, one that is called when a previous Promise is rejected:

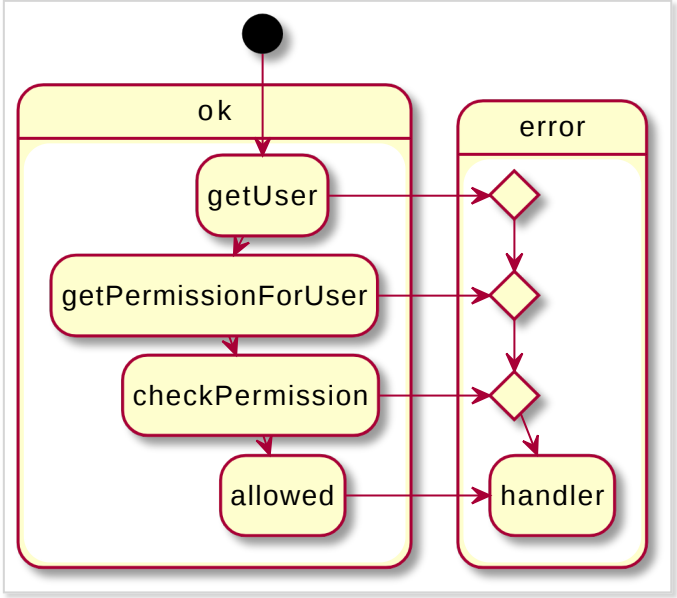
```
const rejectedPromise = new Promise((res, rej) => {
  rej(new Error("rejected"));
});

rejectedPromise.then(undefined, (error) => {
  // error = "rejected"
  // handle error
});
```

Errors in Promise chains move to the next error handler, skipping all previous steps. This makes it easy to collect errors from multiple steps and handle them in a single place:

```
getUser()
  .then(getPermissionsForUser)
  .then(checkPermission)
  .then((allowed) => {
    // handle allowed or not
  }).then(undefined, (error) => {
    // there was an error in one of the previous steps
  });
```

A useful way to think about error propagation through the chain is to think about it as 2 parallel railway tracks with stations. A station is a `then` callback, and its result determines which track the train continues.

> **Tip**
>
> Instead of `.then(undefined, (error) => {...})`, you can use the shorthand `.catch((error) => {...})`.



Error handler in a Promise chain

This way of thinking allows untangling way more complicated chains. Let's dissect what's happening here:

```
getPromise()
  .then(handler1)
  .then(handler2, catch1)
  .then(handler3)
  .then(undefined, catch2)
  .then(handler4)
```

This yields the following diagram:

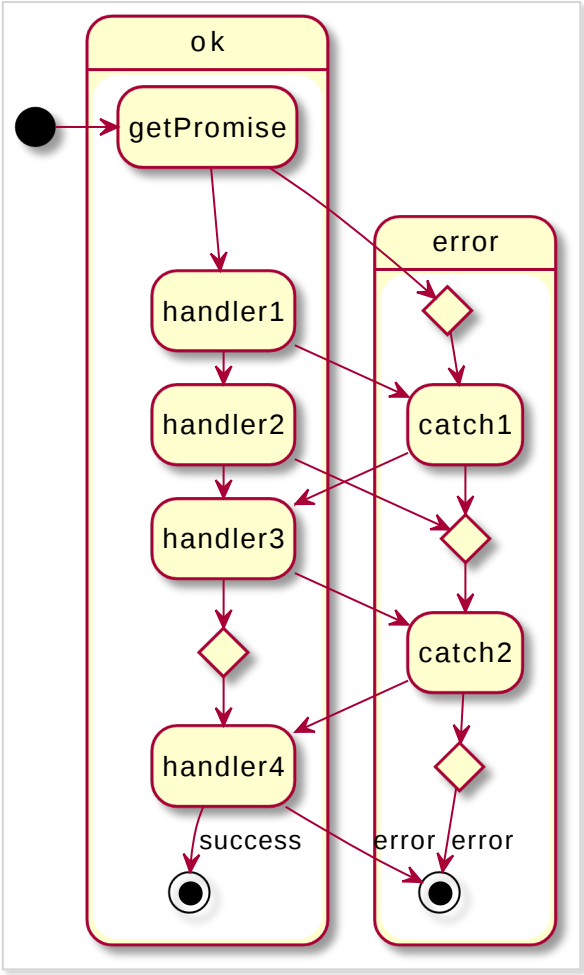


Diagram of the promise chain

It looks complicated, but there are simple rules:

- Each function, no matter which track it is, can move the execution to the OK as well as the error tracks
- When a handler is missing (the diamonds in the above diagram) the execution stays on the same track

For example, if `getPromise()` is rejected, the next function will be `catch1`. If it resolves, then `handler3` is the next one. If it resolves, `handler4` is called, and it will be the result of the whole chain, no matter if it resolves or rejects.

`Error handlers`

Notice this line:

```
.then(handler2, catch1)
```

A common source of errors is that if there is an error in `handler2` then `catch1` **won't run**. Each error handler handles errors from *previous steps*.

Parallel and sequential processing

When you use `await` in an async function, it stops the execution until the Promise is resolved. This makes each line wait for the previous, which yields a code that looks almost like a synchronous one:

```
await s3.copyObject({/*...*/}).promise();
await s3.deleteObject({/*...*/}).promise();
```

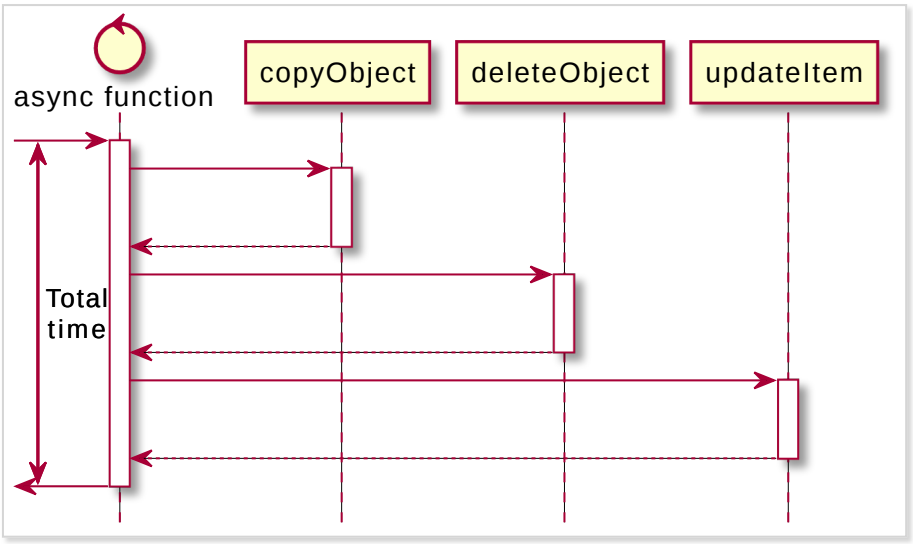
The above code copies an object, then it deletes one. In case you wanted to *move* the object, it is the desirable chain of events as you want the copy operation to finish before deleting the original.

Parallel processing with Promise.all

But what if there is an unrelated call, such as an update to a database?

```
await s3.copyObject({/*...*/}).promise();
await s3.deleteObject({/*...*/}).promise();

await dynamodb.updateItem({/*...*/}).promise();
```

While the `dynamodb.updateItem` has nothing to do with the S3 object, it still waits for the move operation (copy + delete) to finish. This makes the overall process longer than necessary.



Serial execution

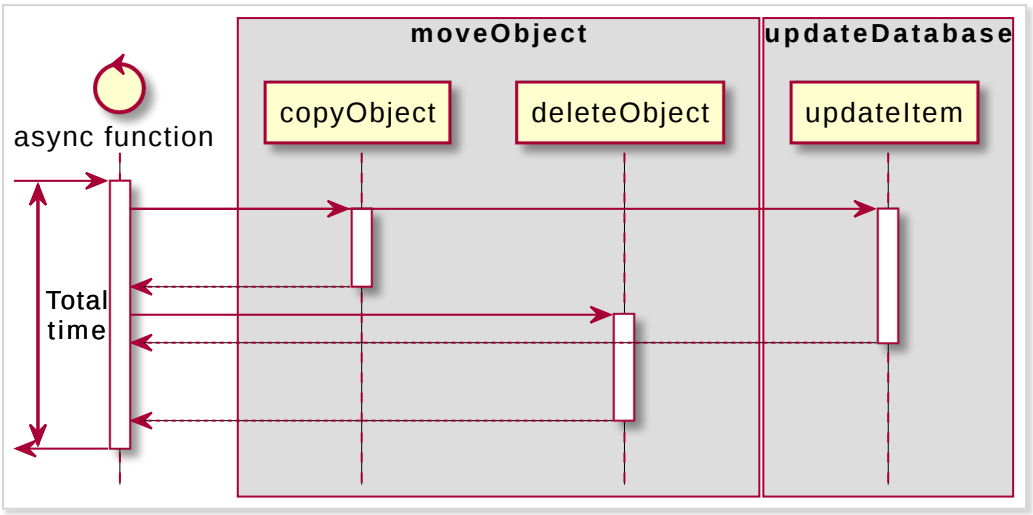
In this case, a better implementation is to run the two parts in parallel. To make it easier to see, let's group the two operations with async functions:

```
const moveObject = async () => {
  await s3.copyObject({/*...*/}).promise();
  await s3.deleteObject({/*...*/}).promise();
};

const updateDatabase = async () => {
  await dynamodb.updateItem({/*...*/}).promise();
};
```

Javascript has a built-in convenience method to run things in parallel. This is the `Promise.all`, that gets a *collection of Promises* and returns a Promise with all the results. Whenever you want to run multiple things concurrently, this is the tool to use.

To use it to run the two operations in parallel, use:

```
await Promise.all([moveObject(), updateDatabase()]);
```



Parallel execution

There are multiple things to notice here. First, `Promise.all` gets *Promises*. Remember that *calling* an async function returns a Promise, so don't forget to call the functions you want to run.

Second, it needs a *collection of Promises*, which is usually an array. Notice the `([ ... ])` structure for the call to `Promise.all`.

And third, `await` is only used for the `Promise.all` and not for the individual async function calls. This is because you want to wait for *all of them* to finish before moving on.

> **Tip**
>
> `await` stops the async function execution. In the case of `Promise.all` you want to stop only once, to wait for all the input Promises to settle. That's why you only need one `await`.

Results

In the above example none of the operations returned a value. Let's see how to handle values returned from `Promise.all`!

For the sake of illustration, let's say there are two unrelated piece of data an async function needs: a user object and a list of groups. Both of these are stored in a database and thus they are available via async functions.

```
const getUser = async () => {
  // gets the user object
  return user;
}
const getGroups = async () => {
  // gets the groups
  return groups;
}
```

Notice in this example that the `getGroups` function does not need the user object. This makes them able to run in parallel.

A serial implementation calls these functions separately and stores their results in variables:

```
const user = await getUser();
const groups = await getGroups();

// user and groups are available
```

There are two `await`s here, so the function will stop twice. No matter which you put first, the total execution time will be the sum of the calls. As we've seen previously, the `Promise.all` runs the Promises returned by the async function in parallel. It also returns a Promise with an *array* of the results.

To run the two calls in parallel and extract the results, use the array destructuring operator:

```
const [user, groups] = await Promise.all([
  getUser(),
  getGroups(),
]);

// user and groups are available
```

Errors in Promise.all

When *any* of the input Promises are rejected then the resulting Promise will be rejected as well. This behavior allows errors to propagate seamlessly.

```
try {
  const [user, groups] = await Promise.all([
    getUser(), // getUser returns a value
    getGroups(), // getGroups throws an exception
  ]);
} catch(e) {
  // handle error thrown by either of calls
}
```

Even when async functions are independent in a sense that one does not use the result of the other, you might not want to run them in parallel in some cases. In the example above with the `moveObject` and `updateDatabase` async functions, you might not want to run the latter if the former is failed.

```
await Promise.all([moveObject(), updateDatabase()]);
```

When this code is run, it can produce the following results:

Object is moved	Database is updated
✓	✓
✓	-
-	✓
-	-

By using serial execution, you can make sure that the later operations won't run unless the earlier ones are finished:

```
await moveObject();
await updateDatabase();
```

When this code is run, the database is guaranteed to be unchanged if the move object failed:

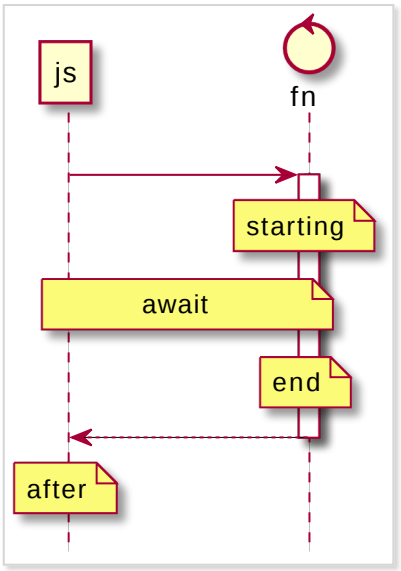
Object is moved	Database is updated
✓	✓
✓	-
-	-

Early init

When you create a Promise, it starts executing. But the async function only stops when it encounters an `await`. This allows a structure that starts the async processing early but only stops for it when the result is needed.

For example, consider this code, which waits for the function to finish before moving on:

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

const fn = async () => {
  console.log("starting");
  await wait(100);
  console.log("end");
}

await fn();
console.log("after");

// starting
// end
// after
```

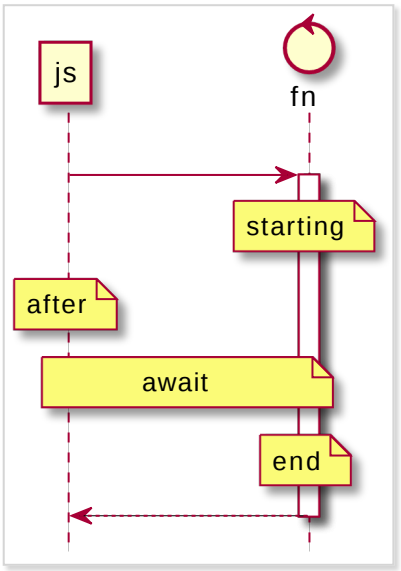


Eager await

But when the `await` is later, the order of execution changes:

```
const fnProm = fn();
console.log("after");
await fnProm;

// starting
// after
// end
```

This allows an easy way to run things "in the background" while also getting the results when they are needed.



Deferred await

For example, a web server might do several things when a request arrives, one of them to fetch the user session from a cookie and a session store.

```
const http = require("http");

const getUserSession = async (req) => {
  // extract the session cookie
  // fetch the session from Redis
  // return the user object
};

const server = http.createServer(async (req, res) => {
  // validate request
  // fetch configuration
  const session = await getUserSession(req);
  // handle request
});

server.listen(8080);
```

In the above example, everything happens sequentially. The server processes the request and then fetches the user session.

To start getting the user session right after the request comes, you can separate calling the async function and the `await`:

```
const server = http.createServer(async (req, res) => {
  const sessionProm = getUserSession(req);
  // validate request
  // fetch configuration
  const session = await sessionProm;
  // handle request
});
```

In this case, the session is still available when it's needed, but the process starts sooner. This reduces the total time the user experiences.

There is catch though. As we've discussed in the Error handling chapter, if the async function throws an error it will be propagated through the `await`. This means if the execution reaches no `await`s for that Promise, the error won't be handled.

This is a problem.

Consider this code:

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


const fn = async () => {
  await wait(100);
  throw new Error("Something bad happened");
}

try{
  const p = fn();
  throw new Error("There was an error");
  await p;
}catch(e) {
  console.log(e.message);
}
```

This prints:

```
There was an error
(node:170) UnhandledPromiseRejectionWarning: Error: Something
bad happened
(node:170) UnhandledPromiseRejectionWarning: Unhandled promise
rejection. This error originated either by throwing inside of an
async function without a catch block, or by rejecting a promise
which was not handled with .catch(). To terminate the node
process on unhandled promise rejection, use the CLI flag
`--unhandled-rejections=strict` (see
https://nodejs.org/api/cli.html#cli_unhandled_rejections_mode).
(rejection id: 1)
(node:170) [DEP0018] DeprecationWarning: Unhandled promise
rejections are deprecated. In the future, promise rejections that
are not handled will terminate the Node.js process with a
non-zero exit code.
```

An unhandled rejection is bad. There is a catch-all `unhandledRejection` event that allows handling it, but that leads to unmaintainable code. A proper solution is to make sure that the Promise is always `await`ed upon. This usually leads to more complicated code.

There are valid cases where the early init pattern is useful, especially when any exception is catastrophic. But when you have a backend server that multiple clients can call, it is better to avoid.

Multiple awaits

What happens if there are mutliple `await`s for a Promise? This does not result in calling the function multiple times. When a Promise is settled (resolved or rejected), `await` just returns the result (or throws an exception).

Consider the following code:

```
const fn = async () => {
  console.log("called");
  await wait(100);
  return "result";
}

(async () => {
  const p = fn();
  console.log(await p);
  console.log(await p);
})();


// called
// result
// result
```

Returning multiple values

As we've discussed in the The Promise constructor chapter, just like normal functions, a Promise can have a single return value. But oftentimes you'll want to return multiple things.

Fortunately, Javascript supports the object and the array destructuring operators that makes it easy to use a single variable to hold multiple things.

For example, a Promise can resolve with an array of values:

```
const getValues = () => new Promise((res) => {
  const user = "user";
  const group = "group";
  res([user, group]);
});

const [user, group] = await getValues();
// user, group
```

Or an object with known fields:

```
const getValues = () => new Promise((res) => {
  const user = "user";
  const group = "group";
  res({user, group});
});

const {user, group} = await getValues();
// user, group
```

Of course, it's the same pattern that you'd use for synchronous functions:

```
const getValues = () => {
  const user = "user";
  const group = "group";
  return {user, group};
};

const {user, group} = getValues();
// user, group
```

And works the same with async functions too:

```
const getValues = async () => {
  const user = "user";
  const group = "group";
  return {user, group};
};

const {user, group} = await getValues();
// user, group
```

Convert between Promises and callbacks

While Promises and async/await are increasingly the primary way to write asynchronous code in Javascript, callbacks are still used in many places. Several libraries that adopted that style are slow to migrate to the more modern alternative, and browser (and Node) APIs are also slow to change.

For example, marked, a markdown compiler needs a callback when it's used in asynchronous mode:

```
marked(text, options, (err, result) => {
  // result is the compiled markdown
});
```

Similarly, `setTimeout` invokes a function when the time is up:

```
setTimeout(callback, 100);
```

Not to mention a ton of web APIs, such as Indexed DB, FileReader, and others. Callbacks are still everywhere, and it's a good practice to convert them to Promises especially if your code is already using async/await.

Callback styles

Callbacks implement the continuation-passing style programming where **a function** instead of returning a value **calls** a continuation, in this case, **an argument function**. It is especially prevalent in Javascript as it does not support synchronous waiting. Everything that involves some future events, such as network calls, an asynchronous API, or a simple timeout is only possible by using a callback mechanism.

There are several ways callbacks can work. For example, `setTimeout` uses a callback-first pattern:

```
setTimeout(callback, ms);
```

Or functions can get multiple functions and call them when appropriate:

```
const checkAdmin = (id, isAdmin, notAdmin) => {
  if (/* admin logic */) {
    isAdmin();
  }else {
    notAdmin();
  }
};
```

How the callback is invoked can also vary. For example, it might get multiple arguments:

```
const getUserData = (id, cb) => {
  const user = /* get user */
  cb(user.id, user.profile, user.avatar);
};
```

Also, asynchronicity might be implemented as an object that emits events:

```
const reader = new FileReader();
reader.onload = (event) => {
  // event.target.result
}
reader.onerror = (error) => {
  // handle error
};
reader.readAsDataURL(blob);
```

Node-style callbacks

As there are multiple equally reasonable ways to implement callbacks, it was a mess at first. A de-facto standard emerged, which is now called Node-style or error-first callbacks. It is used almost everywhere in Javascript whenever a callback is needed.

A Node-style callback has three characteristics:

- The callback function is the last argument

- It is called with an error object first, then a result (`(error, result)`)
- It returns only one result

As an illustration, this function implements a Node-style callback:

```
const getUser = (id, cb) => {
  const user = /* get user */;
  if (user) {
    // success
    cb(null, user);
  }else {
    // error
    cb(new Error("Failed to get user"));
  }
};
```

Notice that when there is no error, the first argument is `null`. This allows the caller to easily check whether the execution failed or succeeded:

```
getUser(15, (error, result) => {
  if (error) {
    // handle error
  } else {
    // handle result
  }
});
```

With a callback structure that is used almost exclusively, it is possible to convert between Promises and callbacks more easily.

Convert callbacks to Promises

This is the more prominent direction as it's a common task to integrate a callback-based function into an async/await flow. Let's see how to do it!

Promise constructor

The Promise constructor is the low-level but universally applicable way to convert callbacks to Promises. It works for every callback style and it needs only a few lines of code.

The Promise constructor gets a function with two arguments: a `resolve` and a `reject` function. When one of them is called, the Promise will settle with either a result passed to the `resolve` function, or an error, passed to the `reject`.

```
new Promise((res, rej) => {

  // resolve with a value
  // res(value)

  // reject with error:
  // rej(error)
})
```

This makes it easy to call a callback-based function and convert it to a Promise which then can be `await`-ed:

```
// Node-style
new Promise((res, rej) => getUser(15, (err, result) => {
  if (err) {
    rej(err);
  }else {
    res(result);
  }
}))

// setTimeout
new Promise((res) => setTimeout(res, 100));

// event-based
new Promise((res, rej) => {
  const reader = new FileReader();
  reader.onload = (event) => {
    // resolve the Promise
    res(event.target.result);
  }
  reader.onerror = (error) => {
    // reject the Promise
    rej(error)
  };
  reader.readAsDataURL(blob);
});
```

Promisified functions

The above examples show how to call a callback-based function and get back a Promise, but it requires wrapping every call with the Promise boilerplate. It would be better to have a function that mimics the original one but without the callback. Such a function would get the same arguments minus the callback and return the Promise.

To make promisified versions of functions, it is only a matter of wrapping the Promise constructor in a function that gets the arguments before the callback:

```
// setTimeout
const promisifiedSetTimeout = (ms) =>
  new Promise((res) => setTimeout(res, ms));

// FileReader
const promisifiedFileReader = (blob) =>
  new Promise((res, rej) => {
    const reader = new FileReader();
    reader.onload = (event) => {
      res(event.target.result);
    }
    reader.onerror = (error) => {
      rej(error)
    };
    reader.readAsDataURL(blob);
  });

// checkAdmin
const promisifiedCheckAdmin = (id) => new Promise((res) => {
  if (/* admin logic */) {
    res(true);
  }else {
    res(false);
  }
};
```

These functions are direct replacements to the callback-based originals and can be directly used in an async/await workflow:

```
// timeout
setTimeout(() => {
  console.log("Timeout reached");
}, 100);

await promisifiedSetTimeout(100);
console.log("Timeout reached");

// checkAdmin
const checkAdmin = (id, isAdmin, notAdmin) => {
  if (/* admin logic */) {
    isAdmin();
  }else {
    notAdmin();
  }
};

checkAdmin(15, () => {
  console.log("User is an admin");
}, () => {
  console.log("User is not an admin");
});

const admin = await promisifiedCheckAdmin(15);
console.log(`User is admin: ${admin}`);

// FileReader
const dataURI = await promisifiedFileReader(blob);
```

util.promisify

While the Promise constructor offers a universal way to transform callbacks to Promises, when the callback pattern follows the Node-style there is an easier way. The `util.promisify` gets the callback-based function and returns a promisified version.

```
import util from "util";

const promisifiedGetUser = util.promisify(getUser);

const user = await promisifiedGetUser(15);
// user is the result object
```

When the promisified function is called, the callback argument is added automatically and it also converts the result (or error) to the return value of the Promise.

How it works is no magic though. It uses the Promise constructor pattern we've discussed above, and uses the spread syntax to allow arbitrary amount of arguments. A simplified implementation looks like this:

```
const promisify = (fn) => (...args) =>
  new Promise((res, rej) => {
    fn(...args, (err, result) => {
      if (err) {
        rej(err);
      }else {
        res(result);
      }
    });
  });
```

Usual problems

Handle this

The value of `this` is complicated in Javascript. It can get different values depending on how you call a function.

For example, classes can have instance variables, attached to `this`:

```
class C {
  constructor() {
    this.var = "var";
  }
  fn() {
    console.log(this.var);
  }
}
```

But when you have an object of this class, whether you call this function directly on the object or extract it to another variable makes a difference in the value of `this`:

```
new C().fn(); // var

let fn = new C().fn;
fn(); // TypeError: Cannot read property 'var' of undefined
```

This affects how to promisify the methods of this object as the `util.promisify` requires just the function and not the whole object. Which, in turn, breaks `this`.

For example, let's say there is a Database object that creates a connection in its constructor then it offers methods to send queries:

```
class Database {
  constructor() {
    this.connection = "database connection";
  }
  getUser(id, cb) {
    if (!this.connection) {
      throw new Error("No connection");
    }
    setTimeout(() => {
      if (id >= 0) {
        cb(null, `user: ${id}`);
      }else {
        cb(new Error("id must be positive"));
      }
    }, 100);
  }
}

const database = new Database();
database.getUser(15, (error, user) => {
  // handle error or user
})
```

Using `util.promisify` would break the `getUser` function as it changes the value of `this`:

```
const promisifiedDatabaseGetUser =
  util.promisify(database.getUser);
await promisifiedDatabaseGetUser(15);
// Error: Cannot read property 'connection' of undefined
```

To solve this, you can bind the object to the function, forcing the value of `this`:

```
const promisifiedDatabaseGetUser =
  util.promisify(database.getUser.bind(database));
const user = await promisifiedDatabaseGetUser(15);
```

function.length

The `length` of a function is how many arguments it needs. Don't confuse this with the Array's `length`, as that is how many *elements* in the array.

For example, this function needs 2 arguments, so its `length` is 2:

```
const fn = (a, cb) => {};

console.log(fn.length); // 2
```

It is rarely used, but some libraries depend on it having the correct value, such as memoizee, which determines how to cache the function call or marked, a Markdown compiler, to decide whether its configuration is called async or sync.

While the length of the function is rarely used, it can cause problems. `util.promisify` does not change it, so the resulting function will have the same length as the original one, even though it needs fewer arguments.

```
console.log(fn.length); // 2

const promisified = util.promisify(fn);

console.log(promisified.length); // 2
```

Convert Promises to callbacks

The other direction is to have a Promise-returning function (usually an `async` function) and you need to convert it to a callback-based one. It is much rarer than the other way around, but there are cases when it's needed, usually when a library expects an asynchronous function and it supports only callbacks.

For example, the marked library supports a `highlight` option that gets the code block and returns a formatted version. The highlighter gets the code, the language, and a callback argument, and it is expected to call the last one with the result.

```
import marked from "marked";
import util from "util";

const promisifiedMarked = util.promisify(marked);

const res = await promisifiedMarked(md, {
  // highlight gets a callback
  highlight: (code, lang, cb) => {
    const result = "Code block";
    cb(null, result);
  }
});
```

As with the promisification, there is a universal and a Node-callback-specific way to convert an async function to use callbacks. The universal way is to use an async IIFE (Immediately-Invoked Function Expression) and add use `then` to interface with the callback when there is a result or an error:

```
highlight: (code, lang, cb) => {
  (async () => {
    const result = "Code block";
    return result;
  })().then((res) => cb(null, res), (err) => cb(err));
}
```

This structure allows all callback styles as you control how a result or an error is communicated with the callback function.

For Node-style callbacks, you can use the `util.callbackify` function that gets an async function and returns a callbackified version:

```
highlight: util.callbackify(async (code, lang) => {
  const result = "Code block";

  return result;
})
```

This yields a convenient structure and it is suitable in most cases.

Also, it changes the resulting function's `length` by one, as it needs a callback as well as all the original arguments:

```
console.log(fn.length) // 2
console.log(util.callbackify(fn).length) // 3
```

Promise timeouts

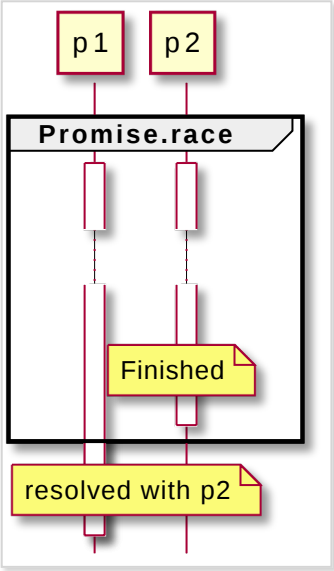
Promises in Javascript have no concept of time. When you use `await` or attach a function with `.then()`, it will **wait until the Promise is either resolved or rejected**. This is usually not a problem as most async tasks finish within a reasonable time and their result is needed.

But when a client is waiting for a response of, let's say, an HTTP server, it's better to return early with an error giving the caller a chance to retry rather than to wait for a potentially long time.

Fortunately, there is a built-in Promise helper function that can be used to add timeout functionality to any Promise-based construct.

Promise.race

The Promise.race is a global property of the Promise object. **It gets an array of Promises and waits *for the first one* to finish**. Whether the race is resolved or rejected depends on the winning member.



p2 finishes first

For example, the following code races two Promises. The second one resolves sooner, and the result of the other one is discarded:

```
const p1 =
  new Promise((res) => setTimeout(() => res("p1"), 1000));
const p2 =
  new Promise((res) => setTimeout(() => res("p2"), 500));

const result = await Promise.race([p1, p2]);
// result = p2
```

Similarly, it works for rejections also. If the winning Promise is rejected, the race is rejected:

```
const p1 =
  new Promise((res) => setTimeout(() => res("p1"), 1000));
const p2 =
  new Promise((_r, rej) => setTimeout(() => rej("p2"), 500));

try {
  const result = await Promise.race([p1, p2]);
} catch(e) {
  // e = p2
}
```

The arguments of the `Promise.race` function are *Promises*. This makes it work with `async` functions too:

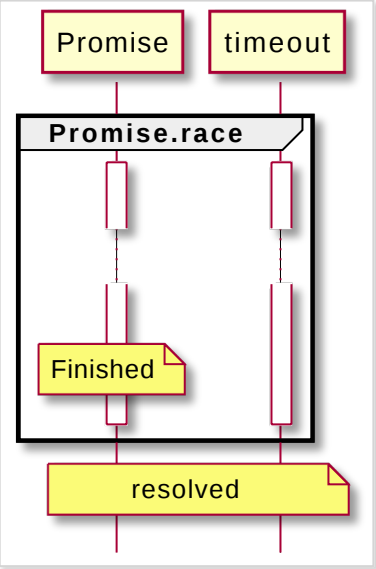
```
const fn = async (time, label) => {
  await new Promise((res) => setTimeout(res, time));
  return label;
}

const result =
  await Promise.race([fn(1000, "p1"), fn(500, "p2")])
// result = p2
```

Just don't forget to *call* the `async` functions so that the race gets Promises. This can be a problem with anonymous functions and those need to be wrapped IIFE-style:

```
const result = await Promise.race([
  fn(1000, "p1"),
  (async () => {
    await new Promise((res) => setTimeout(res, 500));
    return "p2";
  })(),
]);
// result = p2
```

Timeout implementation

With `Promise.race`, it's easy to implement a timeout that supports any Promises. Along with the async task, **start another Promise that rejects when the timeout is reached**. Whichever finishes first (the original Promise or the timeout) will be the result.

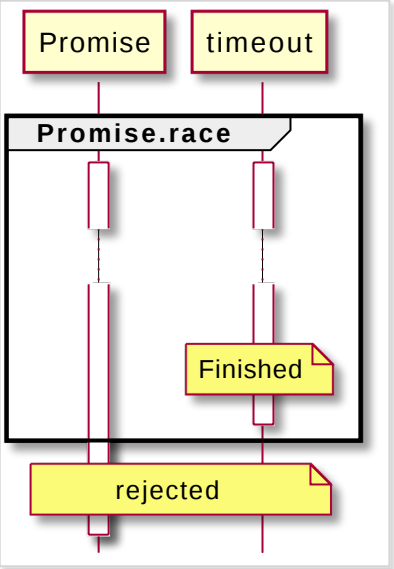


The Promise finishes before the timeout

```
const timeout = (prom, time) =>
  Promise.race([
    prom,
    new Promise((_r, rej) => setTimeout(rej, time))
  ]);
```

With this helper function, wrap any Promise and it will reject if it does not produce a result in the specified time.

```
// resolves in 500 ms
const fn = async () => {
  await new Promise((res) => setTimeout(res, 500));
  return "p2";
}

// finishes before the timeout
const result = await timeout(fn(), 1000);
// result = p2

// timeouts in 100 ms
await timeout(fn(), 100);
// error
```



The timeout finishes before the Promise

It's important to note that it *does not terminate the Promise if the timeout is reached*, just discard its result. If it consists of multiple steps, they will still run to completion eventually.

Clear timeout

The above solution uses a `setTimeout` call to schedule the rejection. Just as the original Promise does not terminate when the timeout is reached, the timeout Promise **won't cancel this timer when the race is finished**.

While this does not change how the resulting Promise works, it can cause side-effects. The event loop needs to check whether the timer is finished, and some environments might work differently if there are unfinished ones.

Let's make the wrapper function use `Promise.finally` to clear the timeout!

```
const timeout = (prom, time) => {
  let timer;
  return Promise.race([
    prom,
    new Promise((_r, rej) => timer = setTimeout(rej, time))
  ]).finally(() => clearTimeout(timer));
}
```

The above implementation saves the `setTimeout`'s result as `timer` and clears it when the race is over.

Error object

The race is over and there is a rejection. Was it because of the timeout or there was an error thrown from the Promise?

The above implementation does not distinguish between errors and this makes it hard to handle timeouts specifically.

```
const fn = async () => {
  throw new Error();
};

try {
  const result = await timeout(fn(), 1000);
}catch(e) {
  // error or timeout?
}
```

The solution is to add a third argument that is the timeout rejection value. This way there is an option to differentiate between errors:

```
const timeout = (prom, time, exception) => {
  let timer;
  return Promise.race([
    prom,
    new Promise((_r, rej) =>
      timer = setTimeout(rej, time, exception)
    )
  ]).finally(() => clearTimeout(timer));
}
```

What should be a timeout error object?

Symbols in Javascript are unique objects that are only equal to themselves. This makes them perfect for this use-case. Pass a Symbol as the timeout error argument then check if the rejection is that Symbol.

```
const timeoutError = Symbol();
try {
  const result = await timeout(prom, 1000, timeoutError);
  // handle result
}catch (e) {
  if (e === timeoutError) {
    // handle timeout
  }else {
    // other error
    throw e;
  }
}
```

This construct allows handling the timeout error specifically.

The async serializer pattern

Let's say you have a function that does some calculation but only if it hasn't been done recently, effectively caching the result for a short amount of time. In this case, **subsequent calls to the function affect each other**, as when there is a refresh all of them need to wait for it. An implementation that does not take concurrency into account can potentially run the refresh several times in parallel when the cache expires.

This implementation is broken:

```
// returns a function that caches its result for 2 seconds
// broken, don't use it
const badCachingFunction = (() => {
  const cacheTime = 2000;
  let lastRefreshed = undefined;
  let lastResult = undefined;
  return async () => {
    const currentTime = new Date().getTime();
    // check if cache is fresh enough
    if (lastResult === undefined ||
      lastRefreshed + cacheTime < currentTime) {
      // refresh the value
      lastResult = await refresh();
      lastRefreshed = currentTime;
    }
    return lastResult;
  }
})();
```

When two concurrent calls come when the function needs to refresh the value, the `refresh` function is called twice:

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

const refresh = async () => {
  console.log("refreshing")
  await wait(1000);
};

const badCachingFunction = ...;

badCachingFunction();
badCachingFunction();
// refreshing
// refreshing
```

The correct way to handle this is to **make the second function wait for the refresh process** without starting a separate one.

One way to do this is to **serialize the calls** to the caching function. This makes *every* function to wait for all the previous ones to finish, so multiple calls only trigger a single refresh process.

This can be done in a variety of ways, but the easiest one is to make the functions run one after the other. In this case, when one needs to refresh the value, the other ones will wait for it to finish and won't start their own jobs.

Another use-case I needed a solution like this is when backend calls needed a token and that token expired after some time. When a call hit an Unauthorized error, it refreshed the token and used the new one to retry. Other backend calls needed to wait for the new token before they could be run. In this case, it wasn't just performance-related as a new token invalidated all the old ones.

Why await is not a solution

The trivial solution is to use `await` that achieves serialization easily. After all, that's what that keyword is for.

```
await badCachingFunction();
badCachingFunction();
// refreshing
```

But that requires *collaboration between the calls*, and that is not always possible. For example, when the function is called in response to multiple types of events, `await` is not possible to coordinate between them:

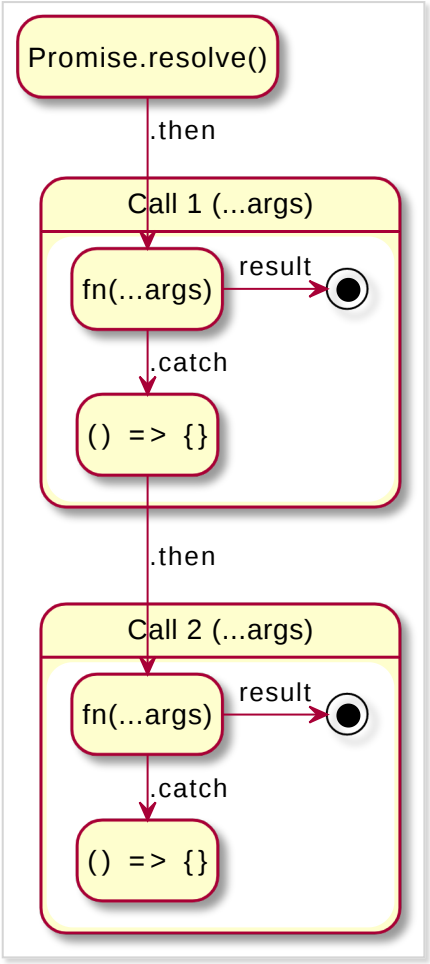
```
document.querySelector("#btn").addEventListener("click", () => {
  fn();
})

window.addEventListener("message",  => {
  fn();
});
```

In this case, the function call must do the coordination.

A general-purpose solution

The solution is to keep a queue of Promises that chains them one after the other. It is just a few lines of code and it is general purpose, allowing any function be serialized:

```
const serialize = (fn) => {
  let queue = Promise.resolve();
  return (...args) => {
    const res = queue.then(() => fn(...args));
    queue = res.catch(() => {});
    return res;
  };
};
```



Queue structure

The `Promise.resolve()` is the start of the queue. Every other call is appended to this Promise.

The `queue.then(() => fn(...args))` adds the function call to the queue and it saves its result in `res`. It will be resolved when *the current and all the previous calls are resolved*.

The `queue = res.catch(() => {})` part makes sure that the queue won't get stuck in rejection when one part of it is rejected.

Wrapping the caching function with this serializer makes sure that a single refresh is run even for multiple calls:

```
const fn = serialize((() => {
  const cacheTime = 2000;
  let lastRefreshed = undefined;
  let lastResult = undefined;
  return async () => {
    const currentTime = new Date().getTime();
    // check if cache is fresh enough
    if (lastResult === undefined ||
      lastRefreshed + cacheTime < currentTime) {
      // refresh the value
      lastResult = await refresh();
      lastRefreshed = currentTime;
    }
    return lastResult;
  }
})());

fn();
fn();
// refreshing
```

The async disposer pattern

A recurring pattern is to **run some initialization code** to set up some resource or configuration, then use the thing, **and** finally **do some cleanup**. It can be a global property, such as freezing the time with timekeeper, starting a Chrome browser with Puppeteer, or creating a temp directory. In all these cases, you need to make sure the modifications/resources are properly disposed of, otherwise, they might spill out to other parts of the codebase.

For example, this code creates a temp directory in the system tmpdir then when it's not needed it deletes it. This can be useful when, for example, you want to use ffmpeg to extract some frames from a video and need a directory to tell the `ffmpeg` command to output the images to.

```
// create the temp directory
const dir = await fs.mkdtemp(
  await fs.realpath(os.tmpdir()) + path.sep
);
try {

  // use the temp directory

} finally {
  // remove the directory
  fs.rmdir(dir, {recursive: true});
}
```

A similar construct is `console.time` that needs a `console.timeEnd` to output the time it took for a segment of the code to run:

```
console.time("name");
try {
  // ...
} finally {
  console.timeEnd("name");
}
```

Or when you launch a browser, you want to make sure it's closed when it's not needed:

```
const browser = await puppeteer.launch({/* ... */});
try {
  // use browser
} finally {
  await browser.close();
}
```

All these cases share the `try..finally` structure. Without it, **an error can jump over the cleanup logic**, leaving the resource initialized (or the console timing still ticking):

```
const browser = await puppeteer.launch({/* ... */});

// if there is an error here the browser won't close!

await browser.close();
```

In other languages, such as Java, this is known as try-with-resources and it is built into the language. For example, the `BufferedReader` is closed after the block:

```
try (BufferedReader br =
  new BufferedReader(new FileReader(path))
) {
  return br.readLine();
}
```

This builds on the AutoCloseable interface's `close` method so it's easy to adapt to custom resource types.

But there is no such structure in Javascript. Let's see how to implement one!

Disposer pattern

One problem with the `try..finally` structure we've seen above is **how to return a value from inside the `try` block**. Let's say you want to take a screenshot of a website and want to use the resulting image later.

```
const browser = await puppeteer.launch({/* ... */});
try {
  const page = await browser.newPage();
  // ...
  const screenshot = await page.screenshot({/* ... */});

  // the browser can be closed
  // but how to use the screenshot outside the try..finally?
} finally {
  await browser.close();
}
```

A suboptimal solution is to declare a variable outside the block and use it to store the image:

```
let screenshot;

const browser = await puppeteer.launch({/* ... */});
try {
  const page = await browser.newPage();
  // ...
  screenshot = await page.screenshot({/* ... */});
} finally {
  await browser.close();
}

// use screenshot
```

A better solution is to **use a function**:

```
const makeScreenshot = async () => {
  const browser = await puppeteer.launch({/* ... */});
  try {
    const page = await browser.newPage();
    // ...
    return await page.screenshot({/* ... */});
  } finally {
    await browser.close();
  }
}

const screenshot = await makeScreenshot();
```

This is the basis of the disposer pattern. The difference is that **instead of hardcoding the logic inside the `try..finally` block, it gets a function** that implements that part:

```
const withBrowser = async (fn) => {
  const browser = await puppeteer.launch({/* ... */});
  try {
    return await fn(browser);
  } finally {
    await browser.close();
  }
}

const screenshot = await withBrowser(async (browser) => {
  const page = await browser.newPage();
  // ...
  return await page.screenshot({/* ... */});
});
```

The `withBrowser` function contains the logic to launch and close the browser, and **the `fn` function gets and uses the browser instance**. Whenever the argument function returns, the browser is automatically closed, no additional cleanup logic is needed. This structure provides an elegant way to prevent non-closed resources hanging around.

An interesting aspect of this pattern is that it is one of the few cases where there is a difference between `return fn()` and `return await fn()`. Usually, it does not matter if an async function returns a Promise or the *result* of the Promise. But in this case, without the `await` the `finally` block runs *before* the `fn()` call is finished.

A potential problem is when there is some task is running in the argument function when it returns. This can happen when it starts an asynchronous task and does not wait for it to finish. In this case, the cleanup logic runs and closes the browser instance that can cause an error. It is usually the sign that an `await` is missing.

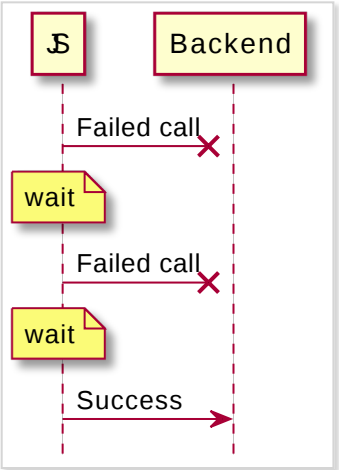
Retrying async operations

Backoff algorithm

A Javascript application made a `fetch` call to a remote server and it failed. How should it retry the request so that it eventually succeeds while also minimizing the calls made?

A backoff algorithm makes sure that **when a target system can not serve a request it is not flooded with subsequent retries**. It achieves this by introducing a **waiting period between the retries** to give the target a chance to recover.

The need for a backoff builds on the observation that a service is unavailable when it is overloaded and sending more requests only exacerbates the problem. When all callers temporarily cease adding more load to the already overloaded service usually smooths the traffic spikes with only a slight delay.



Waiting between retries

The backoff algorithm used determines how much to wait between the retries. The best configuration is actively researched in the case of network congestion, such as when a mobile network is saturated. It's fascinating to see that different implementations yield drastically different effective bandwidth.

Retrying a failed call to a remote server is a much easier problem and doing it right does not require years of research. In this article, you'll learn how to implement a backoff solution in Javascript that is good enough for all practical purposes.

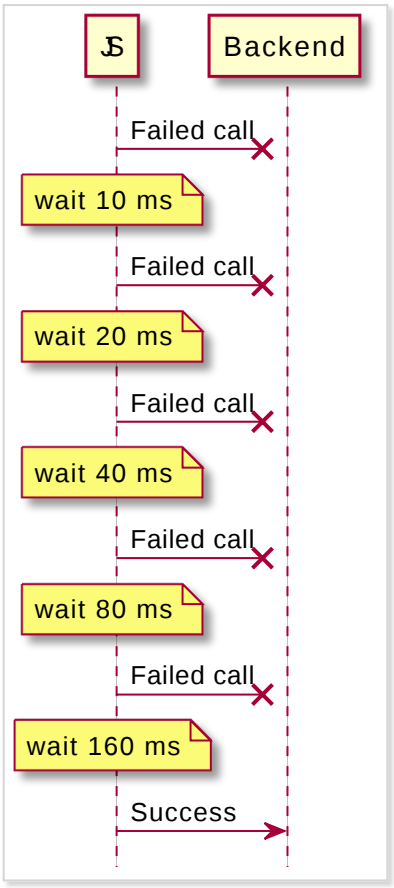
Exponential backoff

But let's first revisit the problem of choosing the backoff strategy, i.e. **how much to wait between the retries**! Sending requests too soon puts more load on the potentially struggling server, while waiting too long introduces too much lag.

The exponential backoff became the standard algorithm to use. It waits exponentially longer for subsequent retries, which makes the first few tries happen with only a slight delay while it reaches longer periods quickly.

It has two parameters: the **delay of the first period** and the **growth factor**. For example, when the first retry waits 10ms and the subsequent ones double the previous values, the waiting times are: 10, 20, 40, 80, 160, 320, 640, 1280, ...

Notice that the sum of the first 3 attempts is less than 100 ms, which is barely noticeable. But it reaches >1 second in just 8 tries.



Exponential backoff

Javascript implementation

There are two pitfalls when implementing a backoff algorithm.

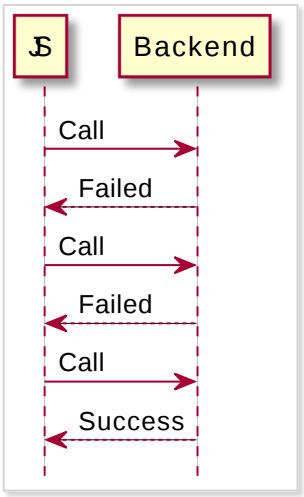
First, make sure to **wait only before retries and not the first request**. Waiting first and sending a request then introduces a delay even for successful requests.

And second, make sure to **put a limit to the number of retries** so that the code eventually gives up and throws an error. Even if the call eventually succeeds, something upstream timeouts in the meantime, and the user likely gets an error. Returning an error in a timely manner and letting the user retry the operation is a good UX practice.

There are two types of operation in terms of retrying: rejection-based and progress-based. Let's discuss each of them!

Rejection-based retrying

This type of call **either succeeds or fails** and it is usually implemented as a Promise resolving or rejecting. A canonical example is a "Save" button that may fail and in that case, retrying means sending the request again.



Rejection-based retrying

For illustration, this operation emulates a network call that fails 90% of the time:

```
const wait = (ms) => new Promise((res) => setTimeout(res, ms));
const maybeFail = (successProbability, result, error) =>
  new Promise((res, rej) => Math.random() < successProbability
    ? res(result)
    : rej()
  );

const maybeFailingOperation = async () => {
  await wait(10);
  return maybeFail(0.1, "result", "error");
}
```

Calling this function without a retry mechanism is bad UX. The user would need to click the button repeatedly until it succeeds.

A general-purpose solution that can take any async function and retry it for a few times before giving up:

```
const callWithRetry = async (fn, depth = 0) => {
  try {
    return await fn();
  }catch(e) {
    if (depth > 7) {
      throw e;
    }
    await wait(2 ** depth * 10);

    return callWithRetry(fn, depth + 1);
  }
}
```

And to use it with the operation defined above:

```
const result = await callWithRetry(maybeFailingOperation);
```

In the retry implementation, the `depth` argument indicates the number of calls made to the service so far. It determines how much to wait (`await wait(2 ** depth * 10)`) and when to give up (`if (depth > 7) {...}`). When the function calls itself it increments this value.

Interestingly, this is a scenario where using `return await` makes a difference. Usually, it does not matter if an async function returns a Promise or its resolved value. But not in this case, as the `try..catch` needs to wait for the `fn()` call to finish to handle exceptions thrown from it.

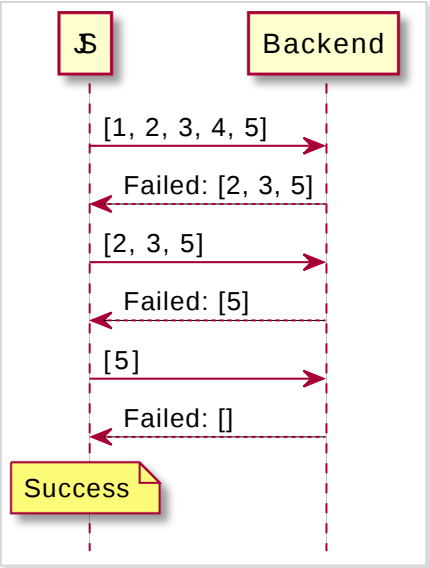
Note that this implementation retries for *any error*, even if it is a non-retriable one, such as sending invalid data.

Progress-based retrying

Another class of retriable operations is when **each call might make some progress towards completion but might not reach it**.

A great example of this is how DynamoDB's `batchWriteItem` call works. You define the items to store, and the call tries to save all of them. It then returns a list in the response indicating which elements failed. The retry operation only

needs to include these missed items and not the whole request.



Progress-based retrying

As an illustration, this function emulates a progressing call to a remote server. It returns a `progress` field that goes from 0 to 1, and a `result` when the call is fully finished. It also needs to know the *current progress*, which is a parameter for the call:

```
const progressingOperation = async (startProgress = 0) => {
  await wait(10);
  const progress = Math.round(
    Math.min(startProgress + Math.random() / 3, 1) * 10
  ) / 10;
  return {
    progress,
    result: progress === 1 ? "result" : undefined,
  };
}
```

In this case, not reaching completion still resolves the Promise, which mimics the DynamoDB API. But a different API might reject it. Fortunately, it's easy to convert a rejected Promise to a resolved one: `.catch()`.

In the DynamoDB `batchWriteItem` example, the `progress` return field is the list of the unprocessed items, and the `startProgress` parameter is the items to save.

To retry the above call until it reaches completion:

```
const callWithProgress = async (fn, status, depth = 0) => {
  const result = await fn(status);

  // check completion
  if (result.progress === 1) {
    // finished
    return result.result;
  }else {
    // unfinished
    if (depth > 7) {
      throw result;
    }
    await wait(2 ** depth * 10);

    return callWithProgress(fn, result.progress, depth + 1);
  }
}
```

To use this retrying call, use:

```
const result = await callWithProgress(progressingOperation);
```

Note that this implementation is not general-purpose as it builds on how the call returns progress.

Paginating with async generators

Many APIs limit how many results they return for a single call. For example, the functions of the AWS SDK that return lists are paginated operations. That means that you get a limited number of elements in one call along with a token to get the next batch. This can be a problem if you are not aware of it, as you might get all the elements during development but your function might break in the future.

But while getting the first batch is just `await lambda.listFunctions().promise()`, paginating until all the pages are retrieved requires more work.

Let's see how to make an `await`-able structure that reliable gets all the elements from an AWS API!

Pagination

In the case of Lambda functions, the `lambda.listFunctions` call returns a structure with a list of all your lambdas if you don't have too many of them (at most 50):

```
const functions = await lambda.listFunctions().promise();
```

```
{
  Functions: [...]
}
```

To simulate a paginated case, you can set the `MaxItems` parameter:

```
const functions = await lambda.listFunctions({
  MaxItems: 1
}).promise();
```

This time a `NextMarker` is also returned, indicating there are more items:

```
{
  Functions: [...],
  NextMarker: ...
}
```

To get the next batch, provide the last marker:

```
const functions = await lambda.listFunctions({
  MaxItems: 1,
  Marker: functions.NextMarker,
}).promise();
```

Then do so until no `NextMarker` is returned.

A solution with async generators

Async generators are a relatively new feature of Javascript. They are like traditional generator functions, but they are async, meaning you can `await` inside them.

To collect all the Lambda functions no matter how many calls are needed, use:

```
const getAllLambdas = async () => {
  const EMPTY = Symbol("empty");

  const res = [];
  for await (const lf of (async function*() {
    let NextMarker = EMPTY;
    while (NextMarker || NextMarker === EMPTY) {
      const functions = await lambda.listFunctions({
        Marker: NextMarker !== EMPTY ? NextMarker : undefined,
      }).promise();
      yield* functions.Functions;
      NextMarker = functions.NextMarker;
    }
  })()) {
    res.push(lf);
  }

  return res;
}

// use it
const functions = await getAllLambdas();
```

Async generator functions

A generator function that can use `await`. When it encounters a `yield` or a `yield*` the execution stops until a subsequent element is requested.

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

const gen = async function*() {
  console.log("start");
  await wait(100);
  yield 1;
  await wait(100);
  console.log("next");
  yield 2;
  await wait(100);
  console.log("end");
};

const it = gen();
console.log(await it.next());
console.log(await it.next());
console.log(await it.next());

// start
// { value: 1, done: false }
// next
// { value: 2, done: false }
// end
// { value: undefined, done: true }
```

Breaking it down

The most important thing is to keep track of the `NextMarker` returned by the last call and use that for making the next one. For the first call, `Marker` should be `undefined`, and to differentiate between the first and the last one (the one that returns no `NextMarker`), a Symbol is a safe option as it cannot be returned by the API.

```
const EMPTY = Symbol("empty");
let NextMarker = EMPTY;
while (NextMarker || NextMarker === EMPTY) {
  // Marker: NextMarker !== EMPTY ? NextMarker : undefined

  NextMarker = functions.NextMarker;
}
```

After the call, we need to `yield` the functions returned:

```
yield* functions.Functions;
```

The `yield*` makes sure that each element is returned as a separate value by the generator.

Finally, a `for await..of` loop collects the results and returns them as an Array:

```
const res = [];
for await (const lf of (async function*() {
  ...
})()) {
  res.push(lf);
}
return res;
```

To use it, just call the function and wait for the resulting Promise to resolve:

```
const functions = await getAllLambdas();
```

Making it generic

The same `Marker`/`NextMarker` pattern appears throughout the AWS SDK. But unfortunately, the naming is different for different services. For example, getting the CloudWatch Logs log groups you need to provide a `nextToken` parameter. This makes it impossible to support all the listing functions with a generic wrapper.

Luckily, as the pattern is the same, we can make a wrapper function that handles everything but the naming:

```
const getPaginatedResults = async (fn) => {
  const EMPTY = Symbol("empty");
  const res = [];
  for await (const lf of (async function*() {
    let NextMarker = EMPTY;
    while (NextMarker || NextMarker === EMPTY) {
      const {marker, results} = await fn(
        NextMarker !== EMPTY ? NextMarker : undefined
      );

      yield* results;
      NextMarker = marker;
    }
  })()) {
    res.push(lf);
  }

  return res;
};
```

It follows the same structure as before, but it gets an `fn` parameter that does the actual API call and returns the list and the marker.

To get all the Lambda functions with this wrapper:

```
const lambdas = await getPaginatedResults(async (NextMarker) => {
  const functions = await lambda.listFunctions(
    {Marker: NextMarker}
  ).promise();
  return {
    marker: functions.NextMarker,
    results: functions.Functions,
  };
});
```

Adapted to the log groups:

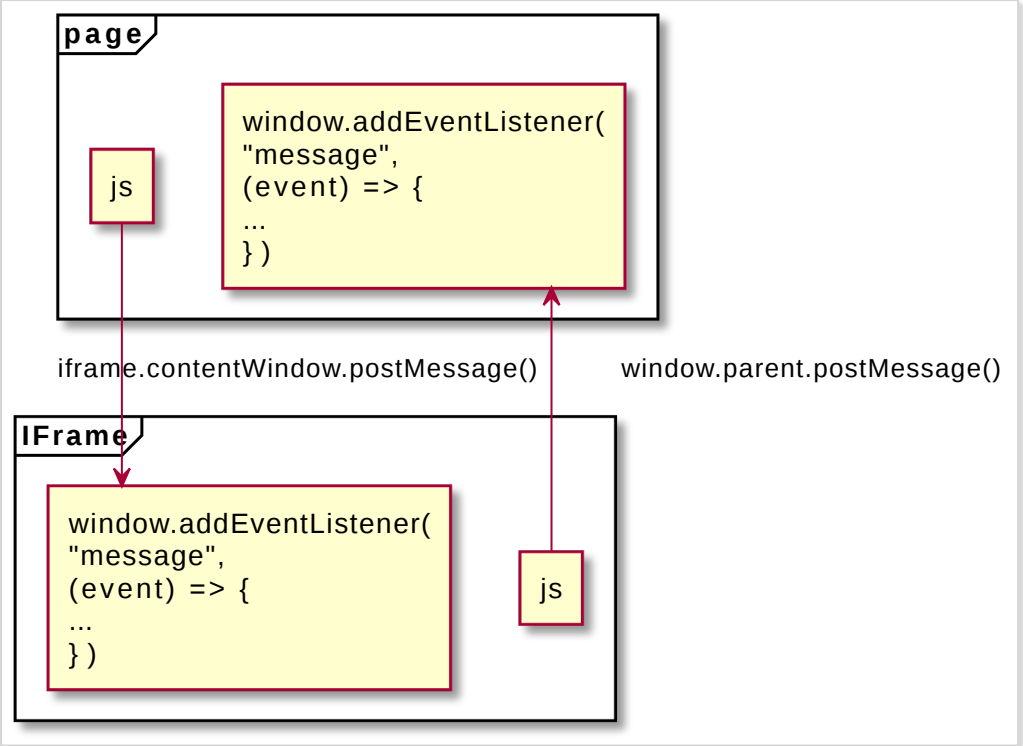
```
const logGroups =
  await getPaginatedResults(async (NextMarker) => {
    const logGroups = await logs.describeLogGroups(
      {nextToken: NextMarker}
    ).promise();
    return {
      marker: logGroups.nextToken,
      results: logGroups.logGroups,
    };
  });
```

Using async functions with postMessage

The `postMessage` call allows an asynchronous communication channel between different browsing contexts, such as with IFrames and web workers, where direct function calls don't work. It works by sending a message to the other side, then the receiving end can listen to `message` events.

For example, the page can communicate with an IFrame via `postMessage` and send events to each others' windows. The `iframe.contentWindow.postMessage()` call sends a message to the IFrame, while the `window.parent.postMessage()` sends a message back to the main page. The two ends can listen to messages from the other using `window.addEventListener("message", (event) => {...})`.

```
// index.html
const iframe = document.querySelector("iframe");

window.addEventListener("message", ({data}) => {
  // 3: receive response
  console.log("Message from iframe: " + data);
});

// 1: send request
iframe.contentWindow.postMessage([5, 2]);

// iframe.html
window.addEventListener("message", ({data}) => {
  // 2: send response
  window.parent.postMessage(event.data[0] + event.data[1]);
});
```



page-IFrame communication

For web workers, each worker has a separate message handler. The page can send a message to a specific worker using the `worker.postMessage()` call and listen for events from that worker using `worker.addEventListener("message", (event) => {...})`. On the other side, the worker sends and receives events using the global functions `postMessage()` and `addEventListener()`:

```
// index.html
const worker = new Worker("worker.js");
worker.addEventListener("message", ({data}) => {
  console.log("Message from worker: " + data); // 3
});

worker.postMessage([5, 5]); // 1

// worker.js
addEventListener("message", (event) => {
  postMessage(event.data[0] + event.data[1]); // 2
}, false)
```

Request-response communication

Both communicating with an IFrame and with a worker has problems. First, sending the request and handling the response is separated. **The receiving side uses a global (or a per-worker) event listener**, which is shared between the calls.

This is good for "notification-style" messages where one end wants to notify the other that something happened but not expecting a response. But **notification-style messaging is rare**. What is usually needed is a "request-response-style" messaging.

For example, one end uses an access token that the other one can refresh. The communication consists of a request to refresh and a response to that with the refreshed token. Or even when a user clicks on a button and the other side needs to handle this. This example seems like a "notification-style" message, but when the button needs to be disabled while the operation takes place (such as a save button) or there is a possibility of an error that the sender needs to know about, it's now a request-response.

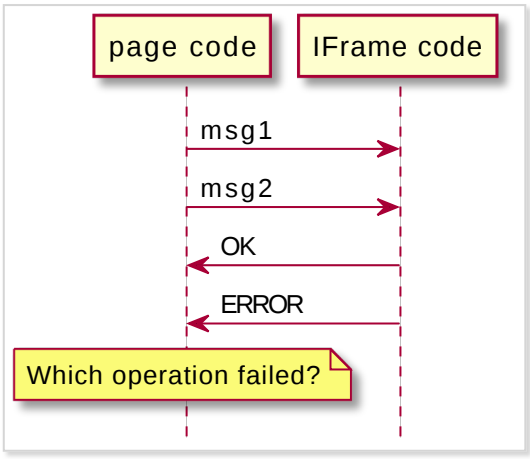
The global (or per-worker) message handler is not suited for pairing responses to requests. The ideal solution would be to hide all these complications behind an async function call and use `await` to wait for the result:

```
const result = await add(10, 5);
```

Let's see how to make such a function!

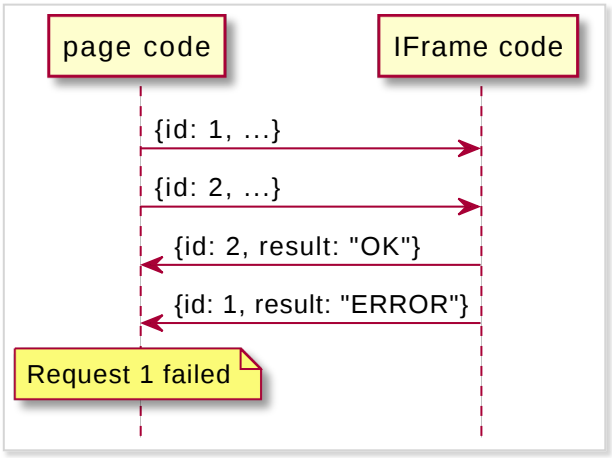
Response identification

The first task is to know which request triggered a given response. The problem with using the global event handler is that it's all too easy to rely on only one communication happening at a time. When you test your webapp, you test one thing at a time but users won't be that considerate. You can't assume only one request-response will happen at any one time.



Non-multiplexed channel

One solution is to use *request identifiers* to pair the responses to the requests.



Request ids

This works, and even though it requires some coding, this is a good solution.

MessageChannel

Fortunately, there is a better solution built into the language. MessageChannel allows a dedicated communication channel attached to the `postMessage` call. The sending end can listen for messages *on that channel* which is separated from all other calls, and the receiving end can send its responses through the *MessagePort* it got.

A MessageChannel creates two MessagePorts, one for each end of the communication. Each port supports both the `onmessage` and the `postMessage`, similar to the basic cross-context communication channels.

```
// index.html
const worker = new Worker("worker.js");
// create a channel
const channel = new MessageChannel();

// listen on one end
channel.port1.onmessage = ({data}) => {
  console.log("Message from channel: " + data); // 3
};

// send the other end
worker.postMessage([15, 2], [channel.port2]); // 1

// worker.js
addEventListener("message", (event) => {
  // respond on the received port
  event.ports[0].postMessage(event.data[0] + event.data[1]); // 2
}, false)
```

To attach a message handler to a port, use `port.onmessage = (event) => {...}`. If you use `addEventListener` then you need to start the channel too:

```
port.addEventListener("message", (event) => {...});
port.start();
```

To attach a port to the `postMessage` call, use the second argument: `postMessage(data, [channel.port2])`. To use this port on the other end, use `event.ports[0].postMessage()`.

By creating a new MessageChannel for each request, pairing the response is solved as whatever comes through that channel is the response to this particular request.

Error handling

An often overlooked aspect of request-response communication is error handling. When the receiving end has a problem and throws an Error, it should be propagated to the sender so that it can handle it appropriately.

The problem with `postMessage` is that it can only send one type of message, there is no `postError` call. How to signal an error then?

A possible solution is similar to how Node-style callbacks propagate errors using only a single function. These callbacks use an error and a result value together, and only one of them is defined:

```
(err, result) => {
  if (err) {
    // error
  } else {
    // result
  }
}
```

To implement the same with messages, use an object with `error` and `result` properties. When the former is non-null that indicates that an error happened.

```
// worker.js
try{
  event.ports[0].postMessage(
    {result: event.data[0] + event.data[1]}
  );
}catch(e) {
  event.ports[0].postMessage({error: e});
}

// index.html
channel.port1.onmessage = ({data}) => {
  if (data.error) {
    // error
  }else {
    // data.result
  }
};
```

Don't forget to wrap the receiving end in a try-catch to propagate runtime errors.

Using Promises

With a separated response channel and error propagation, it's easy to wrap the call in a Promise constructor.

```
const worker = new Worker("worker.js");

const add = (a, b) => new Promise((res, rej) => {
  const channel = new MessageChannel();

  channel.port1.onmessage = ({data}) => {
    channel.port1.close();
    if (data.error) {
      rej(data.error);
    }else {
      res(data.result);
    }
  };

  worker.postMessage([a, b], [channel.port2]);
});

console.log(await add(3, 5)); // 8
```

With a Promise hiding all the complexities of the remote call, everything that works with async/await works with these calls too:

```
// parallel execution
console.log(await Promise.all([
  add(1, 1),
  add(5, 5),
])); // [2, 10]

// async reduce
console.log(await [1, 2, 3, 4].reduce(async (memo, i) => {
  return add(await memo, i);
}), 0); // 10
```

And the Promise rejects when it should, allowing proper error handling on the sending end:

```
// worker.js
addEventListener("message", (event) => {
  try{
    if (
      typeof event.data[0] !== "number" ||
      typeof event.data[1] !== "number"
    ) {
      throw new Error("both arguments must be numbers");
    }
    event.ports[0].postMessage(
      {result: event.data[0] + event.data[1]}
    );
  }catch(e) {
    event.ports[0].postMessage({error: e});
  }
}, false)

// index.html
try {
  await add(undefined, "b");
}catch(e) {
  console.log("Error: " + e.message); // error
}
```

One restriction is what can be sent through the `postMessage` call. It uses the *structured clone algorithm* which supports complex objects but not everything.

Collection processing with async functions

While async/await is great to make async *commands* look like synchronous ones, collection processing is not that simple. It's not just adding an `async` before the function passed to `Array.reduce` and it will magically work correctly. But without async functions, you can not use `await` and provide a result *later* which is required for things like reading a database, making network connections, reading files, and a whole bunch of other things.

Let's see some examples!

When all the functions you need to use are synchronous, it's easy. For example, a string can be doubled without any asynchronicity involved:

```
const double = (str) => {
  return str + str;
};

double("abc");
// abcabc
```

If the function is async, it's also easy for a single item using `await`. For example, to calculate the SHA-256 hash, Javascript provides a `digest()` async function:

```
// https://developer.mozilla.org/docs/Web/API/SubtleCrypto/digest
const digestMessage = async (message) => {
  const msgUint8 = new TextEncoder().encode(message);
  const hashBuffer =
    await crypto.subtle.digest("SHA-256", msgUint8);
  const hashArray = Array.from(new Uint8Array(hashBuffer));
  const hashHex = hashArray
    .map(b => b.toString(16).padStart(2, "0"))
    .join('');
  return hashHex;
};

await digestMessage("msg");
// e46b320165eec91e6344fa1034...
```

This looks almost identical to the previous call, the only difference is an `await`.

But for collections, handling asynchronicity becomes different.

To calculate the double for each element in a **collection of strings** is simple with a `map`:

```
const strings = ["msg1", "msg2", "msg3"];

strings.map(double);
// ["msg1msg1", "msg2msg2", "msg3msg3"]
```

But to calculate the **hash** of each string in a collection, it does not work:

```
const strings = ["msg1", "msg2", "msg3"];

await strings.map(digestMessage);
// [object Promise],[object Promise],[object Promise]
```

This is an everyday problem as async functions make a bigger category than synchronous ones. A sync function can work in an async environment, but an async one can not be converted to work in a synchronous way.

For a more realistic scenario, querying a database is inherently async as it needs to make network connections. To get the score for a single `userId` from a hypothetical database is simple:

```
const getUserObject = async (id) => {
  // get user by id
};

const userId = 15;
const userObject = await getUserObject(userId);
const userScore = userObject.score;
```

But how to query the database for a *collection* of `userId`s?

```
const userIds = [15, 16, 21];
// const userObjects = ???
```

And it's not just about using `map` to transform one value to another. Is the user's score above 3?

```
const userObject = await getUserObject(userId);
const above3 = userObject.score > 3;
```

But is *any* of the users' score above 3?

```
// userIds.some(???)
```

Or what is the average score?

```
// userIds.reduce(???)
```

To make things more interesting, adding an `async` to a `map` at least gives some indication what is changed:

```
// synchronous
[1, 2, 3].map((i) => {
  return i + 1;
});
// [2, 3, 4]

// asynchronous
[1, 2, 3].map(async (i) => {
  return i + 1;
});
// [object Promise],[object Promise],[object Promise]
```

But a `filter` just does something entirely wrong:

```
// synchronous
[1, 2, 3, 4, 5].filter((i) => {
  return i % 2 === 0;
});
// [2, 4]

// asynchronous
[1, 2, 3, 4, 5].filter(async (i) => {
  return i % 2 === 0;
});
// [1, 2, 3, 4, 5]
```

Because of this, async collection processing requires some effort, and it's different depending on what kind of function you want to use. An async `map` works markedly differently than an async `filter` or and async `reduce`.

In this chapter, you'll learn how each of them works and how to efficiently use them.

Async for iteration

But first, let's talk about `for` loops!

I don't like `for` loops as they tend to promote bad coding practices, like nested loops that do a lot of things at once or `continue`/`break` statements scattered around that quickly descend into an unmaintainable mess.

Also, a more functional approach with functions like `map`/`filter`/`reduce` promote a style where one function does only one thing and everything inside it is scoped and stateless. And the functional approach is not only possible 99% of the time but it comes with no perceivable performance drop and it also yields simpler code (well, at least when you know the functions involved).

But async/await is in the remaining 1%.

For loops have a distinctive feature, as they don't rely on *calling* a function. In effect, you can use `await` inside the loop and it will just work.

```
// synchronous
{
  const res = [];
  for (let i of [1, 2, 3]){
    res.push(i + 1);
  }
  // res: [2, 3, 4]
}

// asynchronous
{
  const res = [];
  for (let i of [1, 2, 3]){
    await sleep(10);
    res.push(i + 1);
  }
  // res: [2, 3, 4]
}
```

As `for` loops are a generic tool, they can be used for all kinds of requirements when working with collections.

But their downside is still present, and while it's not trivial to adapt the functional approach to async/await, once you start seeing the general pattern it's not that hard either.

Async functions with reduce

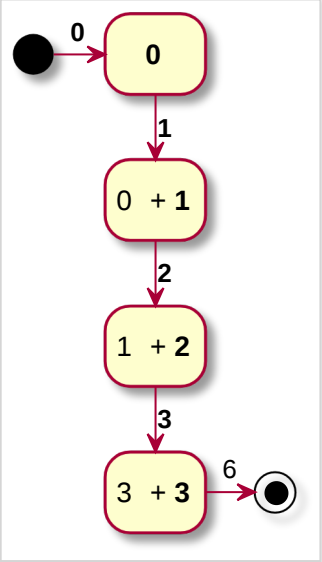
The `reduce` function iteratively constructs a value and returns it, which is not necessarily a collection. That's where the name comes from, as it *reduces* a collection to a value.

The iteratee function gets the previous result, called `memo` in the examples below, and the current value, `e`.

The following function sums the elements, starting with 0 (the second argument of `reduce`):

```
const arr = [1, 2, 3];

const syncRes = arr.reduce((memo, e) => {
  return memo + e;
}, 0);

console.log(syncRes);
// 6
```

`memo`	`e`	**result**
0 (initial)	**1**	1
1	**2**	3
3	**3**	(end result) 6



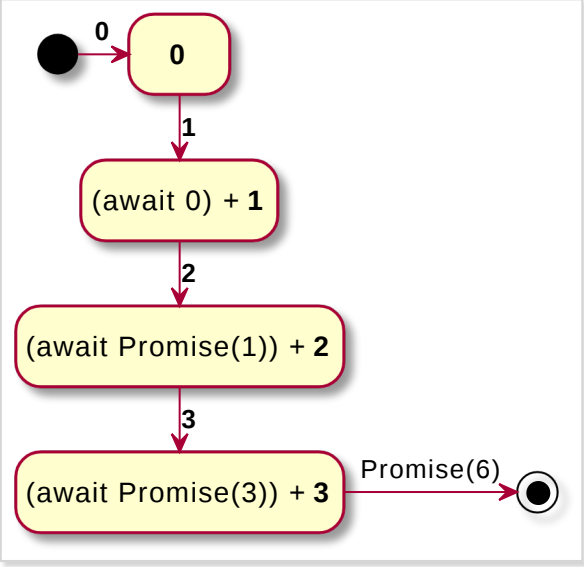
The reduce function

Asynchronous reduce

An async version is almost the same, but it returns a Promise on each iteration, so `memo` will be the *Promise* of the previous result. The iteratee function needs to `await` it in order to calculate the next result:

```
// utility function for sleeping
const sleep = (n) => new Promise((res) => setTimeout(res, n));

const arr = [1, 2, 3];

const asyncRes = await arr.reduce(async (memo, e) => {
  await sleep(10);
  return (await memo) + e;
}, 0);

console.log(asyncRes);
// 6
```

`memo`	`e`	**result**
0 (initial)	**1**	Promise(1)
Promise(1)	**2**	Promise(3)
Promise(3)	**3**	(end result) Promise(6)



Async reduce function

With the structure of `async (memo, e) => await memo`, the `reduce` can handle any async functions and it can be `await`ed.

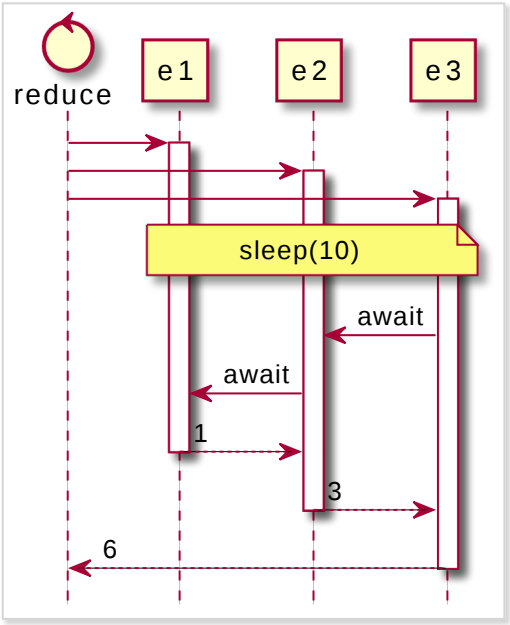
Timing

Concurrency has an interesting property when it comes to `reduce`. In the synchronous version, elements are processed one-by-one, which is not surprising as they rely on the *previous* result. But when an async `reduce` is run, all the iteratee functions start running in parallel and wait for the previous result (`await memo`) only when needed.

await memo last

In the example above, all the `sleep`s happen in parallel, as the `await memo`, which makes the function to wait for the previous one to finish, comes later.

```
const arr = [1, 2, 3];

const startTime = new Date().getTime();

const asyncRes = await arr.reduce(async (memo, e) => {
  await sleep(10);
  return (await memo) + e;
}, 0);

console.log(`Took ${new Date().getTime() - startTime} ms`);
// Took 11-13 ms
```

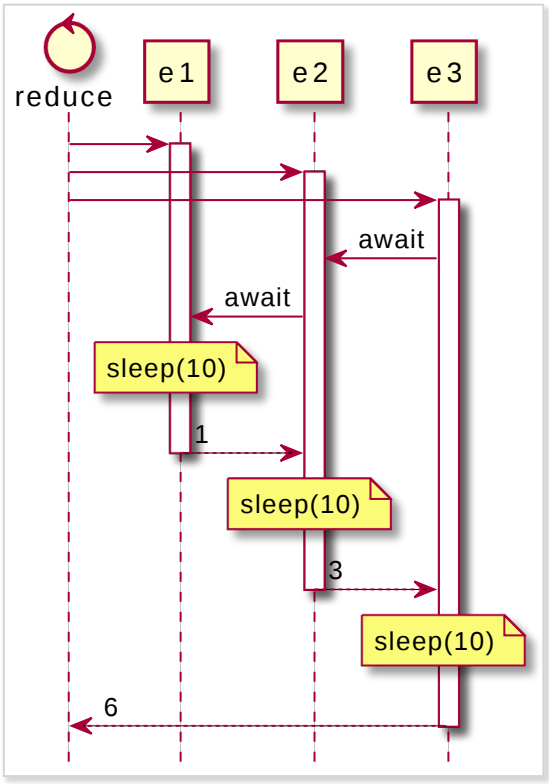


Async reduce with "await memo" last

await memo first

But when the `await memo` comes first, the functions run sequentially:

```
const arr = [1, 2, 3];

const startTime = new Date().getTime();

const asyncRes = await arr.reduce(async (memo, e) => {
  await memo;
  await sleep(10);
  return (await memo) + e;
}, 0);

console.log(`Took ${new Date().getTime() - startTime} ms`);
// Took 36-38 ms
```



Async reduce with "await memo" first

This behavior is usually not a problem as it naturally means everything that is not dependent on the previous result will be calculated immediately, and only the dependent parts are waiting for the previous value.

When parallelism matters

But in some cases, it might be unfeasible to do something ahead of time.

For example, I had a piece of code that prints different PDFs and concatenates them into one single file using the pdf-lib library.

This implementation runs the resource-intensive `printPDF` function in parallel:

```
const result = await printingPages.reduce(async (memo, page) => {
  const pdf = await PDFDocument.load(await printPDF(page));

  const pdfDoc = await memo;

  (await pdfDoc.copyPages(pdf, pdf.getPageIndices()))
    .forEach((page) => pdfDoc.addPage(page));

  return pdfDoc;

}, PDFDocument.create());
```

I noticed that when I have many pages to print, it would consume too much memory and slow down the overall process.

A simple change made the `printPDF` calls wait for the previous one to finish:

```
const result = await printingPages.reduce(async (memo, page) => {
  const pdfDoc = await memo;

  const pdf = await PDFDocument.load(await printPDF(page));

  (await pdfDoc.copyPages(pdf, pdf.getPageIndices()))
    .forEach((page) => pdfDoc.addPage(page));

  return pdfDoc;

}, PDFDocument.create());
```

Async functions with map

The `map` is the easiest and most common collection function. It runs each element through an iteratee function and returns an array with the results.

The synchronous version that adds one to each element:

```
const arr = [1, 2, 3];

const syncRes = arr.map((i) => {
  return i + 1;
});

console.log(syncRes);
// 2,3,4
```

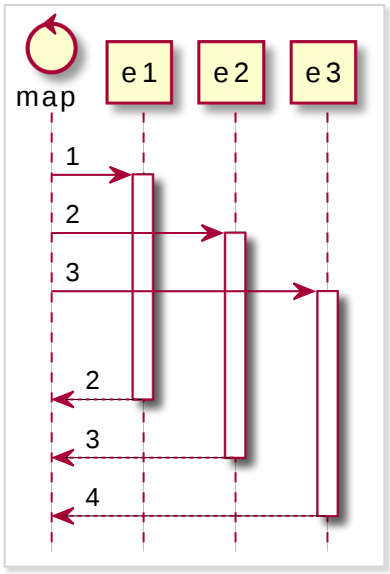
An async version needs to do two things. First, it needs to **map every item to a *Promise*** with the new value, which is what adding `async` before the function does.

And second, it needs to **wait for all the Promises** then **collect the results** in an Array. Fortunately, the `Promise.all` built-in call is exactly what we need for step 2.

This makes the general pattern of an async `map` to be `Promise.all(arr.map(async (...) => ...))`.

An async implementation doing the same as the sync one:

```
const arr = [1, 2, 3];

const asyncRes = await Promise.all(arr.map(async (i) => {
  await sleep(10);
  return i + 1;
}));

console.log(asyncRes);
// 2,3,4
```



Async map

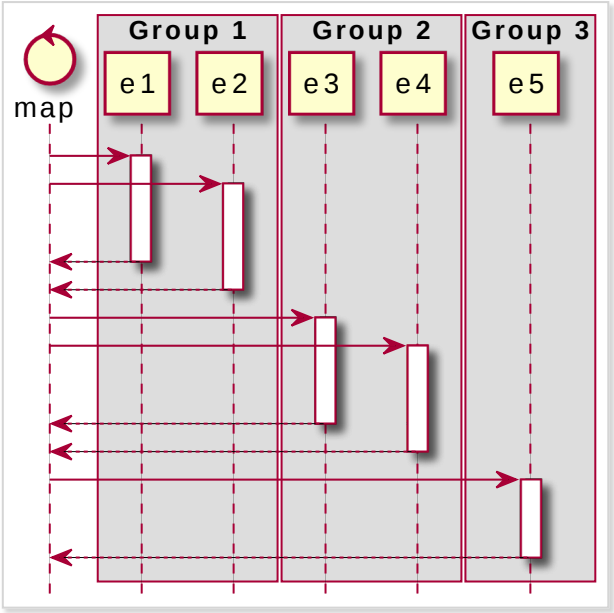
Concurrency

The above implementation runs the iteratee function in parallel for each element of the array. This is usually fine, but in some cases, it might consume too much resources. This can happen when the async function hits an API or consumes too much RAM that it's not feasible to run too many at once.

While an async `map` is easy to write, adding concurrency controls is more involved. In the next few examples, we'll look into different solutions.

Batch processing

The easiest way is to group elements and process the groups one by one. This gives you control of the maximum amount of parallel tasks that can run at once. But since one group has to finish before the next one starts, the *slowest* element in each group becomes the limiting factor.



Mapping in groups

To make groups, the example below uses the `groupBy` implementation from Underscore.js. Many libraries provide an implementation and they are mostly interchangeable. The exception is Lodash, as its groupBy does not pass the index of the item.

If you are not familiar with `groupBy`, it runs each element through an iteratee function and returns an object with the keys as the results and the values as lists of the elements that produced that value.

To make groups of at most `n` elements, an iteratee `Math.floor(i / n)` where `i` is the *index* of the element will do. As an example, a group of size 3 will map the elements like this:

```
0 => 0
1 => 0
2 => 0
3 => 1
4 => 1
5 => 1
6 => 2
...
```

In Javascript:

```
const arr = [30, 10, 20, 20, 15, 20, 10];

console.log(
  _.groupBy(arr, (_v, i) => Math.floor(i / 3))
);
// {
//     0: [30, 10, 20],
//     1: [20, 15, 20],
//     2: [10]
// }
```

The last group might be smaller than the others, but all groups are guaranteed not to exceed the maximum group size.

To map one group, the usual `Promise.all(group.map(...))` construct is fine.

To map the groups sequentially, we need a reduce that concatenates the previous results (`memo`) with the results of the current group:

```
return Object.values(groups)
  .reduce(async (memo, group) => [
    ...(await memo),
    ...(await Promise.all(group.map(iteratee)))
  ], []);
```

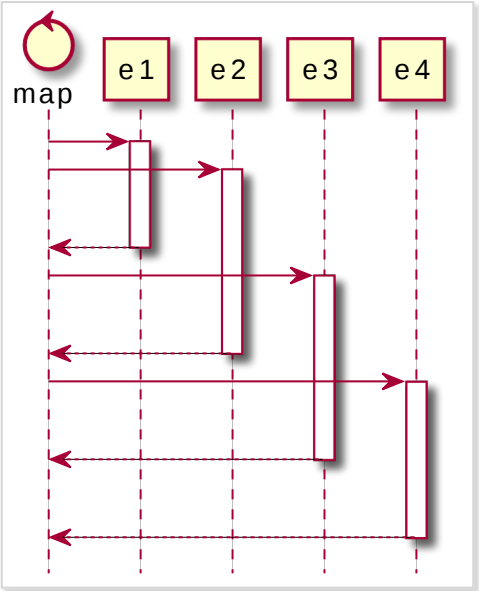
This implementation is based on the fact that the `await memo`, which waits for the previous result, will be completed before moving on to the next line.

The full implementation that implements batching:

```
const arr = [30, 10, 20, 20, 15, 20, 10];

const mapInGroups = (arr, iteratee, groupSize) => {
  const groups = _.groupBy(arr, (_v, i) =>
    Math.floor(i / groupSize));

  return Object.values(groups)
    .reduce(async (memo, group) => [
      ...(await memo),
      ...(await Promise.all(group.map(iteratee)))
    ], []);
};

const res = await mapInGroups(arr, async (v) => {
  console.log(`S ${v}`);
  await sleep(v);
  console.log(`F ${v}`);
  return v + 1;
}, 3);

// -- first batch --
// S 30
// S 10
// S 20
// F 10
// F 20
// F 30
// -- second batch --
// S 20
// S 15
// S 20
// F 15
// F 20
// F 20
// -- third batch --
// S 10
// F 10

console.log(res);
// 31,11,21,21,16,21,11
```

Parallel processing

Another type of concurrency control is to run at most `n` tasks in parallel, and start a new one whenever one is finished.



Controlled concurrency

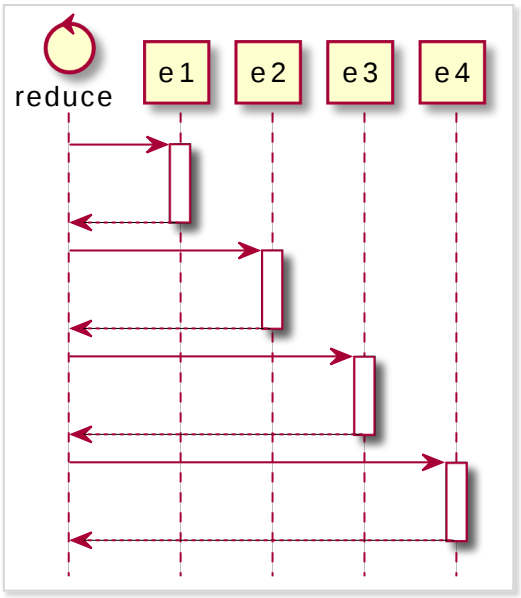
I couldn't come up with a simple implementation for this, but fortunately, the Bluebird library provides one out of the box. This makes it straightforward, just import the library and use the `Promise.map` function that has support for the `concurrency` option.

In the example below the concurrency is limited to `2`, which means 2 tasks are started immediately, then whenever one is finished, a new one is started until there are none left:

```
const arr = [30, 10, 20, 20, 15, 20, 10];

// Bluebird promise
const res = await Promise.map(arr, async (v) => {
  console.log(`S ${v}`)
  await sleep(v);
  console.log(`F ${v}`);
  return v + 1;
}, {concurrency: 2});

// S 30
// S 10
// F 10
// S 10
// F 30
// S 20
// F 10
// S 15
// F 20
// S 20
// F 15
// S 20
// F 20
// F 20

console.log(res);
// 31,11,21,21,16,21,11
```

Sequential processing

Sometimes any concurrency is too much and the elements should be processed one after the other.



Mapping sequentially

A trivial implementation is to use Bluebird's Promise with a concurrency of 1. But for this case, it does not warrant including a library as a simple `reduce` would do the job:

```
const arr = [1, 2, 3];

const res = await arr.reduce(async (memo, v) => {
  const results = await memo;
  console.log(`S ${v}`)
  await sleep(10);
  console.log(`F ${v}`);
  return [...results, v + 1];
}, []);

// S 1
// F 1
// S 2
// F 2
// S 3
// F 3

console.log(res);
// 2,3,4
```

Make sure to `await` the memo before `await`-ing anything else, as without that it will still run concurrently!

Async functions with forEach

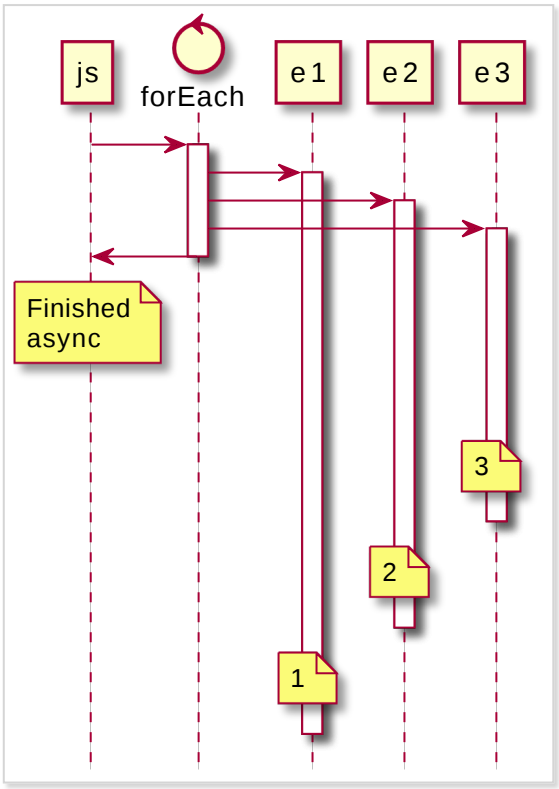
The `forEach` function is similar to the `map`, but instead of transforming the values and using the results, it runs the function for each element and discards the result. Effectively, the important part is the *side effects* of calling the function.

For example, printing each element to the console, synchronously:

```
const arr = [1, 2, 3];

arr.forEach((i) => {
  console.log(i);
});

// 1
// 2
// 3

console.log("Finished sync");
// Finished sync
```

As the result is not important, using an async function as the iteratee would work:

```
const arr = [1, 2, 3];

arr.forEach(async (i) => {
  // each element takes a different amount of time to complete
  await sleep(10 - i);
  console.log(i);
});

console.log("Finished async");
// Finished async

// 3
// 2
// 1
```



Async forEach

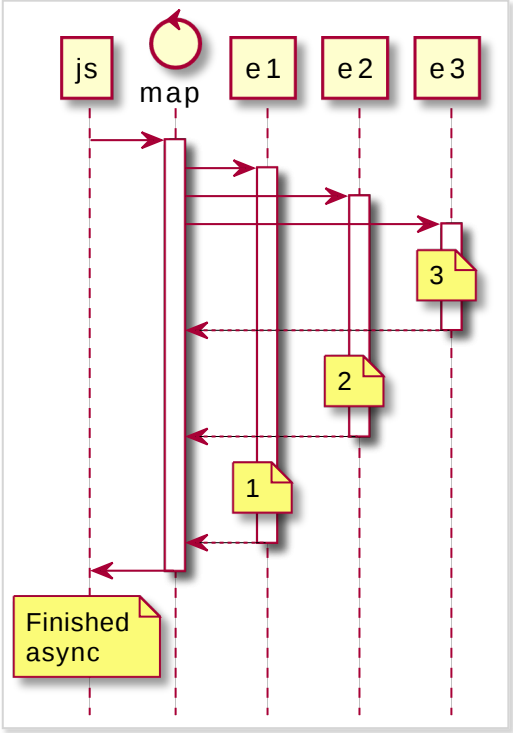
Controlling the timing

Waiting for finish

But, not unsurprisingly, the function is called **asynchronously**, and the program execution goes past the call. This is an important difference from the sync version, as, by the time the next line is executed, the synchronous `forEach` is already done, while the async version is not. That's why the "Finished async" log appears *before* the elements.

To wait for all the function calls to finish before moving on, use a `map` with a `Promise.all` and discard the results:

```
const arr = [1, 2, 3];

await Promise.all(arr.map(async (i) => {
  await sleep(10 - i);
  console.log(i);
}));

// 3
// 2
// 1

console.log("Finished async");
// Finished async
```



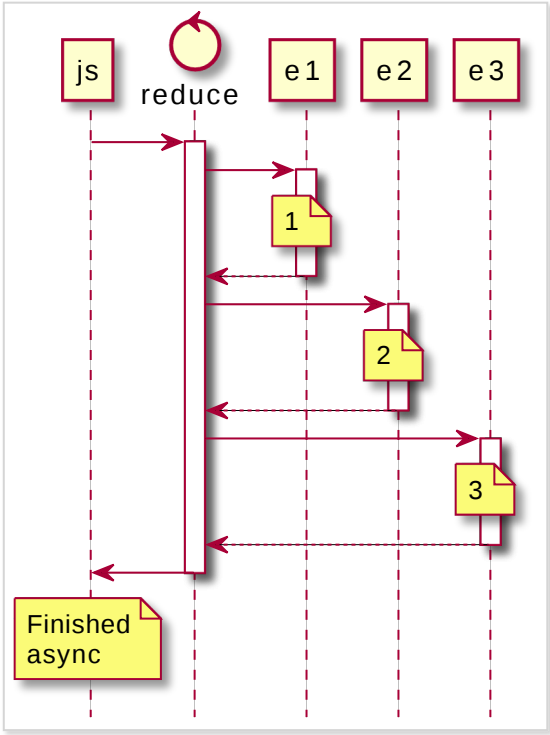
Async forEach, waiting for the results

With this change, the "Finished async" comes last.

Sequential processing

But notice that the iteratee functions are called in parallel. To faithfully follow the synchronous `forEach`, use a `reduce` with an `await memo` first:

```
const arr = [1, 2, 3];

await arr.reduce(async (memo, i) => {
  await memo;
  await sleep(10 - i);
  console.log(i);
}, undefined);

// 1
// 2
// 3

console.log("Finished async");
// Finished async
```



Async forEach, sequential processing

This way the elements are processed in-order, one after the other, and the program execution waits for the whole array to finish before moving on.

Async functions with filter

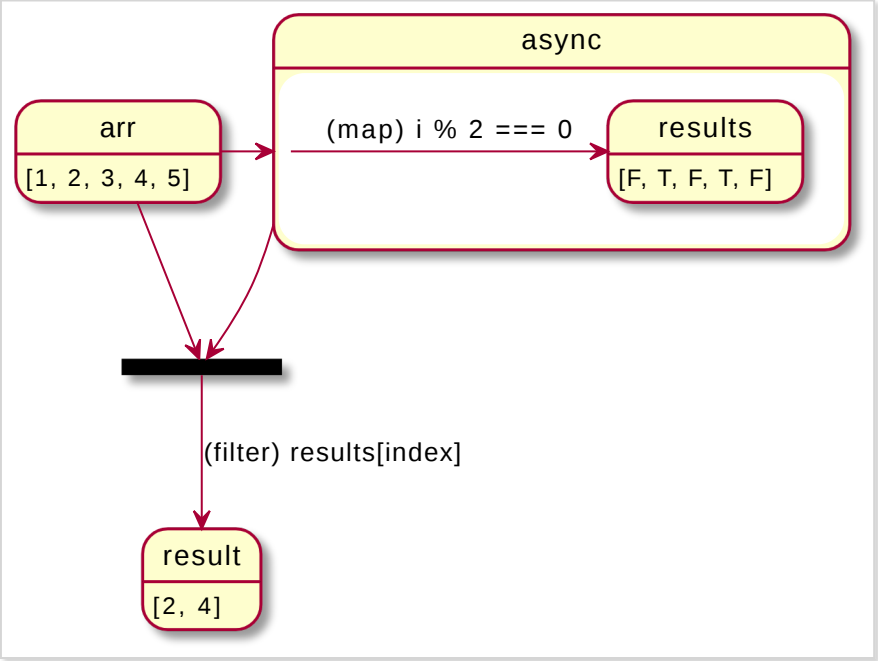
The `filter` function keeps only the elements that pass a condition. It gets a function, this time it's called a *predicate*, and this function returns true/false (truthy and falsy, to be more precise) values. The resulting collection only

contains the elements where the predicate returned true.

```
const arr = [1, 2, 3, 4, 5];

const syncRes = arr.filter((i) => {
  return i % 2 === 0;
});

console.log(syncRes);
// 2,4
```

Async filter with map

The async version is a bit more complicated this time and it works in two phases. The first one maps the array through the predicate function asynchronously, producing true/false values. Then the second step is a synchronous `filter` that uses the results from the first step.

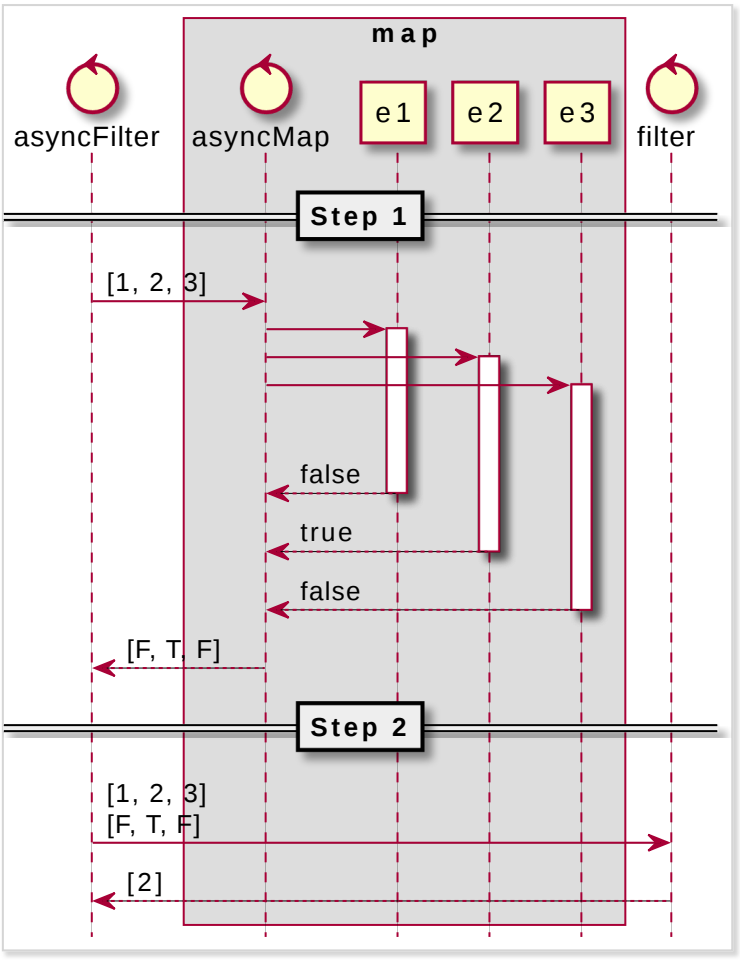


Async filter

```
const arr = [1, 2, 3, 4, 5];

const asyncFilter = async (arr, predicate) => {
  const results = await Promise.all(arr.map(predicate));

  return arr.filter((_v, index) => results[index]);
}

const asyncRes = await asyncFilter(arr, async (i) => {
  await sleep(10);
  return i % 2 === 0;
});

console.log(asyncRes);
// 2,4
```

Or a one-liner implementation:

```
const asyncFilter = async (arr, predicate) =>
  Promise.all(arr.map(predicate))
  .then((results) => arr.filter((_v, index) => results[index]));
```



Async filter with map

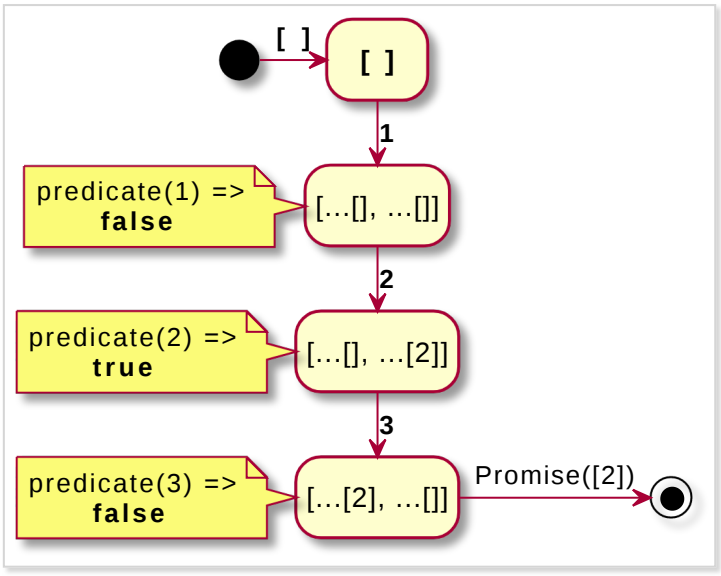
Concurrency

The above implementation runs all of the predicate functions concurrently. This is usually fine, but, as with all other functions, it might strain some resources too hard. Fortunately, since the above implementation relies on `map`, the same concurrency controls can be used.

Async filter with reduce

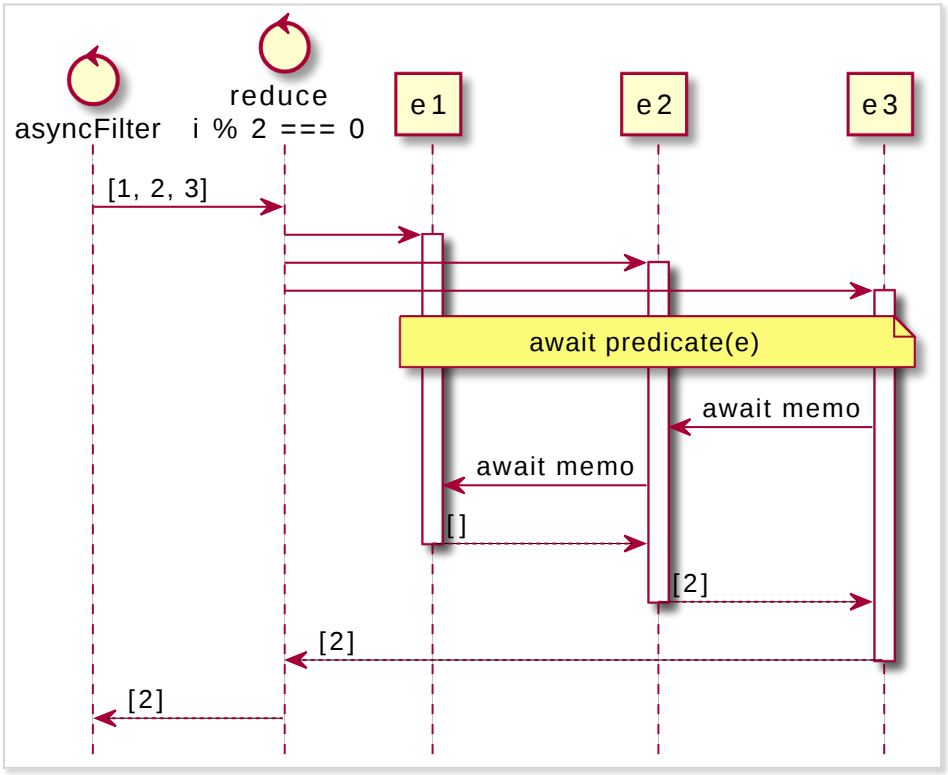
Instead of using an async `map` with a sync `filter`, an async `reduce` could also do the job. Since it's just one function, the structure is even easier though it does not provide the same level of control.

First, start with an empty array (`[]`). Then run the next element through the predicate function and if it passes, append it to the array. If not, skip it.



Async filter with reduce

```
// concurrently
const asyncFilter = async (arr, predicate) =>
  arr.reduce(async (memo, e) =>
    await predicate(e) ? [...await memo, e] : memo
  , []);
```



Async filter with reduce

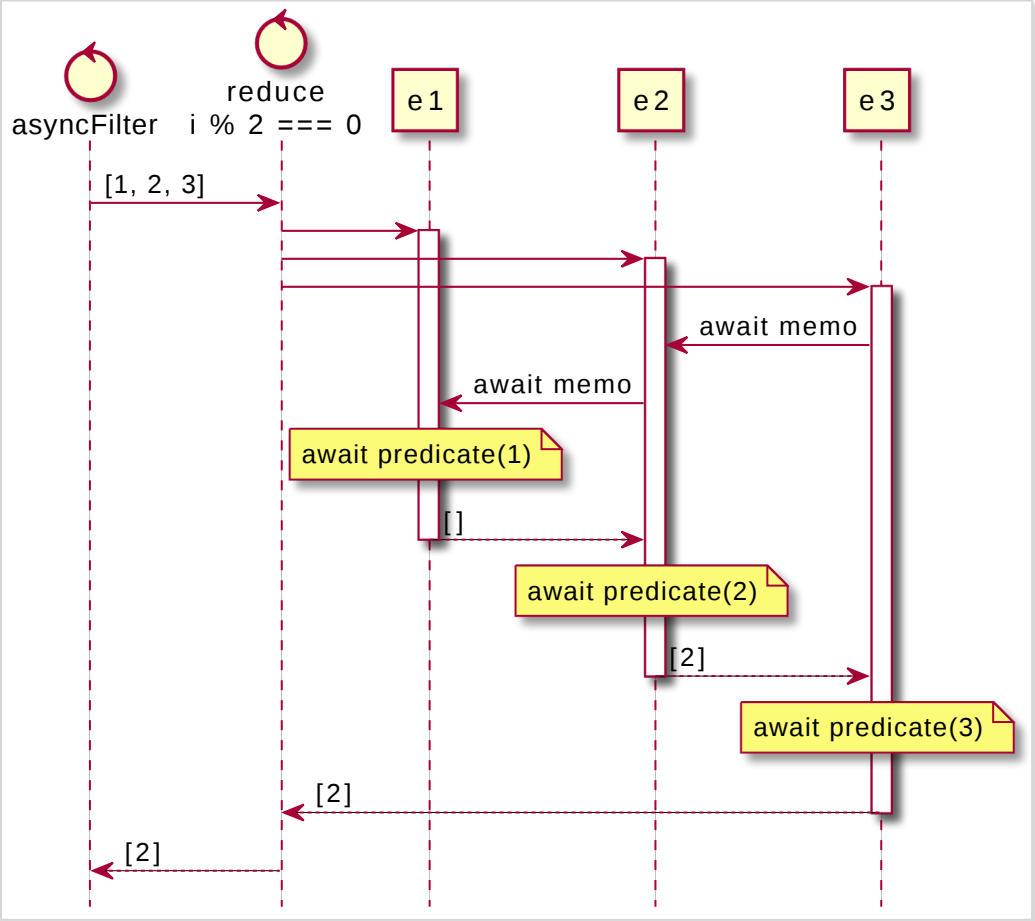
Notice that the `await predicate(e)` comes *before* the `await memo`, which means those will be called in parallel.

Sequential processing

To wait for a predicate function to finish before calling the next one, change the order of the `await`s:

```
// sequentially
const asyncFilter = async (arr, predicate) =>
  arr.reduce(async (memo, e) =>
    [...await memo, ...await predicate(e) ? [e] : []]
  , []);
```

This implementation waits for the previous element then conditionally append an element depending on the result of the predicate (`...[e]` or `...[]`).



Async filter with reduce running sequentially

Async functions with some/every

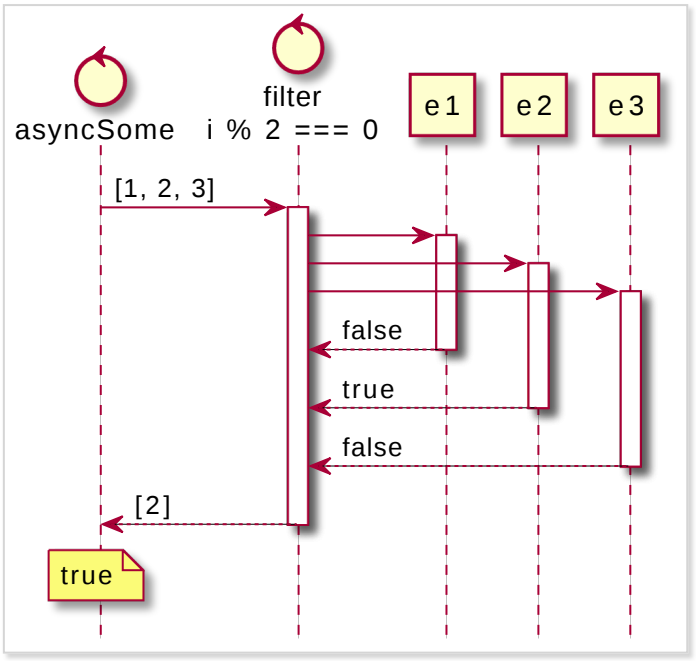
These functions get an iteratee function, just like the `filter`, but they return a single true/false, depending on whether the predicate returned a specific value. In the case of the `some`, if *any* of the predicates returned true, the result will be true. For the `every` function, if any returned false, the result will be false.

```
const arr = [1, 2, 3];

const someRes = arr.some((i) => {
  return i % 2 === 0;
});

console.log(someRes);
// true

const everyRes = arr.every((i) => {
  return i < 2;
});

console.log(everyRes);
// false
```

Using an async filter

Considering only the result, these functions can be emulated with an async `filter`, which is already covered in a previous article how to convert to async.

```
// sync
const some = (arr, predicate) =>
  arr.filter(predicate).length > 0;
const every = (arr, predicate) =>
  arr.filter(predicate).length === arr.length;

// async
const asyncSome = async (arr, predicate) =>
  (await asyncFilter(arr, predicate)).length > 0;
const asyncEvery = async (arr, predicate) =>
  (await asyncFilter(arr, predicate)).length === arr.length;
```

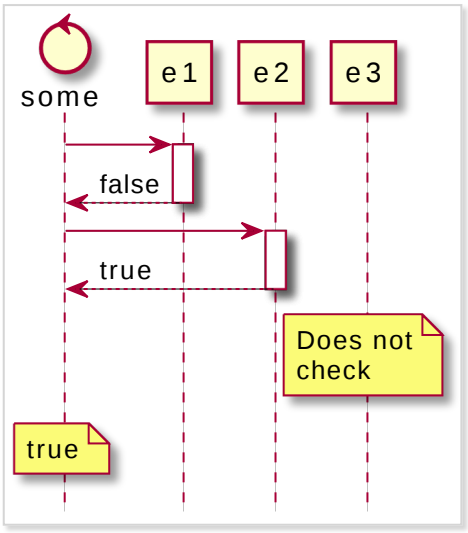


Filter-based async some

Short-circuiting

But there is an important difference between the built-in `some`/`every` functions and the `filter`-based implementations. When there is an element that returns true for a `some`, it **short-circuits** and does not process the remaining elements:

```
const arr = [1, 2, 3];

const res = arr.some((i) => {
  console.log(`Checking ${i}`);
  return i % 2 === 0;
});

// Checking 1
// Checking 2

console.log(res);
// true
```



Synchronous some

Similarly, `every` stops after the first false result:

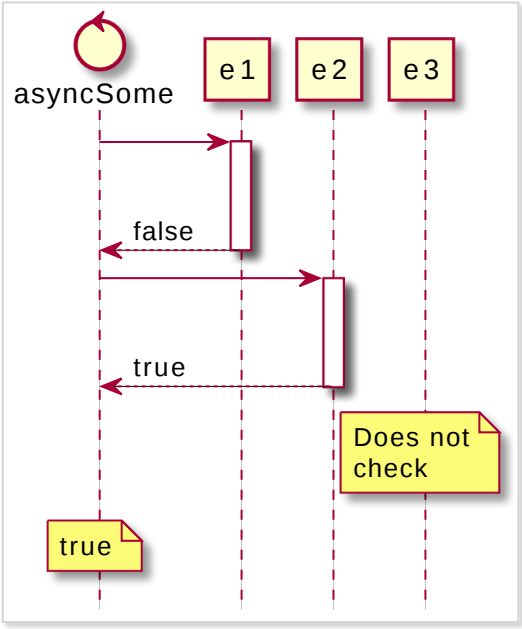
```
const arr = [1, 2, 3];

const res = arr.every((i) => {
  console.log(`Checking ${i}`);
  return i < 2;
});

// Checking 1
// Checking 2

console.log(res);
// false
```

Let's see how to code an async version that works in a similar way and does the least amount of work!

Async some

The best solution is to use an async **for iteration** that returns as soon as it finds a truthy result:

```
const arr = [1, 2, 3];

const asyncSome = async (arr, predicate) => {
  for (let e of arr) {
    if (await predicate(e)) return true;
  }
  return false;
};
const res = await asyncSome(arr, async (i) => {
  console.log(`Checking ${i}`);
  await sleep(10);
  return i % 2 === 0;
});

// Checking 1
// Checking 2

console.log(res);
// true
```



For-based async some

For the first element `predicate(e)` returns true, it concludes the for-loop.

Async every

The similar structure works for `every`, it's just a matter of negating the conditions:

```
const arr = [1, 2, 3];

const asyncEvery = async (arr, predicate) => {
  for (let e of arr) {
    if (!await predicate(e)) return false;
  }
  return true;
};
const res = await asyncEvery(arr, async (i) => {
  console.log(`Checking ${i}`);
  await sleep(10);
  return i < 2;
});

// Checking 1
// Checking 2

console.log(res);
// false
```

Whenever there is a false value returned by `predicate(e)`, the function is ended without checking the other elements.

Parallel processing

The short-circuiting implementation processes the elements sequentially, which is efficient in terms of resource usage, but it might result in longer execution.

For example, if the iteratee sends requests via a network, it might take some time to send them one at a time. On the other hand, while it might result in more requests sent, sending all of them at the same time would be faster.

Common errors

Not propagating errors

When you convert a callback to a Promise, make sure that you also handle the error case. Without that, the Promise is never resolved and an `await` waits forever.

For example, the `gapi.load` provides separate callbacks for results and errors:

```
await new Promise((res, rej) => {
  gapi.load("client:auth2", {callback: res, onerror: rej});
});
```

If you don't attach the `rej` callback to the `onerror` handler, errors won't propagate through the chain. This leads to hard-to-debug problems and components that "just stops".

Missing await in try..catch

When you use `try..catch` or `try..finally` in an async function, make sure that you use `await` too. For example, this code works fine when the inner Promise resolves:

```
const fn = async () => {
  return "result";
}

const outer = async () => {
  try {
    return fn(); // <= missing await!
  }catch(e) {
    console.log(e.message);
  }
}

await outer(); // result
```

But the `try..catch` won't handle exceptions:

```
const fn = async () => {
  throw new Error("error");
}

const outer = async () => {
  try {
    return fn(); // <= missing await!
  }catch(e) {
    console.log(e.message);
  }
}

await outer();
// exception is thrown!
```

This is because exceptions are thrown from the `await` and while returning a Promise from an async function flattens the result, it will be returned without waiting for it. And without waiting, an exception is not thrown for the `try..catch` but when the caller uses `await`.

```
const fn = async () => {
  throw new Error("error");
}

const outer = async () => {
  try {
    return await fn();
  }catch(e) {
    console.log(e.message); // error
  }
}

await outer();
// no exception here
```

Not checking the return value

When a Promise is resolved it does not necessarily mean that everything is alright. For example, the `fetch` rejects when there is a problem *with the connection* but it resolves even if the response status returns an error.

When the call can not establish a connection, the Promise rejects:

```
try {
  await fetch("htp://example.com");
}catch(e) {
  console.log(e.message); // Failed to fetch
}
```

But when the response status code indicates an error (500), the Promise still resolves:

```
try {
  await fetch("https://httpbin.org/status/500");

  // no error
}catch(e) {
  console.log(e.message);
}
```

Because of this, you need to check the result for error responses. The `fetch` result provides the `ok` field to do this:

```
const response = await fetch("https://httpbin.org/status/500");

if (!response.ok) {
  throw new Error("Error response");
}
```

Not closing resources in case of a rejection

It's common to open some resource, then close it after it's not needed. For example, this code creates a temporary directory then deletes it later:

```
const fs = require("fs").promises;
const os = require("os");
const path = require("path");

const dir = await fs.mkdtemp(
  await fs.realpath(os.tmpdir()) + path.sep
);

// use dir to store temporary things

await fs.rmdir(dir, {recursive: true});
```

This works unless there is an exception that jumps over the cleanup code.

```
const dir = await fs.mkdtemp(
  await fs.realpath(os.tmpdir()) + path.sep
);

throw Error("error");

await fs.rmdir(dir, {recursive: true});
// the cleanup won't run!
```

Use `try..finally` to implement reliable cleanup:

```
const dir = await fs.mkdtemp(
  await fs.realpath(os.tmpdir()) + path.sep
);
try {
  // use dir to store temporary things
}finally {
  await fs.rmdir(dir, {recursive: true});
}
```

Even better, you can move the resource lifecycle to a dedicated function, as discussed in the The async disposer pattern chapter:

```
const withTempDir = async (fn) => {
  const dir = await fs.mkdtemp(
    await fs.realpath(os.tmpdir()) + path.sep
  );
  try {
    return await fn(dir);
  }finally {
    fs.rmdir(dir, {recursive: true});
  }
};

await withTempDir((dir) => {
  // use dir to store temporary things
})
```

Using both the rejection and resolve handlers in one then function

A common source of problems is to write error handling like this:

```
getPromise()
  .then(process, handleError);
```

As we've discussed in the Error propagation in Promise chains chapter, the second handler of the `then()` is called for errors but only from the previous steps.

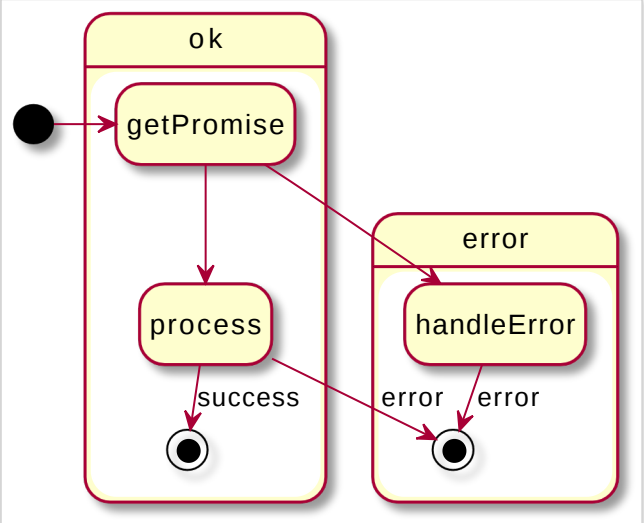


Diagram of the promise chain

Notice that the `handleError` function only gets errors from the `getPromise`. If the result of the `process` function is rejected, it won't be handled.

A better solution is to use `.catch` to be explicit about errors:

```
getPromise()
  .then(process)
  .catch(handleError);
```

This way it's clear that both steps are covered by the error handlers.

Not waiting for an async forEach

The `forEach` function iterates over a collection and calls an iteratee function. A typical problem is not waiting for the iteration to complete:

```
[1, 2, 3].forEach(async (e) => {
  console.log(e);
  return makePromise(e);
});
console.log("finished");
// finished
// 1
// 2
// 3
```

It is a best practice to wait until the whole collection is processed, which can be achived with a `Promise.all` and a `map`:

```
await Promise.all([1, 2, 3].map(async (e) => {
  console.log(e);
  return makePromsie(e);
}));
console.log("finished");
// 1
// 2
// 3
// finished
```

> **More info**
>
> This topic is covered in more detail in the Async functions with forEach chapter.

Glossary

Async function

A function that returns a Promise. The `await` keyword is only usable in an async function.

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

const fn = async () => {
  await wait(100);
  return "result";
}

fn(); // Promise
```

Async generator function

A generator function that can use `await`. When it encounters a `yield` or a `yield*` the execution stops until a subsequent element is requested.

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

const gen = async function*() {
  console.log("start");
  await wait(100);
  yield 1;
  await wait(100);
  console.log("next");
  yield 2;
  await wait(100);
  console.log("end");
};

const it = gen();
console.log(await it.next());
console.log(await it.next());
console.log(await it.next());

// start
// { value: 1, done: false }
// next
// { value: 2, done: false }
// end
// { value: undefined, done: true }
```

See Generator function.

await keyword

Usable in async functions, it stops the execution until the argument Promise becomes fulfilled. It then returns the value or throws an exception. See Async function.

Callback

A function passed as an argument that is called with the return value. It allows asynchronous results, as the callback can be invoked later.

For example, the `setTimeout` function needs a callback that it calls when the time is up:

```
setTimeout(() => {
  // 1 second later
}, 1000);
```

See Continuation-passing style.

Collection processing

A series of functions that work on a collection of elements (usually, an array), such as transforming (`map`), filtering (`filter`), and reducing to a value (`reduce`).

Continuation-passing style

A programming construct that uses callback functions (continuations) for function results instead of returning them directly. This allows asynchronous results as these callbacks can be invoked later.

```
// direct style
const directFn = () => {
  return "result";
};

// CPS
const cpsFn = (callback) => {
  callback("result");
};
```

See Callback.

Error propagation

When there is an error, it moves up in context until there is a piece of code that can handle it.

For example, an unhandled exception gets thrown to the caller:

```
const fn = () => {
  // the error is thrown here
  throw new Error("error");
}

try {
  fn();
}catch(e) {
  // and handled here
}
```

With Promises, errors propagate to the next handlers:

```
getUser() // returns a Promise
  .then(() => {
    // the error is thrown here
    throw new Error("error");
  })
  .catch((e) => {
    // and handled here
  });
```

Error-first callbacks

A callback style where the first argument is an error object. The receiving end needs to check if this value is non-null (or non-undefined).

```
const fn = (callback) => {
  try {
    const result = "result";
    // error is null
    callback(null, result);
  }catch(e) {
    // signal error
    callback(e);
  }
}

fn((err, result) => {
  if (err) {
    // handle error
  }else {
    // handle result
  }
});
```

See Node-style callbacks.

Fulfilled Promise

A state of a Promise that is either resolved or rejected. See Promise states.

Generator function

A function that returns an iterable structure. It can be frozen with a `yield` statement and only resumes if more elements are requested.

```
const gen = function*() {
  console.log("start");
  yield 1;
  console.log("next");
  yield 2;
  console.log("end");
}

const it = gen();
console.log(it.next());
console.log(it.next());
console.log(it.next());

// start
// { value: 1, done: false }
// next
// { value: 2, done: false }
// end
// { value: undefined, done: true }
```

Node-style callbacks

A callback style where:

- The callback function is the last argument
- It is called with an error object first, then a result (`(error, result)`)
- It returns only one result

```
const getUser = (id, cb) => {
  const user = /* get user */;
  if (user) {
    // success
    cb(null, user);
  }else {
    // error
    cb(new Error("Failed to get user"));
  }
};
```

Pagination

When a method (usually an API) does not return all results for a request and it requires multiple calls to get all items.

Pending Promise

A Promise state where the asynchronous result is not yet available. See Promise states.

postMessage

An API to communicate cross-context, such as with IFrames and Web Workers. It is an event-based communication where one end listens for `message` events and the other uses the `postMessage` function to send messages.

Promise

A value that holds an asynchronous result. You can use the `then` callback to handle when the Promise becomes fulfilled, or use `await` in an async function.

Promise chain

A series of `then()` callbacks, each getting the previous state of the Promise and returns the next one.

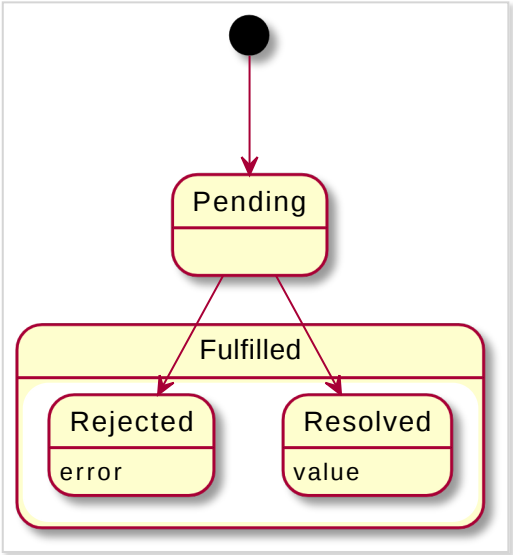
```
getUser() // returns a Promise
  .then(async function getPermissionsForUser(user) {
    const permissions = await // ...;
    return permissions;
  })
  .then(async function checkPermission(permissions) {
    const allowed = await // ...;
    return allowed;
  })
  .then((allowed) => {
    // handle allowed or not
  });
```

Promise states

A Promise can be in one of multiple states:

- Pending: The value is not yet available
- Resolved: The value is available
- Rejected: There was an error

When the Promise is either resolved or rejected then it's fulfilled or settled.



Promise states

Promise.all

A utility function that gets multiple Promises and resolves when all of them are resolved and returns an array with the results. It rejects if any of the input Promises rejects.

```
console.log(
  await Promise.all([
    Promise.resolve("a"),
    Promise.resolve("b"),
  ])
);

// ["a", "b"]
```

Promise.allSettled

A utility function that gets multiple Promises and resolves when all of them are settled. It resolves even if some input Promises reject.

```
console.log(
  await Promise.allSettled([
    Promise.resolve("a"),
    Promise.reject("b"),
  ])
);

// [
//     { status: 'fulfilled', value: 'a' },
//     { status: 'rejected', reason: 'b' }
// ]
```

Promise.catch

A utility function that attaches a rejection handler to a Promise. It's equivalent to `then(undefined, handler)`.

```
Promise.reject(new Error("error"))
  .catch((e) => {
    console.log(e.message); // error
  });
```

Promise.finally

A utility function that is called when the Promise settles, either if it's resolved or rejected. It brings the `try..finally` construct to Promises.

```
getUser() // returns a Promise
  .finally(() => {
  // called both even if the Promise is rejected
    console.log("finished");
  });
// does not change the result value
```

Promise.race

A utility function that gets multiple Promises and resolves (or rejects) when the first one settles.

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

const fn = async (ms, result) => {
  await wait(ms);

  return result;
}
console.log(
  await Promise.race([
    fn(100, "a"),
    fn(50, "b"),
  ])
);
// b
```

Promise.reject

A utility function that rejects with the value provided.

```
Promise.reject(new Error("error"))
  .catch((v) => {
    console.log(v.message); // error
  })
```

Promise.resolve

A utility function that returns a Promise that is resolved with the value provided. Useful to start a Promise chain.

```
Promise.resolve("value")
  .then((v) => {
    console.log(v); // value
  })
```

Promise.then

A function of a Promise object that attaches a function that will be called when the Promise is settled. See Promise chain.

Promisification

The process of converting callbacks to Promises.

Rejected Promise

A Promise state that indicates an error. See Promise states.

Resolved Promise

A Promise state where the asynchronous value is available. See Promise states.

Settled Promise

An alternative name for the fulfilled state. See Promise states.

Synchronous function

A function that returns a value when it's run. See Continuation-passing style.

About the author

Tamás Sallai

Hey, I'm Tamás! I'm a software developer specializing in web technologies and cloud computing. I'm especially interested in exploring edge cases and "what ifs" in order to write dependable software.

I co-author the https://advancedweb.hu blog where I've published more than 200 articles in various technology topics. You can always find what I'm currently working on there.