ASYNC JAVASCRIPT

🌂 PATTERNS AND GOTCHAS ☔️

Hugo Di Francesco

Engineer at Concrete.cc

Slides: codewithhugo.com/async-js

Twitter: @hugo__df

CONTENTS 🐳

1. Asynchronicity in JavaScript (a history lesson)
2. Why `async/await`?
3. Gotchas
4. Patterns

ASYNCHRONICITY IN JAVASCRIPT

👽

Primitives:

- Callbacks
- Promises
- (Observables)
- `async/await`

What's asynchronous in a web application?

What's asynchronous in a web application?

tl;dr Most things

1. any network calls (HTTP, database)

1. any network calls (HTTP, database)
2. timers (`setTimeout`, `setInterval`)

1. any network calls (HTTP, database)
2. timers (`setTimeout`, `setInterval`)
3. filesystem access

1. any network calls (HTTP, database)
2. timers (`setTimeout`, `setInterval`)
3. filesystem access
... Anything else that can be offloaded

In JavaScript, these operations are non-blocking.

HTTP Request in Python:

```python
data = request(myUrl)
print(data)
```

HTTP Request in JavaScript:

```javascript
request(myUrl, (err, data) => {
  console.log(data);
});
```

Why non-blocking I/O?

You don't want to freeze your UI while you wait.

Why non-blocking I/O?

You don't want to freeze your UI while you wait.

Non-blocking -> waiting doesn't cost you compute cycles.

How non-blocking I/O is implemented (in JavaScript):

- pass a "callback" function
- it's called with the outcome of the async operation

NODE-STYLE CALLBACKS 🍬

```
myAsyncFn((err, data) => {
  if (err) dealWithIt(err);
  doSomethingWith(data);
})
```

A callback is:

- "just" a function
- in examples, usually anonymous functions (pass `function () {}` directly)
- according to some style guides, should be an arrow function (`() => {}`)
- called when the async operation

A Node-style callback is:

- called with any error(s) as the first argument/parameter, if there's no error, `null` is passed
- called with any number of "output" data as the other arguments

ie. `(err, data) => { /* more logic */ }`

NODE-STYLE CALLBACKS: PROBLEMS 🐹

1. CALLBACK *HELL* 🔼

```
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

For each asynchronous operation:

- extra level of indent
- lots of names for async output: `data`, `secondData`

2. SHADOWING VARIABLES 👻

```javascript
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

- `err` (in `myAsyncFn` callback) !== `err` (in `myOtherAsyncFn` callback) despite having the same name

3. DUPLICATED ERROR HANDLING ❄️

- 1 call to `handle(err)` per operation

```
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

4. SWALLOWED ERRORS 🔊

Ideal failure:

- fail early
- fail fast
- fail loud

Spot the unhandled error:

```
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

Silent error

```
myAsyncFn((err, data) => {
  if (err) handle(err)
  myOtherAsyncFn(data, (err, secondData) => {
    // Missing error handling!
    fun(data, secondData, (err) => {
      if (err) handle(err)
    })
    fn(data, secondData, (err) => {
      if (err) handle(err)
    })
  })
})
```

- `err` doesn't get handled
- 🤞 hope your linter caught that

CALLBACK PROBLEMS 📞

1. Callback hell (indents 👎)
2. Shadowed variables
3. Duplicated error-handling
4. Swallowed errors

BRING ON THE PROMISE 🙏

```
myAsyncFn()
  .then((data) => Promise.all([
    data,
    myOtherAsyncFn(data),
  ]))
  .then(([data, secondData]) => Promise.all([
    fun(data, secondData),
    fn(data, secondData),
  ]))
  .then(/* do anything else */)
  .catch((err) => handle(err));
```

Pros:

- Chainable: no crazy indent stuff
- Single error handler (`.catch`)
- Lots of tightly scoped functions

Cons:

- Lots of tightly scoped functions
- Very verbose way of returning multiple things.

```
.then((data) => Promise.all([
  data,
  myOtherAsyncFn(data),
]))
```

PROMISE GOTCHAS 👍

Gotcha: 🙅‍♀️ nesting them is super tempting.

```
myAsyncFn()
  .then((data) =>
    myOtherAsyncFn(data)
      .then(
        ([data, secondData]) =>
          Promise.all([
            fun(data, secondData),
            fn(data, secondData),
          ])
      )
  )
  .catch((err) => handle(err))
```

Solution: Avoid the Pyramid of Doom 💀

```
myAsyncFn()
  .then((data) => Promise.all([
    data,
    myOtherAsyncFn(data),
  ]))
  .then(([data, secondData]) => Promise.all([
    fun(data, secondData),
    fn(data, secondData),
  ]))
  .then(/* do anything else */)
  .catch((err) => handle(err))
```

Promises "flatten":

- you can return a Promise from a `then` and keep chaining

Gotcha: `onRejected` callback

The following works:

```
myAsyncFn()
  .then(
    (data) => myOtherAsyncFn(data),
    (err) => handle(err)
  );
```

But we're back to doing per-operation error-handling like in callbacks (potentially swallowing errors etc.)

Solution: avoid it, in favour of `.catch`

```
myAsyncFn()
  .then(
    (data) => myOtherAsyncFn(data)
  )
  .catch((err) => handle(err));
```

Unless you specifically need it

PROMISE 👎 RECAP 🚕

- Lots of tightly scoped functions
- Very verbose way of returning/passing multiple things

```
fn()
  .then((data) => Promise.all([
    data,
    myOtherAsyncFn(data),
  ]))
  .then(([data, secondData]) => {})
```

ASYNC/AWAIT 👣

```
(async () => {
  try {
    const data = await myAsyncFn();
    const secondData = await myOtherAsyncFn(data);
    const final = await Promise.all([
      fun(data, secondData),
      fn(data, secondData),
    ]);
    /* do anything else */
  } catch (err) {
    handle(err);
  }
})();
```

Given a Promise (or any object that has a `.then` function), `await` takes the value passed to the callback in `.then`

- `await` can only be used inside a function that is `async` *

```javascript
(async () => {
  console.log('Immediately invoked function expressions (IIFEs
  const res = await fetch('https://jsonplaceholder.typicode.co
  const data = await res.json()
  console.log(data)
})()

// SyntaxError: await is only valid in async function
const res = await fetch(
  'https://jsonplaceholder.typicode.com/todos/2'
)
```

* top-level (ie. outside of async functions) await is coming

- `async` functions are "just" Promises

```javascript
const arrow = async () => { return 1 }
const implicitReturnArrow = async () => 1
const anonymous = async function () { return 1 }
async function expression () { return 1 }

console.log(arrow()); // Promise { 1 }
console.log(implicitReturnArrow()); // Promise { 1 }
console.log(anonymous()); // Promise { 1 }
console.log(expression()); // Promise { 1 }
```

LOOP THROUGH SEQUENTIAL CALLS 🔁

With async/await:

```
async function fetchSequentially(urls) {
  for (const url of urls) {
    const res = await fetch(url);
    const text = await res.text();
    console.log(text.slice(0, 100));
  }
}
```

With promises:

```
function fetchSequentially(urls) {
  const [ url, ...rest ] = urls
  fetch(url)
    .then(res => res.text())
    .then(text => console.log(text.slice(0, 100)))
    .then(fetchSequentially(rest));
}
```

SHARE DATA BETWEEN CALLS 🛁

```
async function run() {
  const data = await myAsyncFn();
  const secondData = await myOtherAsyncFn(data);
  const final = await Promise.all([
    fun(data, secondData),
    fn(data, secondData),
  ]);

  return final
}
```

We don't have the whole

```
.then(() => Promise.all([dataToPass, promiseThing]))
.then(([data, promiseOutput]) => { })
```

ERROR HANDLING 🎤

```javascript
async function withErrorHandling(url) {
  try {
    const res = await fetch(url);
    const data = await res.json();
    return data
  } catch(e) {
    console.log(e.stack)
  }
}

withErrorHandling(
  'https://jsonplaceholer.typicode.com/todos/2'
  // The domain should be jsonplaceholder.typicode.com
).then(() => { /* but we'll end up here */ })
```

CONS OF `async/await` 🤘

- Browser support is only good in latest/modern browsers
  - polyfills (async-to-gen, regenerator runtime) are sort of big
  - supported in Node 8+ though 🤷‍♀️
- Keen functional programming people would say it's leads to a more "imperative" style of programming

GOTCHAS 🏜️

CREATING AN ERROR 👝

- `throw`-ing inside an `async` function and `Promise.reject` work the same
- `.reject` and `throw Error` objects please 🙏

```javascript
async function asyncThrow() {
  throw new Error('asyncThrow');
}
function rejects() {
  return Promise.reject(new Error('rejects'))
}
async function swallowError(fn) {
  try { await asyncThrow() }
  catch (e) { console.log(e.message, e.__proto__) }
  try { await rejects() }
  catch (e) { console.log(e.message, e.__proto__) }
}
swallowError() // asyncThrow Error {} rejects Error {}
```

WHAT HAPPENS WHEN YOU FORGET await? 🐙

- values are undefined
- TypeError: x.fn is not a function

```js
async function forgotToWait() {
  try {
    const res = fetch('https://jsonplaceholer.typicode.com/tod
    const text = res.text()
  } catch (e) {
    console.log(e);
  }
}

forgotToWait()
// TypeError: res.text is not a function
```

WHAT HAPPENS WHEN YOU FORGET await? 🐙

- console.log of Promise/async function
- *inserts 100th reminder*: an `async` function is a Promise

```
async function forgotToWait() {
  const res = fetch('https://jsonplaceholer.typicode.com/todos
  console.log(res)
}

forgotToWait()
// Promise { <pending> }
```

PROMISES EVALUATE EAGERLY ✨

- Promises don't wait for anything to execute, when you create it, it runs:

```javascript
new Promise((resolve, reject) => {
  console.log('eeeeager');
  resolve();
})
```

TESTING GOTCHAS 📙

- Jest supports Promises as test output (therefore also `async` functions)
- what if your test fails?

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

test('it should pass', async () => {
  doSomeSetup();

  await runCodeUnderTest();
  // the following never gets run
  doSomeCleanup();
})
```

TESTING GOTCHAS 📙

- *BUT* do your cleanup in "before/after" hooks, async test bodies crash and don't clean up which might make multiple tests fail

```
describe('feature', () => {
  beforeEach(() => doSomeSetup())
  afterEach(() => doSomeCleanup())
  test('it should pass', async () => {
    await runCodeUnderTest();
  })
})
```

PATTERNS 🌆

A lot of these are to avoid the pitfalls we've looked in the "gotchas" section.

RUNNING PROMISES IN PARALLEL 🏃

- `Promise.all`

```javascript
function fetchParallel(urls) {
  return Promise.all(
    urls.map(
      (url) =>
      fetch(url).then(res => res.json())
    )
  );
}
```

RUNNING PROMISES IN PARALLEL 🏃

- `Promise.all` + `map` over an `async` function
- Good for logging or when you've got non-trivial/business logic

```javascript
function fetchParallel(urls) {
  return Promise.all(
    urls.map(async (url) => {
      const res = await fetch(url);
      const data = await res.json();
      return data;
    })
  );
}
```

DELAY EXECUTION OF A PROMISE ✋

- Promises are eager, they just wanna run!
- Use a function that returns the Promise

```
function getX(url) {
  return fetch(url)
}

// or

const delay = url => fetch(url)
```

- No Promise, no eager execution
- Fancy people call the above "thunk"

SEPARATE MIX SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS 🏇

async fetch > do stuff in memory > async write back

```javascript
const fs = require('fs').promises

const fetchFile = () =>
  fs.readFile('path', 'utf-8');
const replaceAllThings = (text) =>
  text.replace(/a/g, 'b');
const writeFile = (text) =>
  fs.writeFile('path', text, 'utf-8');

(async () => {
  const text = await fetchFile();
  const newText = replaceAllThings(text);
  await writeFile(newText);
})();
```

RUNNING PROMISES SEQUENTIALLY 🔐

- using recursion + rest/spread and way too much bookkeeping

```javascript
function fetchSequentially(urls, data = []) {
  if (urls.length === 0) return data
  const [url, ...rest] = urls
  return fetch(url)
    .then(res => res.text())
    .then(text =>
      fetchSequentially(
        rest,
        [...data, text]
      ));
}
```

RUNNING PROMISES SEQUENTIALLY 🔐

- using `await` + a loop?

```
async function fetchSequentially(urls) {
  const data = []
  for (const url of urls) {
    const res = await fetch(url);
    const text = await res.text();
    data.push(text)
  }
  return data
}
```

PASSING DATA IN SEQUENTIAL ASYNC CALLS 💾

- return array + destructuring in next call, very verbose in Promise chains

```javascript
async function findLinks() { /* some implementation */ }

function crawl(url, parentText) {
  console.log('crawling links in: ', parentText);
  return fetch(url)
    .then(res => res.text())
    .then(text => Promise.all([
      findLinks(text),
      text
    ]))
    .then(([links, text]) => Promise.all(
      links.map(link => crawl(link, text))
    ));
}
```

PASSING DATA IN SEQUENTIAL ASYNC CALLS 💾

- `await` + data in the closure

```javascript
async function findLinks() { /* someimplementation */ }

async function crawl(url, parentText) {
  console.log('crawling links in: ', parentText);
  const res = await fetch(url);
  const text = await res.text();
  const links = await findLinks(text);
  return crawl(links, text);
}
```

ERROR HANDLING ❌

- try/catch, or `.catch` 🤷‍♀️

```
function withCatch() {
  return fetch('borked_url')
    .then(res => res.text())
    .catch(err => console.log(err))
}

async function withBlock() {
  try {
    const res = await fetch('borked_url');
    const text = await res.text();
  } catch (err) {
    console.log(err)
  }
}
```

WORKSHOP 🔪

- "callbackify"-ing a Promise-based API 🙈
- getting data in parallel using callbacks 💊
- "promisify"-ing a callback-based API (read/write file)
- Why we don't mix async and sync operations

FURTHER READING 📖

- Slides/write up (including workshop examples) at codewithhugo.com/async-js
- About non-blocking I/O in Node.js docs: nodejs.org/en/docs/guides/blocking-vs-non-blocking/
- Async JavaScript: From Callbacks, to Promises, to Async/Await - Tyler McGinnis