



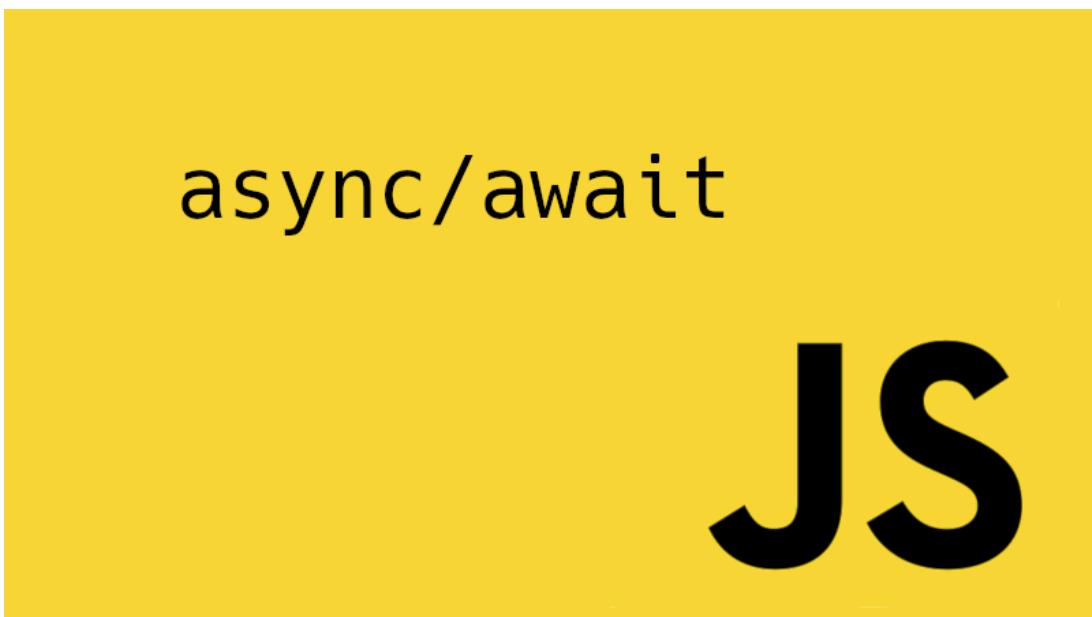
Charlee Li

[Follow](#)

Full stack engineer &amp; Tech writer @ Toronto.

Jun 8 · 7 min read

# JavaScript async/await: The Good Part, Pitfalls and How to Use



The `async/await` introduced by ES7 is a fantastic improvement in asynchronous programming with JavaScript. It provided an option of using synchronous style code to access resources asynchronously, without blocking the main thread. However it is a bit tricky to use it well. In this article we will explore `async/await` from different perspectives, and will show how to use them correctly and effectively.

## The good part in `async/await`

The most important benefit `async/await` brought to us is the synchronous programming style. Let's see an example.

```
// async/await
async getBooksByAuthorWithAwait(authorId) {
  const books = await bookModel.fetchAll();
  return books.filter(b => b.authorId === authorId);
}

// promise
getBooksByAuthorWithPromise(authorId) {
  return bookModel.fetchAll()
    .then(books => books.filter(b => b.authorId ===
authorId));
}
```

It is obvious that the `async/await` version is way easier understanding than the promise version. If you ignore the `await` keyword, the code just looks like any other synchronous languages such as Python.

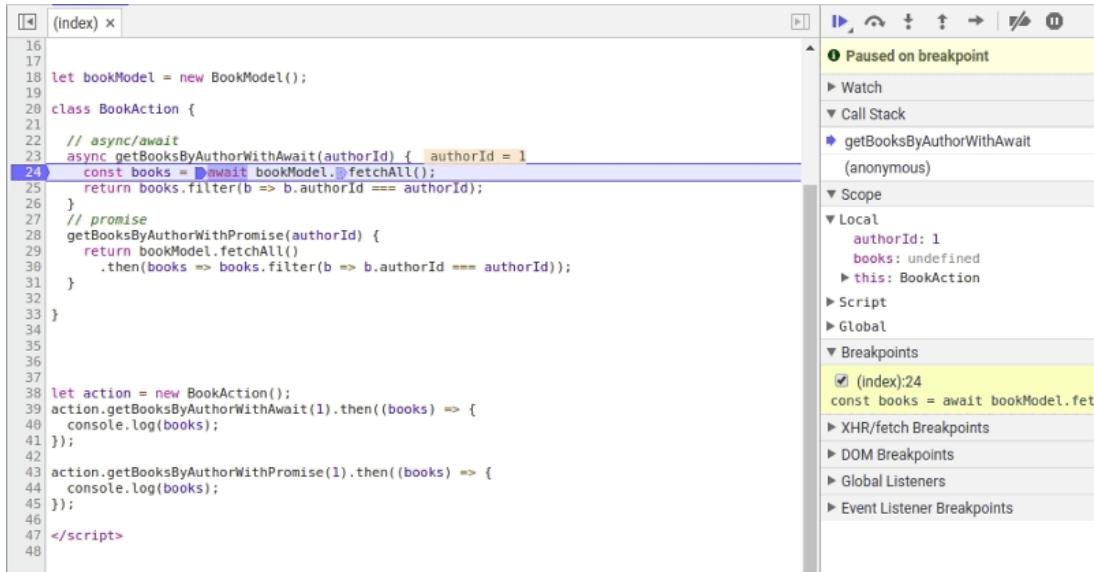
And the sweet spot is not only readability. `async/await` has native browser support. As of today, all the mainstream browsers have full support to `async` functions.



All mainstream browsers support Async functions. (Source: <https://caniuse.com/>)

Native support means you don't have to transpile the code. More importantly, it facilitates debugging. When you set a breakpoint at the function entry point and step over the `await` line, you will see the debugger halt for a short while while `bookModel.fetchAll()` doing its

job, then it moves to the next `.filter` line! This is much easier than the promise case, in which you have to setup another breakpoint on the `.filter` line.



The screenshot shows a browser developer tools debugger interface. The left pane displays a script file named '(index) x' with code. Line 24 contains the code: `const books = await bookModel.fetchAll();`. The right pane shows the debugger sidebar with the following details:

- Paused on breakpoint**
- Call Stack**: getBooksByAuthorWithAwait (anonymous)
- Scope**: Local, Global
- Breakpoints**: A checkbox labeled '(index):24 const books = await bookModel.fetchAll()' is checked.

Debugging async function. Debugger will wait at the await line and move to the next on resolved.

Another less obvious benefit is the `async` keyword. It declares that the `getBooksByAuthorWithAwait()` function return value is guaranteed to be a promise, so that callers can call

`getBooksByAuthorWithAwait().then(...)` or `await getBooksByAuthorWithAwait()` safely. Think about this case (bad practice!):

```
getBooksByAuthorWithPromise(authorId) {
  if (!authorId) {
    return null;
  }
  return bookModel.fetchAll()
    .then(books => books.filter(b => b.authorId ===
authorId));
}
```

In above code, `getBooksByAuthorWithPromise` may return a promise (normal case) or a `null` value (exceptional case), in which case caller cannot call `.then()` safely. With `async` declaration, it becomes impossible for this kind of code.

## Async/await Could Be Misleading

Some articles compare `async/await` with `Promise` and claim it is the next generation in the evolution of JavaScript asynchronous programming, which I respectfully disagree. `Async/await` IS an improvement, but it is no more than a syntactic sugar, which will not change our programming style completely.

Essentially, `async` functions are still promises. You have to understand promises before you can use `async` functions correctly, and even worse, most of the time you need to use promises along with `async` functions.

Consider the `getBooksByAuthorWithAwait()` and `getBooksByAuthorWithPromises()` functions in above example. Note that they are not only identical functionally, they also have exactly the same interface!

This means `getBooksByAuthorWithAwait()` will return a promise if you call it directly.

Well, this is not necessarily a bad thing. Only the name `await` gives people a feeling that “Oh great this can convert asynchronous functions to synchronous functions” which is actually wrong.

## Async/await Pitfalls

So what mistakes may be made when using `async/await`? Here are some common ones.

### Too Sequential

Although `await` can make your code look like synchronous, keep in mind that they are still asynchronous and care must be taken to avoid being too sequential.

```
async getBooksAndAuthor(authorId) {
  const books = await bookModel.fetchAll();
  const author = await authorModel.fetch(authorId);
  return {
    author,
    books: books.filter(book => book.authorId === authorId),
  };
}
```

This code looks logically correct. However this is wrong.

1. `await bookModel.fetchAll()` will wait until `fetchAll()` returns.
2. Then `await authorModel.fetch(authorId)` will be called.

Notice that `authorModel.fetch(authorId)` does not depend on the result of `bookModel.fetchAll()` and in fact they can be called in parallel! However by using `await` here these two calls become sequential and the total execution time will be much longer than the parallel version.

Here is the correct way:

```
async getBooksAndAuthor(authorId) {
  const bookPromise = bookModel.fetchAll();
  const authorPromise = authorModel.fetch(authorId);
  const book = await bookPromise;
  const author = await authorPromise;
  return {
    author,
    books: books.filter(book => book.authorId === authorId),
  };
}
```

Or even worse, if you want to fetch a list of items one by one, you have to rely on promises:

```
async getAuthors(authorIds) {  
    // WRONG, this will cause sequential calls  
    // const authors = _.map(  
    //   authorIds,  
    //   id => await authorModel.fetch(id));  
  
    // CORRECT  
    const promises = _.map(authorIds, id =>  
      authorModel.fetch(id));  
    const authors = await Promise.all(promises);  
}
```

In short, you still need to think about the workflows asynchronously, then try to write code synchronously with `await`. In complicated workflow it might be easier to use promises directly.

## Error Handling

With promises, an `async` function have two possible return values: resolved value, and rejected value. And we can use `.then()` for normal case and `.catch()` for exceptional case. However with `async/await` error handling could be tricky.

### try...catch

The most standard (and my recommended) way is to use `try...catch` statement. When `await` a call, any rejected value will be thrown as an exception. Here is an example:

```

class BookModel {
  fetchAll() {
    return new Promise((resolve, reject) => {
      window.setTimeout(() => { reject({ 'error': 400 }) }, 1000);
    });
  }
}

// async/await
async getBooksByAuthorWithAwait(authorId) {
  try {
    const books = await bookModel.fetchAll();
  } catch (error) {
    console.log(error); // { "error": 400 }
  }
}

```

The `catch` ed error is exactly the rejected value. After we caught the exception, we have several ways to deal with it:

- Handle the exception, and return a normal value. (Not using any `return` statement in the `catch` block is equivalent to using `return undefined;` and is a normal value as well.)
- Throw it, if you want the caller to handle it. You can either throw the plain error object directly like `throw error;`, which allows you to use this `async getBooksByAuthorWithAwait()` function in a promise chain (i.e. you can still call it like
 

```
getBooksByAuthorWithAwait().then(...).catch(error => ...);
```

 Or you can wrap the error with `Error` object, like `throw new Error(error)`, which will give the full stack trace when this error is displayed in the console.
- Reject it, like `return Promise.reject(error)`. This is equivalent to `throw error` so it is not recommended.

The benefits of using `try...catch` are:

- Simple, traditional. As long as you have experience of other languages such as Java or C++, you won't have any difficulty understanding this.
- You can still wrap multiple `await` calls in a single `try...catch` block to handle errors in one place, if per-step error handling is not necessary.

There is also one flaw in this approach. Since `try...catch` will catch every exception in the block, some other exceptions which are not usually caught by promises will be caught. Think about this example:

```
class BookModel {  
    fetchAll() {  
        cb(); // note `cb` is undefined and will result an  
        // exception  
        return fetch('/books');  
    }  
}  
  
try {  
    bookModel.fetchAll();  
} catch(error) {  
    console.log(error); // This will print "cb is not  
    // defined"  
}
```

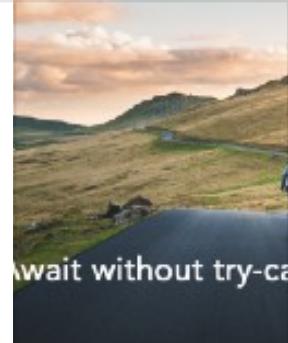
Run this code and you will get an error `ReferenceError: cb is not defined` in the console, in black color. The error was output by `console.log()` but not the JavaScript itself. Sometimes this could be fatal: If `BookModel` is enclosed deeply in a series of function calls and one of the call swallows the error, then it will be extremely hard to find an undefined error like this.

## Making functions return both value

Another way for error handling is inspired by Go language. It allows `async` function to return both the error and the result. See this blog post for the detail:

### How to write async await without try-catch blocks in Javascript

ES7 Async/await allows us as developers to write asynchronous JS code that look synchronous. In...  
[blog.grossman.io](http://blog.grossman.io)



In short, you can use `async` function like this:

```
[err, user] = await to(UserModel.findById(1));
```

Personally I don't like this approach since it brings Go style into JavaScript which feels unnatural, but in some cases this might be quite useful.

## Using `.catch`

The final approach we will introduce here is to continue using `.catch()`.

Recall the functionality of `await`: It will wait for a promise to complete its job. Also please recall that `promise.catch()` will return a promise too! So we can write error handling like this:

```
// books === undefined if error happens,
// since nothing returned in the catch statement
let books = await bookModel.fetchAll()
  .catch((error) => { console.log(error); });
```

There are two minor issues in this approach:

- It is a mixture of promises and async functions. You still need to understand how promises work to read it.
  - Error handling comes before normal path, which is not intuitive.
- . . .

## Conclusion

The `async/await` keywords introduced by ES7 is definitely an improvement to JavaScript asynchronous programming. It can make code easier to read and debug. However in order to use them correctly, one must completely understand promises, since they are no more than syntactic sugar, and the underlying technique is still promises.

Hope this post can give you some ideas about `async/await` themselves, and can help you prevent some common mistakes. Thanks for your reading, and please clap for me if you like this post.



