Table of Contents

How To Use This Book

How To Use This Book

Async/await is the single most valuable feature to land in the JavaScript language spec in the last 15 years. The event loop and asynchronous programming in general are exceptional for building GUIs and servers, but callbacks make error handling tedious and code hard to read. For example, when RisingStack asked Node.js developers what they struggled with in 2017, asynchronous programming topped the list.

What's hardest to get right with Node.js at the moment?
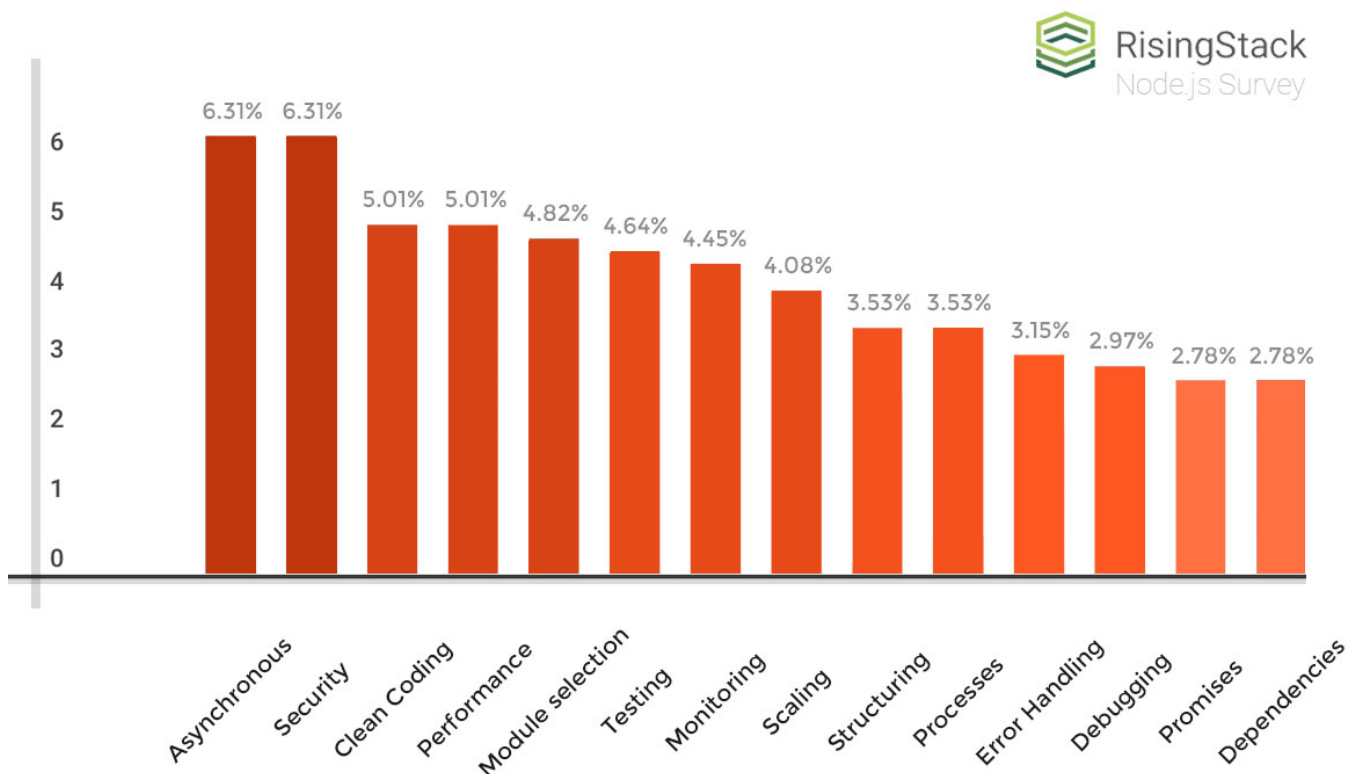
Async/await promises to make asynchronous code as clean and easy to read as synchronous code in most use cases. Tangled promise chains and complex user-land libraries like async can be replaced with `for` loops, `if` statements, and `try/catch` blocks that even the most junior of engineers can make sense of.

The following JavaScript from a 2012 blog post is a typical example of where code goes wrong with callbacks. This code works, but it has a lot of error handling boilerplate and deeply nested `if` statements that obfuscate the actual logic. Wrapping your mind around it takes a while, and proper error handling means copy/pasting `if (err != null)` into every callback.

```javascript
function getWikipediaHeaders() {
  // i. check if headers.txt exists
  fs.stat('./headers.txt', function(err, stats) {
    if (err != null) { throw err; }
    if (stats == undefined) {
      // ii. fetch the HTTP headers
      var options = { host: 'www.wikipedia.org', port: 80 };
      http.get(options, function(err, res) {
        if (err != null) { throw err; }
        var headers = JSON.stringify(res.headers);
        // iii. write the headers to headers.txt
        fs.writeFile('./headers.txt', headers, function(err) {
          if (err != null) { throw err; }
          console.log('Great Success!');
        });
      });
    } else { console.log('headers already collected'); }
  });
}
```

Below is the same code using async/await, assuming that `stat()`, `get()`, and `writeFile()` are properly promisified.

```javascript
async function getWikipediaHeaders() {
  if (await stat('./headers.txt') != null) {
    console.log('headers already collected');
  }
  const res = await get({ host: 'www.wikipedia.org', port: 80 });
  await writeFile('./headers.txt', JSON.stringify(res.headers));
  console.log('Great success!');
}
```

You might not think async/await is a big deal. You might even think async/await is a bad idea. I've been in your shoes: when I first learned about async/await in 2013, I thought it was unnecessary at best. But when I started working with generator-based coroutines (the 2015 predecessor to async/await), I was shocked at how quickly server crashes due to `TypeError: Cannot read property 'x' of undefined` vanished. By the time async/await became part of the JavaScript language spec in 2017, async/await was an indispensable part of my dev practice.

Just because async/await is now officially part of JavaScript doesn't mean the world is all sunshine and rainbows. Async/await is a new pattern that promises to make day-to-day development work easier, but, like any pattern, you need to understand it or you'll do more harm

than good. If your async/await code is a patchwork of copy/pasted StackOverflow answers, you're just trading callback hell for the newly minted async/await hell.

The purpose of this book is to take you from someone who is casually acquainted with promises and async/await to someone who is comfortable building and debugging a complex app whose core logic is built on async/await. This book is only 50 pages and is meant to be read in about 2 hours total. You may read it all in one sitting, but you would be better served reading one chapter at a time, studying the exercises at the end, and getting a good night's sleep to really internalize the information.

This book is broken up into 4 chapters. Each chapter is 12 pages, including exercises at the end of each chapter that highlight key lessons from the chapter. The exercises require more thought than code and should be easy to answer within a few minutes.

The first 3 chapters are focused on promise and async/await fundamentals, and strive to avoid frameworks and outside dependencies. In particular, the first 3 chapters' code samples and exercises are meant to run in Node.js 8.x and the first 3 chapters will **not** cover transpilers. In the interest of providing realistic examples, the code samples will use the `superagent` module for making HTTP requests. The 4th chapter will discuss transpilers and integrating async/await with some common npm modules.

If you find any issues with the code samples or exercises, please report them at github.com/vkarpov15/mastering-async-await-issues.

Are you ready to master async/await? Let's get started!

Async/Await: The Good Parts

The `async` and `await` keywords are new additions to JavaScript as part of the 2017 edition of the language specification. The `async` keyword modifies a function, either a normal `function() {}` or an arrow function `() => {}`, to mark it as an *async function*. In an async function, you can use the `await` keyword to pause the function's execution until a promise settles. In the below function, the `await` keyword pauses the function's execution for approximately 1 second.

Example 1.1

```
async function test() {
  // This function will print "Hello, World!" after 1 second.
  await new Promise(resolve => setTimeout(() => resolve(), 1000));
  console.log('Hello, World!');
}

test();
```

You can use the `await` keyword anywhere in the body of an async function. This means you can use `await` in `if` statements, `for` loops, and `try/catch` blocks. Below is another way to pause an async function's execution for about 1 second.

Example 1.2

```
async function test() {
  // Wait for 100ms 10 times. This function also prints after 1 second.
  for (let i = 0; i < 10; ++i) {
    await new Promise(resolve => setTimeout(() => resolve(), 100));
  }
  console.log('Hello, World!');
}

test();
```

There is one major restriction for using `await`: you can only use `await` within the body of a function that's marked `async`. The following code throws a `SyntaxError`.

Example 1.3

```
function test() {
  const p = new Promise(resolve => setTimeout(() => resolve(), 1000));
  // SyntaxError: Unexpected identifier
  await p;
}
```

In particular, you can't use `await` in a closure embedded in an async function, unless the closure is also an async function. The below code also throws a `SyntaxError`.

Example 1.4

```
const assert = require('assert');

async function test() {
  const p = Promise.resolve('test');
  assert.doesNotThrow(function() {
    // "SyntaxError: Unexpected identifier" because the above function
    // is **not** marked async. "Closure" = function inside a function
    await p;
  });
}
```

As long as you don't create a new function, you can use `await` underneath any number of `for` loops and `if` statements.

Example 1.5

```
async function test() {
  while (true) {
    // Convoluted way to print out "Hello, World!" once per second by
    // pausing execution for 200ms 5 times
    for (let i = 0; i < 10; ++i) {
      if (i % 2 === 0) {
        await new Promise(resolve => setTimeout(() => resolve(), 200));
      }
    }
    console.log('Hello, World!');
  }
}
```

Return Values

You can use `async/await` for more than just pausing execution. The return value of `await` is the value the promise is fulfilled with. This means you can assign a variable to an asynchronously-computed value in code that looks synchronous.

Example 1.6

```javascript
async function test() {
  // You can `await` on a non-promise without getting an error.
  let res = await 'Hello World!';
  console.log(res); // "Hello, World!"

  const promise = new Promise(resolve => {
    // This promise resolves to "Hello, World!" after 1s
    setTimeout(() => resolve('Hello, World!'), 1000);
  });
  res = await promise;
  // Prints "Hello, World!". `res` is equal to the value the
  // promise resolved to.
  console.log(res);

  // Prints "Hello, World!". You can use `await` in function params!
  console.log(await promise);
}
```

An async function **always** returns a promise. When you `return` from an async function, JavaScript resolves the promise to the value you returned. This means calling async functions from other async functions is very natural. You can `await` on the async function call and get the async function's "return value".

Example 1.7

```javascript
async function computeValue() {
  await new Promise(resolve => setTimeout(() => resolve(), 1000));
  // "Hello, World" is the _resolved value_ for this function call
  return 'Hello, World!';
}

async function test() {
  // Prints "Hello, World!" after 1s. `computeValue` returns a promise!
  console.log(await computeValue());
}
```

This book will refer to the value you `return` from an async function as the *resolved value*. In `computeValue` above, "Hello, World!" is the resolved value, `computeValue()` still returns a

promise. This distinction is subtle but important: the value you `return` from an async function body is **not** the value that an async function call like `computeValue()` without `await` returns.

You can also return a promise from an async function. In that case, the promise the async function returns will be fulfilled or rejected whenever the resolved value promise is fulfilled or rejected. Below is an equivalent implementation of an async function that fulfills to the string 'Hello, World!' after 1 second:

Example 1.8

```
async function computeValue() {
  // The resolved value is a promise. JavaScript will fulfill the promise
  // the `computeValue()` function call returns with the same value
  // that the resolved value is fulfilled with, in this case
  // 'Hello, World!'.
  return new Promise(resolve => {
    setTimeout(() => resolve('Hello, World!'));
  }, 1000);
}
```

If you `return` a promise from an async function, the resolved value will still not equal the return value. The below example demonstrates that the `resolvedValue` promise that the function body returns is not the same as the return value from `computeValue()`.

Example 1.9

```
let resolvedValue;

async function computeValue() {
  resolvedValue = new Promise(resolve => {
    setTimeout(() => resolve('Hello, World!'), 1000);
  });
  return resolvedValue;
}

async function test() {
  const returnValue = computeValue();
  // This will print `false`. The return value of an async function
  // never equals the resolved value!
  console.log(returnValue === resolvedValue);
  console.log(await returnValue);
}
```

A common mistake async/await novices make is always converting a value to a promise before returning it. They likely read somewhere that async functions always return a promise, and misinterpreted this fact to mean that you must always `return` a promise from an async function. You can `return` any value in an async function body.

```
async function computeValue() {
  // Adding `Promise.resolve()` below is unnecessary. It adds
  // perf overhead because you're creating an unnecessary promise.
  // "Unnecessary code is not as harmless as I used to think. It
  // sends the misleading signal that it's necessary." - Paul Graham
  return Promise.resolve('Hello, World!');
}
```

Error Handling

One of the most important properties of async/await is that you can use `try/catch` to handle asynchronous errors. Remember that a promise may be either fulfilled or rejected. When a promise `p` is fulfilled, JavaScript evaluates `await p` to the promise's value. What about if `p` is rejected?

Example 1.10

```
async function test() {
  try {
    const p = Promise.reject(new Error('Oops!'));
    // The below `await` throws
    await p;
  } catch (error) {
    console.log(err.message); // "Oops!"
  }
}
```

If `p` is rejected, `await p` throws an error that you can catch with a normal JavaScript `try/catch`. Note that the `await` statement is what throws an error, **not** the promise instantiation.

This `try/catch` behavior is a powerful tool for consolidating error handling. The `try/catch` block above can catch synchronous errors as well as asynchronous ones. Suppose you have code that throws a `TypeError: cannot read property 'x' of undefined` error:

Example 1.11

```
async function test() {
  try {
    const bad = undefined;
    bad.x;
    const p = Promise.reject(new Error('Oops!'));
    await p;
  } catch (error) {
    // "cannot read property 'x' of undefined"
    console.log(err.message);
  }
}
```

In callback-based code, you had to watch out for synchronous errors like `TypeError` separately from asynchronous errors. This lead to a lot of server crashes and red text in Chrome consoles, because discipline doesn't scale.

Consider using a callback-based approach instead of async/await. Suppose you have a black-box function `test()` that takes a single parameter, a `callback`. If you want to ensure you catch every possible error, you need 2 `try/catch` calls: one around `test()` and one around `callback()`. You also need to check whether `test()` called your callback with an error. In other words, every single async operation needs 3 distinct error handling patterns!

Example 1.12

```
function testWrapper(callback) {
  try {
    // There might be a sync error in `test()`
    test(function(error, res) {
      // `test()` might also call the callback with an error
      if (error) {
        return callback(error);
      }
      // And you also need to be careful that accessing `res.x` doesn't
      // throw **and** calling `callback()` doesn't throw.
      try {
        return callback(null, res.x);
      } catch (error) {
        return callback(error);
      }
    });
  }
}
```

When there's this much boilerplate for error handling, even the most rigorous and disciplined developers end up missing a spot. The result is uncaught errors, server downtime, and buggy user interfaces. Below is an equivalent example with async/await. You can handle the 3 distinct error cases from example 1.12 with a single pattern.

Example 1.13

```
async function testWrapper() {
  try {
    // `try/catch` will catch sync errors in `test()`, async promise
    // rejections, and errors with accessing `res.x`.
    const res = await test();
    return res.x;
  } catch (error) {
    throw error;
```

```
      }
   }
```

Let's take a look at how the `throw` keyword works with async functions now that you've seen how `try/catch` works. When you `throw` in an async function, JavaScript will reject the returned promise. Remember that the value you `return` from an async function is called the resolved value. Similarly, this book will refer to the value you `throw` in an async function as the *rejected value*.

Example 1.14

```
async function computeValue() {
  // `err` is the "rejected value"
  const err = new Error('Oops!');
  throw err;
}

async function test() {
  try {
    const res = await computeValue();
    // Never runs
    console.log(res);
  } catch (error) {
    console.log(error.message); // "Oops!"
  }
}
```

Remember that the `computeValue()` function call itself does **not** throw an error in the `test()` function. The `await` keyword is what throws an error that you can handle with `try/catch`. The below code will print "No Error" unless you uncomment the `await` block.

Example 1.15

```
async function computeValue() {
  throw new Error('Oops!');
};

async function test() {
  try {
    const promise = computeValue();
    // With the below line commented out, no error will be thrown
    // await promise;
    console.log("No Error");
  } catch (error) {
    console.log(error.message); // Won't run
  }
}
```

Retrying Failed Requests

Let's tie together loops, return values, and error handling to handle a challenge that's particularly nasty with callbacks: retrying failed requests. Suppose you had logic that relied on a potentially unreliable HTTP request.

With callbacks or promise chains, retrying failed requests requires recursion, and recursion is less readable than the synchronous alternative of writing a `for` loop. Below is a simplified implementation of a `getWithRetry()` function using callbacks and the `superagent` HTTP client.

Example 1.16

```
function getWithRetry(url, numRetries, callback, retriedCount) {
  retriedCount = retriedCount || 0;
  superagent.get(url).end(function(error, res) {
    if (error) {
      if (retriedCount >= numRetries) { return callback(error); }
      return getWithRetry(url, numRetries, callback, retriedCount + 1);
    }
    return callback(null, res.body);
  });
}
```

Recursion is subtle and tricky to understand relative to a loop. Plus, the above code ignores the possibility of sync errors, because the `try/catch` spaghetti highlighted in example 1.10 would make this example unreadable. In short, this pattern is both brittle and cumbersome.

With async/await, you don't need recursion and you need one `try/catch` to handle sync and async errors. The async/await implementation is built on `for` loops, `try/catch`, and other constructs that should be familiar to even the most junior of engineers.

Example 1.17

```
async function getWithRetry(url, numRetries) {
  let lastError = null;
  for (let i = 0; i < numRetries; ++i) {
    try {
      // Can also do `return (await superagent.get(url)).body`
      // Note that `await superagent.get(url).body` does **not** work
      const res = await superagent.get(url);
      // Early return with async functions works as you'd expect
      return res.body;
    } catch (error) {
      lastError = error;
    }
  }
  throw lastError;
}
```

More generally, async/await makes executing async operations in series trivial. For example, let's say you had to load a list of blog posts from an HTTP API and then execute a separate HTTP request to load the comments for each blog post. This example uses the excellent JSONPlaceholder API that provides good test data.

Example 1.18

```
async function run() {
  const root = 'https://jsonplaceholder.typicode.com';
  const posts = await getWithRetry(`${root}/posts`, 3);
  for (const { id } of posts) {
    const comments =
      await getWithRetry(`${root}/comments?postId=${id}`, 3);
    console.log(comments);
  }
}
```

If this example seems trivial, that's good, because that's how programming should be. The JavaScript community has created an incredible hodge-podge of tools for executing asynchronous tasks in series, from `async.waterfall()` to Redux sagas to zones to co. Async/await makes all of these libraries and more unnecessary. Do you even need Redux middleware anymore?