9. Async functions

The ECMAScript feature “Async Functions” was proposed by Brian Terlson.

9.1 Overview

9.1.1 Variants

The following variants of async functions exist. Note the keyword `async` everywhere.

- Async function declarations: `async function foo() {}`
- Async function expressions: `const foo = async function () {};`
- Async method definitions: `let obj = { async foo() {} }`
- Async arrow functions: `const foo = async () => {};`

9.1.2 Async functions always return Promises

Fulfilling the Promise of an async function:

```
async function asyncFunc() {
    return 123;
}

asyncFunc()
.then(x => console.log(x));
    // 123
```

Rejecting the Promise of an async function:

```
async function asyncFunc() {
    throw new Error('Problem!');
}

asyncFunc()
.catch(err => console.log(err));
    // Error: Problem!
```

9.1.3 Handling results and errors of asynchronous computations via `await`

The operator `await` (which is only allowed inside async functions) waits for its operand, a Promise, to be settled:

- If the Promise is fulfilled, the result of `await` is the fulfillment value.
- If the Promise is rejected, `await` throws the rejection value.

Handling a single asynchronous result:

```
async function asyncFunc() {
    const result = await otherAsyncFunc();
    console.log(result);
}
```

```
// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc()
    .then(result => {
        console.log(result);
    });
}
```

Handling multiple asynchronous results sequentially:

```
async function asyncFunc() {
    const result1 = await otherAsyncFunc1();
    console.log(result1);
    const result2 = await otherAsyncFunc2();
    console.log(result2);
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc1()
    .then(result1 => {
        console.log(result1);
        return otherAsyncFunc2();
    })
    .then(result2 => {
        console.log(result2);
    });
}
```

Handling multiple asynchronous results in parallel:

```
async function asyncFunc() {
    const [result1, result2] = await Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ]);
    console.log(result1, result2);
}

// Equivalent to:
function asyncFunc() {
    return Promise.all([
        otherAsyncFunc1(),
        otherAsyncFunc2(),
    ])
    .then([result1, result2] => {
        console.log(result1, result2);
    });
}
```

Handling errors:

```
async function asyncFunc() {
    try {
        const result = await otherAsyncFunc();
    } catch (err) {
        console.error(err);
    }
}

// Equivalent to:
function asyncFunc() {
    return otherAsyncFunc()
    .catch(err => {
        console.error(err);
    });
}
```

9.2 Understanding async functions

Before I can explain async functions, I need to explain how Promises and generators can be combined to perform asynchronous operations via synchronous-looking code.

For functions that compute their one-off results asynchronously, Promises, which are part of ES6, have become popular. One example is the client-side `fetch` API, which is an alternative to XMLHttpRequest for retrieving files. Using it looks as follows:

```
function fetchJson(url) {
    return fetch(url)
    .then(request => request.text())
    .then(text => {
        return JSON.parse(text);
    })
    .catch(error => {
        console.log(`ERROR: ${error.stack}`);
    });
}
fetchJson('http://example.com/some_file.json')
.then(obj => console.log(obj));
```

9.2.1 Writing asynchronous code via generators

co is a library that uses Promises and generators to enable a coding style that looks more synchronous, but works the same as the style used in the previous example:

```
const fetchJson = co.wrap(function* (url) {
    try {
        let request = yield fetch(url);
        let text = yield request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
});
```

Every time the callback (a generator function!) yields a Promise to co, the callback gets suspended. Once the Promise is settled, co resumes the callback: if the Promise was fulfilled, `yield` returns the fulfillment value, if it was rejected, `yield` throws the rejection error. Additionally, co promisifies the result returned by the callback (similarly to how `then()` does it).

9.2.2 Writing asynchronous code via async functions

Async functions are basically dedicated syntax for what co does:

```
async function fetchJson(url) {
    try {
        let request = await fetch(url);
        let text = await request.text();
        return JSON.parse(text);
    }
    catch (error) {
        console.log(`ERROR: ${error.stack}`);
    }
}
```

Internally, async functions work much like generators.

9.2.3 Async functions are started synchronously, settled asynchronously

This is how async functions are executed:

1. The result of an async function is always a Promise `p`. That Promise is created when starting the execution of the async function.
2. The body is executed. Execution may finish permanently via `return` or `throw`. Or it may finish temporarily via `await`; in which case execution will usually continue later on.
3. The Promise `p` is returned.

While executing the body of the async function, `return x` resolves the Promise `p` with `x`, while `throw err` rejects `p` with `err`. The notification of a settlement happens asynchronously. In other words: the callbacks of `then()` and `catch()` are always executed after the current code is finished.

The following code demonstrates how that works:

```
async function asyncFunc() {
    console.log('asyncFunc()'); // (A)
    return 'abc';
}
asyncFunc().
then(x => console.log(`Resolved: ${x}`)); // (B)
console.log('main'); // (C)

// Output:
// asyncFunc()
// main
// Resolved: abc
```

You can rely on the following order:

1. Line (A): the async function is started synchronously. The async function's Promise is resolved via `return`.
2. Line (C): execution continues.
3. Line (B): Notification of Promise resolution happens asynchronously.

9.2.4 Returned Promises are not wrapped

Resolving a Promise is a standard operation. `return` uses it to resolve the Promise `p` of an async function. That means:

1. Returning a non-Promise value fulfills `p` with that value.
2. Returning a Promise means that `p` now mirrors the state of that Promise.

Therefore, you can return a Promise and that Promise won't be wrapped in a Promise:

```
async function asyncFunc() {
    return Promise.resolve(123);
}
asyncFunc()
.then(x => console.log(x)) // 123
```

Intriguingly, returning a rejected Promise leads to the result of the async function being rejected (normally, you'd use `throw` for that):

```
async function asyncFunc() {
    return Promise.reject(new Error('Problem!'));
}
asyncFunc()
.catch(err => console.error(err)); // Error: Problem!
```

That is in line with how Promise resolution works. It enables you to forward both fulfillments and rejections of another asynchronous computation, without an `await`:

```
async function asyncFunc() {
    return anotherAsyncFunc();
}
```

The previous code is roughly similar to – but more efficient than – the following code (which unwraps the Promise of `anotherAsyncFunc()` only to wrap it again):

```
async function asyncFunc() {
    return await anotherAsyncFunc();
}
```

9.3 Tips for using `await`

9.3.1 Don't forget `await`

One easy mistake to make in async functions is to forget `await` when making an asynchronous function call:

```
async function asyncFunc() {
    const value = otherAsyncFunc(); // missing `await`!
    ···
}
```

In this example, `value` is set to a Promise, which is usually not what you want in async functions.

`await` can even make sense if an async function doesn't return anything. Then its Promise is simply used as a signal for telling the caller that it is finished. For example:

```
async function foo() {
    await step1(); // (A)
    ···
}
```

The `await` in line (A) guarantees that `step1()` is completely finished before the remainder of `foo()` is executed.

9.3.2 You don't need `await` if you "fire and forget"

Sometimes, you only want to trigger an asynchronous computation and are not interested in when it is finished. The following code is an example:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello'); // don't wait
```

```
    writer.write('world'); // don't wait
    await writer.close(); // wait for file to close
}
```

Here, we don't care when individual writes are finished, only that they are executed in the right order (which the API would have to guarantee, but that is encouraged by the execution model of async functions – as we have seen).

The `await` in the last line of `asyncFunc()` ensures that the function is only fulfilled after the file was successfully closed.

Given that returned Promises are not wrapped, you can also `return` instead of `await writer.close()`:

```
async function asyncFunc() {
    const writer = openFile('someFile.txt');
    writer.write('hello');
    writer.write('world');
    return writer.close();
}
```

Both versions have pros and cons, the `await` version is probably slightly easier to understand.

9.3.3 `await` is sequential, `Promise.all()` is parallel

The following code make two asynchronous function calls, `asyncFunc1()` and `asyncFunc2()`.

```
async function foo() {
    const result1 = await asyncFunc1();
    const result2 = await asyncFunc2();
}
```

However, these two function calls are executed sequentially. Executing them in parallel tends to speed things up. You can use `Promise.all()` to do so:

```
async function foo() {
    const [result1, result2] = await Promise.all([
        asyncFunc1(),
        asyncFunc2(),
    ]);
}
```

Instead of awaiting two Promises, we are now awaiting a Promise for an Array with two elements.

9.4 Async functions and callbacks

One limitation of async functions is that `await` only affects the directly surrounding async function. Therefore, an async function can't `await` in a callback (however, callbacks can be async functions themselves, as we'll see later on). That makes callback-based utility functions and methods tricky to use. Examples include the Array methods `map()` and `forEach()`.

9.4.1 `Array.prototype.map()`

Let's start with the Array method `map()`. In the following code, we want to download the files pointed to by an Array of URLs and return them in an Array.

```
async function downloadContent(urls) {
    return urls.map(url => {
        // Wrong syntax!
        const content = await httpGet(url);
        return content;
    });
}
```

This does not work, because `await` is syntactically illegal inside normal arrow functions. How about using an async arrow function, then?

```
async function downloadContent(urls) {
    return urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
}
```

There are two issues with this code:

- The result is now an Array of Promises, not an Array of strings.
- The work performed by the callbacks isn't finished once `map()` is finished, because `await` only pauses the surrounding arrow function and `httpGet()` is resolved asynchronously. That means you can't use `await` to wait until `downloadContent()` is finished.

We can fix both issues via `Promise.all()`, which converts an Array of Promises to a Promise for an Array (with the values fulfilled by the Promises):

```
async function downloadContent(urls) {
    const promiseArray = urls.map(async (url) => {
        const content = await httpGet(url);
        return content;
    });
    return await Promise.all(promiseArray);
}
```

The callback for `map()` doesn't do much with the result of `httpGet()`, it only forwards it. Therefore, we don't need an async arrow function here, a normal arrow function will do:

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return await Promise.all(promiseArray);
}
```

There is one small improvement that we still can make: This async function is slightly inefficient – it first unwraps the result of `Promise.all()` via `await`, before wrapping it again via `return`. Given that `return` doesn't wrap Promises, we can return the result of `Promise.all()` directly:

```
async function downloadContent(urls) {
    const promiseArray = urls.map(
        url => httpGet(url));
    return Promise.all(promiseArray);
}
```

9.4.2 `Array.prototype.forEach()`

Let's use the Array method `forEach()` to log the contents of several files pointed to via URLs:

```
async function logContent(urls) {
    urls.forEach(url => {
        // Wrong syntax
        const content = await httpGet(url);
        console.log(content);
    });
}
```

Again, this code will produce a syntax error, because you can't use `await` inside normal arrow functions.

Let's use an async arrow function:

```
async function logContent(urls) {
    urls.forEach(async url => {
        const content = await httpGet(url);
        console.log(content);
    });
    // Not finished here
}
```

This does work, but there is one caveat: the Promise returned by `httpGet()` is resolved asynchronously, which means that the callbacks are not finished when `forEach()` returns. As a consequence, you can't await the end of `logContent()`.

If that's not what you want, you can convert `forEach()` into a `for-of` loop:

```
async function logContent(urls) {
    for (const url of urls) {
        const content = await httpGet(url);
        console.log(content);
    }
}
```

Now everything is finished after the `for-of` loop. However, the processing steps happen sequentially: `httpGet()` is only called a second time *after* the first call is finished. If you want the processing steps to happen in parallel, you must use `Promise.all()`:

```
async function logContent(urls) {
    await Promise.all(urls.map(
        async url => {
            const content = await httpGet(url);
            console.log(content);
        }));
}
```

`map()` is used to create an Array of Promises. We are not interested in the results they fulfill, we only `await` until all of them are fulfilled. That means that we are completely done at the end of this async function. We could just as well return `Promise.all()`, but then the result of the function would be an Array whose elements are all `undefined`.

9.5 Tips for using async functions

9.5.1 Know your Promises

The foundation of async functions is Promises. That's why understanding the latter is crucial for understanding the former. Especially when connecting old code that isn't based on Promises with async functions, you often have no choice but to use Promises directly.

For example, this is a "promisified" version of `XMLHttpRequest`:

```
function httpGet(url, responseType="") {
    return new Promise(
        function (resolve, reject) {
            const request = new XMLHttpRequest();
            request.onload = function () {
                if (this.status === 200) {
                    // Success
                    resolve(this.response);
                } else {
                    // Something went wrong (404 etc.)
                    reject(new Error(this.statusText));
                }
            };
            request.onerror = function () {
                reject(new Error(
                    'XMLHttpRequest Error: '+this.statusText));
            };
            request.open('GET', url);
            xhr.responseType = responseType;
            request.send();
        });
}
```

The API of `XMLHttpRequest` is based on callbacks. Promisifying it via an async function would mean that you'd have to fulfill or reject the Promise returned by the function from within callbacks. That's impossible, because you can only do so via `return` and `throw`. And you can't `return` the result of a function from within a callback. `throw` has similar constraints.

Therefore, the common coding style for async functions will be:

- Use Promises directly to build asynchronous primitives.
- Use those primitives via async functions.

Further reading: chapter "Promises for asynchronous programming" in "Exploring ES6".

9.5.2 Immediately Invoked Async Function Expressions

Sometimes, it'd be nice if you could use `await` at the top level of a module or script. Alas, it's only available inside async functions. You therefore have several options. You can either create an async function `main()` and call it immediately afterwards:

```
async function main() {
    console.log(await asyncFunction());
}
main();
```

Or you can use an Immediately Invoked Async Function Expression:

```
(async function () {
    console.log(await asyncFunction());
})();
```

Another option is an Immediately Invoked Async Arrow Function:

```
(async () => {
    console.log(await asyncFunction());
})();
```

9.5.3 Unit testing with async functions

9. Async functions
Table of contents
Please support this book: buy it (PDF, EPUB, MOBI) or donate

```
from 'assert';

llowing test always succeeds
async code', function () {
() // (A)
lt1 => {
.strictEqual(result1, 'a'); // (B)
 asyncFunc2();

lt2 => {
.strictEqual(result2, 'b'); // (C)

});
```

However, this test always succeeds, because mocha doesn't wait until the assertions in line (B) and line (C) are executed.

You can fix this by returning the result of the Promise chain, because mocha recognizes if a test returns a Promise and then waits until that Promise is settled (unless there is a timeout).

```
return asyncFunc1() // (A)
```

Conveniently, async functions always return Promises, which makes them perfect for this kind of unit test:

```
import assert from 'assert';
test('Testing async code', async function () {
    const result1 = await asyncFunc1();
    assert.strictEqual(result1, 'a');
    const result2 = await asyncFunc2();
    assert.strictEqual(result2, 'b');
});
```

There are thus two advantages to using async functions for asynchronous unit tests in mocha: the code is more concise and returning Promises is taken care of, too.

9.5.4 Don't worry about unhandled rejections

JavaScript engines are becoming increasingly good at warning about rejections that are not handled. For example, the following code would often fail silently in the past, but most modern JavaScript engines now report an unhandled rejection:

```
async function foo() {
    throw new Error('Problem!');
```

```
}
foo();
```

9.6 Further reading

- Async Functions (proposal by Brian Terlson)
- Simplifying asynchronous computations via generators (section in “Exploring ES6”)

9. Async functions
Table of contents
Please support this book: buy it (PDF, EPUB, MOBI) or donate