ES5, Promises and async/await Conclusion

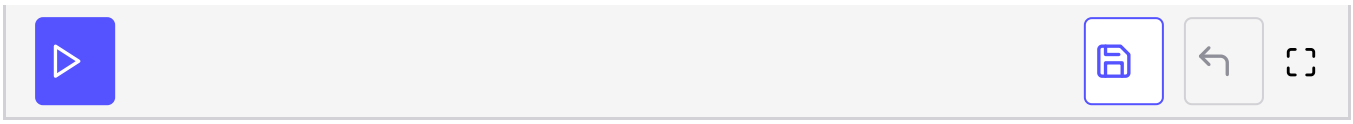
Learn the crucial nuances between the three different forms of asynchronous code we've covered so far.

The Different Forms of Async Code

This is an excellent opportunity to observe the differences between the different forms of asynchronous code.

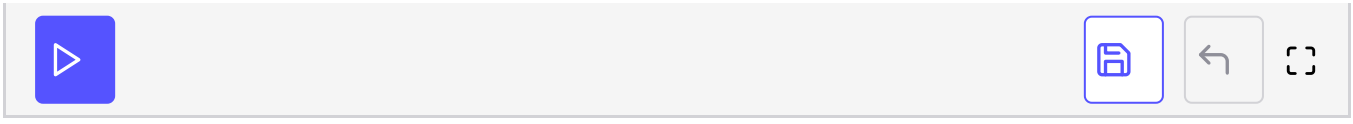
Here's the code we started with, with error handling:

```
onRequest((request, response) => {
    try {
        readFile(request.fileToRead, data =>
            writeFile(request.fileToWrite, data, status =>
                response.send(status)
            )
        );
    } catch(err) {
        console.log(err);
        response.send(err);
    }
});
```



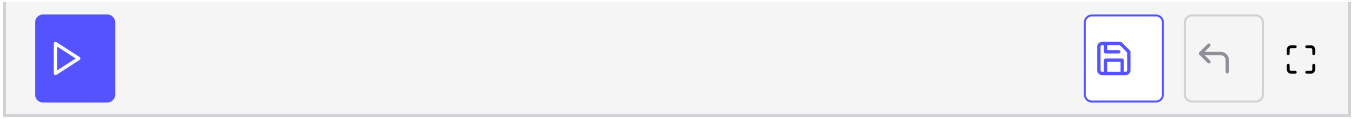
Here's the code transformed using Promises and ES2015.

```
onRequest((request, response) =>
    readFile(request.fileToRead)
    .then(data => writeFile(request.fileToWrite, data))
    .then(status => response.send(status))
    .catch(err => {
        console.log('Caught error:', err);
        response.send(err);
    })
);
```



And here it is using `async`/`await`. There are no special tools to work with errors here. We simply use a standard `try`/`catch` block.

```
onRequest(async (request, response) => {
    try {
        const readFileData = await readFile(request.fileToRead);
        const writeStatus = await writeFile(request.fileToWrite, readFileData);
        response.send(writeStatus);
    } catch(err) {
        console.log('Caught error:', err);
        response.send(err);
    }
});
```



Discussion

Gaining experience with these different mechanisms reveals some insights into their strengths and weaknesses.

ES5

Standard ES5 code will callbacks is easily the most confusing form of code here. We have a pyramid of doom inside a `try`/`catch` block. It may not seem too complicated here; our pyramid is relatively simple. However, in production code, they are always much more complex. There will be longer pieces of logic inside each of the function calls.

There will also typically be more function calls. We might have to nest another function that ensures the user is authenticated into the site. We'll have a check to ensure that the user hasn't exceeded their allowed capacity. We'll have another to send information about the event to a separate database to maintain a record of operations. If we're charging the user for the service, we'll have to deal with a large amount of code that accepts a credit card.

In short, it gets very hairy very quickly. The *only* time I can recommend using ES5 code is with a platform that doesn't support ES2015+. As time goes on, these become more and more rare, so it shouldn't be an issue.

Promises

Promises are a huge improvement over standard callbacks. We won't have deeply nested functions. We'll be able to scan the code from top to bottom and

the chain of `.then` methods makes the order of events very clear.

Promises also have some functionality that `async`/`await` doesn't. For example, we can achieve true parallelism for asynchronous function calls, speeding up our code.

A problem with Promises is the amount of time and effort they take to learn. A senior JavaScript engineer can pick up a concept like this in minutes and use it successfully, but for novices, it can often prove a difficult mechanism to acclimate to, due to the nature of asynchronous code and callbacks in general.

`async`/`await`

This is the newest mechanism in place for asynchronous code. It is easily the cleanest form of async code that we've seen so far. It makes asynchronous code more accessible to all JavaScript engineers.

Both keywords signify a clear purpose. The presence of `async` indicates that `await` will be used, and `await` tells us that the function is going to pause until its result comes back.

There are real drawbacks to `async`/`await` however. As we saw in previous lessons, using it incorrectly can lead to functions that are unnecessarily slow. A programmer who isn't careful may make function calls in series when they should really be made in parallel.

This mechanism also unfortunately suffers from the problem of complexity. While the intent of the code is easy to see, the underlying fundamentals are unclear. The fact that we're waiting for the resolution of a Promise whenever we use `await` might never occur to someone without proper knowledge.

In the end, `async`/`await` is really just a wrapper around Promises. It provides an alternative, easier syntax to deal with asynchronous code. The fact that the syntax is easier to read, however, is what gives this mechanism its power. Simpler code means less cognitive stress when we're trying to write code.

This results in code that more engineers can understand and work with. The clarity and accessibility of the code makes it so that code written with this mechanism is often less bug-prone and bugs are easier to spot by others.

That's it for asynchronous JavaScript. We've come quite a ways.