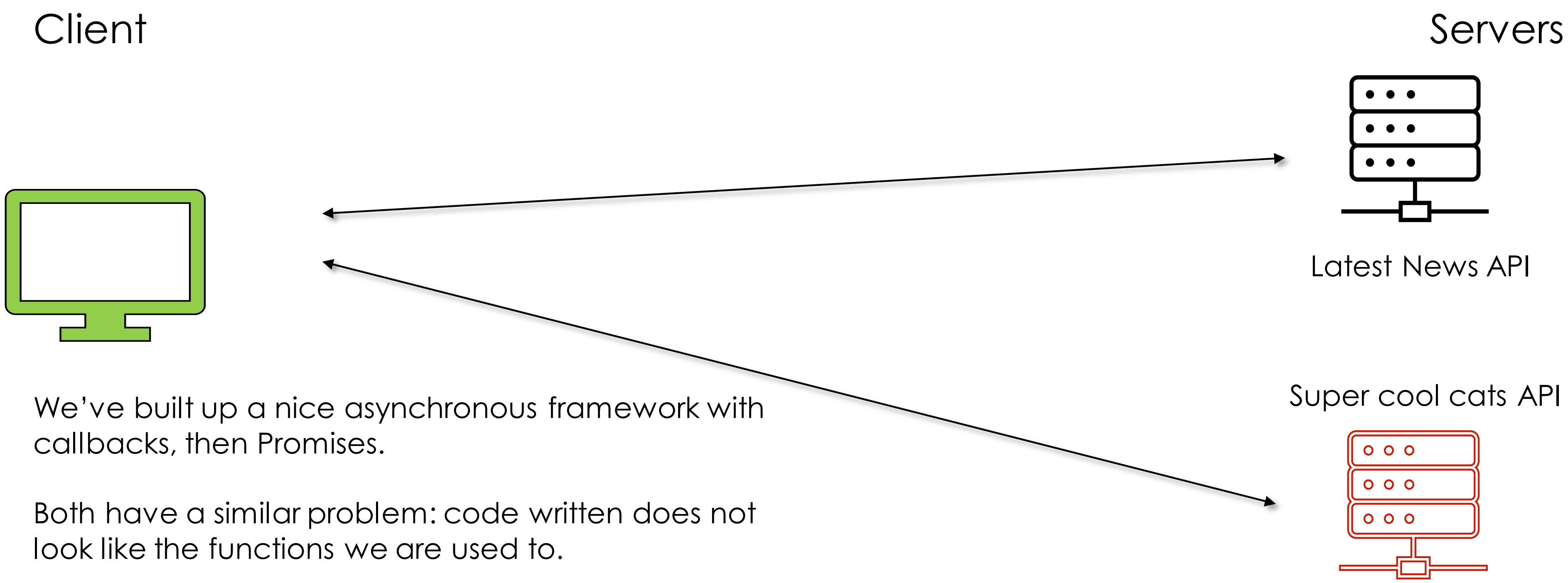
ASYNCHRONOUS NETWORKING

Networking with `async/await & fetch()`

OVERVIEW

- Client-Server Model + AJAX
- Concurrency & JS
- Networking with `XMLHttpRequest()`
- Networking with Promises & `fetch()`
- Networking with `async/await & fetch()`

RECAP

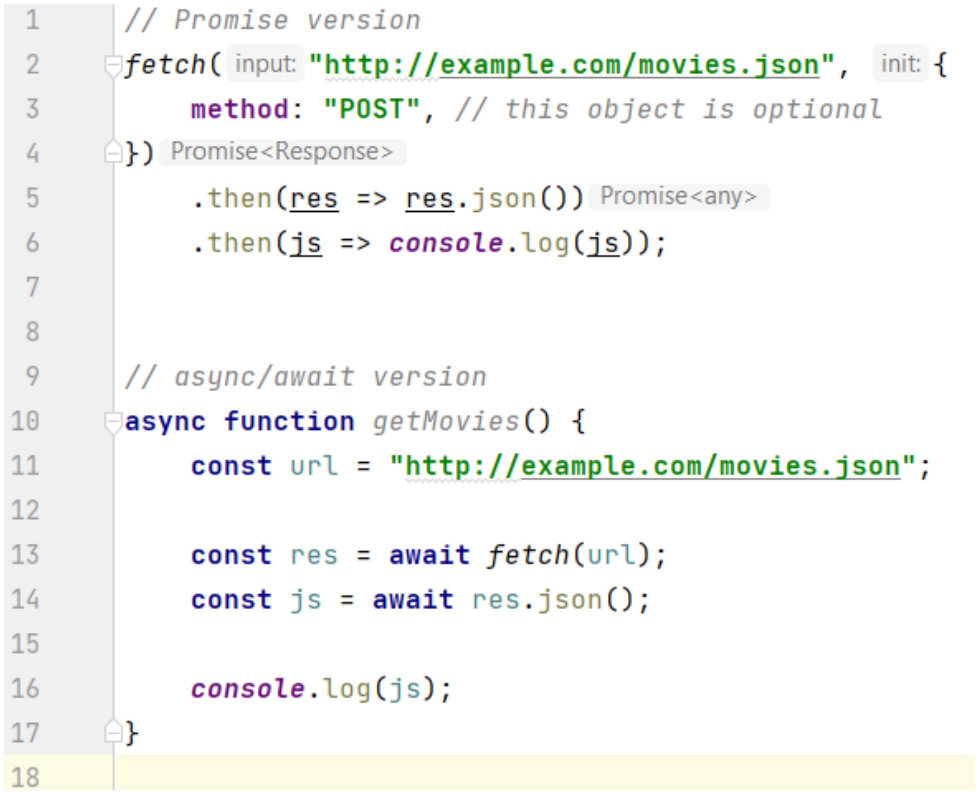


We've built up a nice asynchronous framework with callbacks, then Promises.

Both have a similar problem: code written does not look like the functions we are used to.

That's where `async/await` come in

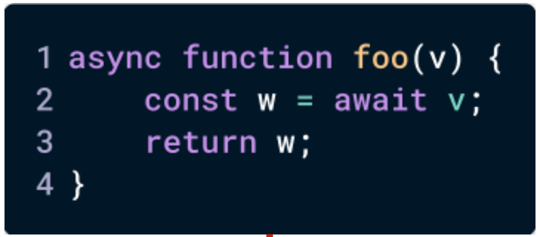
ASYNC/AWAIT



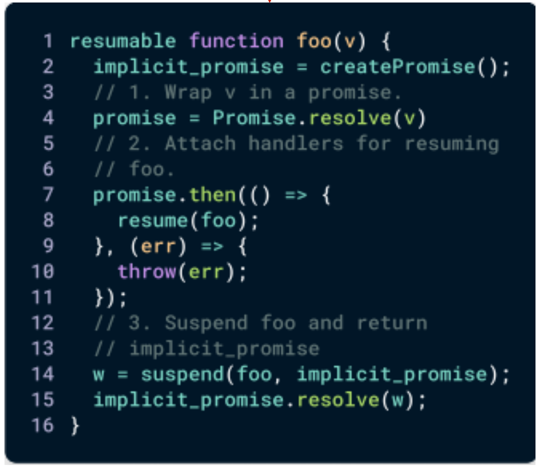
```
// Promise version
fetch( input: "http://example.com/movies.json",  init: {
    method: "POST", // this object is optional
}) Promise<Response>
    .then(res => res.json()) Promise<any>
    .then(js => console.log(js));


// async/await version
async function getMovies() {
    const url = "http://example.com/movies.json";

    const res = await fetch(url);
    const js = await res.json();

    console.log(js);
}

```

- Keywords introduced in ES2017
  - Supported by virtually all browsers
  - So-called "coroutines" i.e. resumable functions
- Built on top of Promises
- `async` functions always return a Promise
- `await` expressions always unwrap a resolved Promise
- Rejections are thrown as exceptions
- Program asynchronous code in a familiar and intuitive way again

UNDER THE HOOD
AN INTUITIVE UNDERSTANDING



```
async function foo(v) {
    const w = await v;
    return w;
}
```



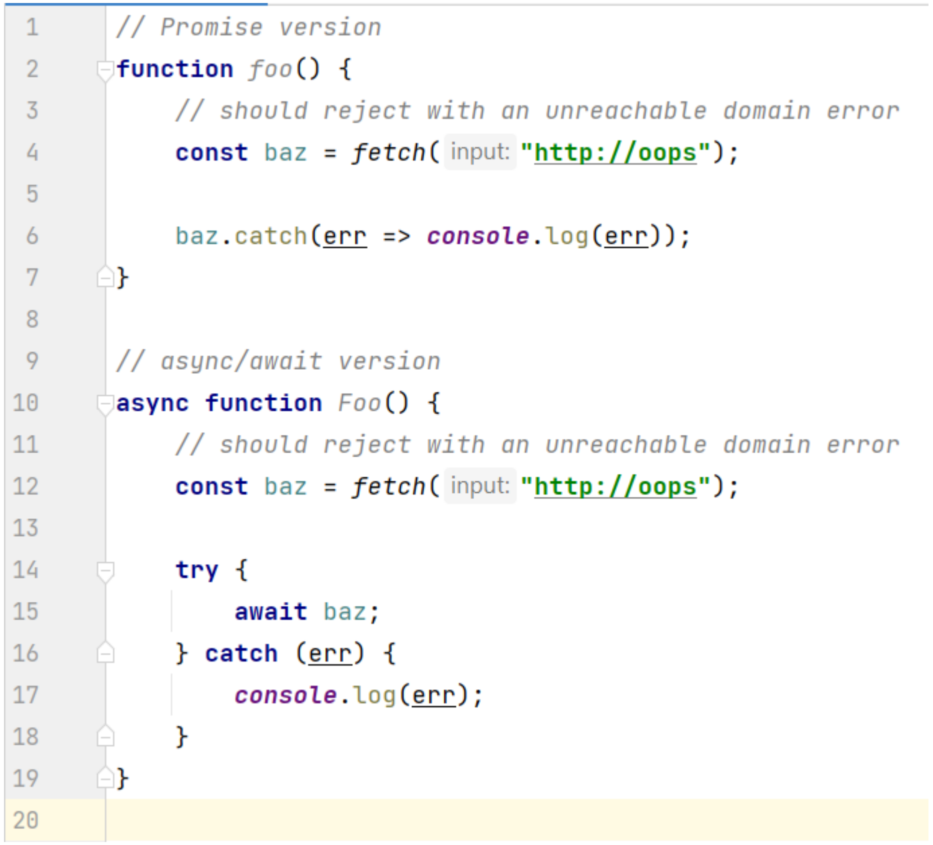
```
resumable function foo(v) {
  implicit_promise = createPromise();
  // 1. Wrap v in a promise.
  promise = Promise.resolve(v)
  // 2. Attach handlers for resuming
  // foo.
  promise.then(() => {
    resume(foo);
  }, (err) => {
    throw(err);
  });
  // 3. Suspend foo and return
  // implicit_promise
  w = suspend(foo, implicit_promise);
  implicit_promise.resolve(w);
}
```

Image Credit: Zain Afzal

N.B. It is *as-if* this is what happens. This is not real code.

- Non-promise values/thenables are awaitable like promises
- v always wrapped into a Promise
- This Promise then chained with injected logic to resume the function at the point where `await` was used
- JS runtime suspends execution of the function whilst returning a Promise to the caller that represents the suspended function's eventual result
- The returned Promise resolves once the suspended function is resumed and reaches the end of its body

ERROR-HANDLING REVISITED



```
// Promise version
function foo() {
    // should reject with an unreachable domain error
    const baz = fetch( input: "http://oops");

    baz.catch(err => console.log(err));
}

// async/await version
async function Foo() {
    // should reject with an unreachable domain error
    const baz = fetch( input: "http://oops");

    try {
        await baz;
    } catch (err) {
        console.log(err);
    }
}

```

- `async/await` functions always maintain their function scope
- Errors and rejections handled by `try/catch` again
- Careful:
  - `await`'ing a Promise will cause an exception
  - `async` functions called synchronously will still return a pending Promise

ASYNC/AWAIT FETCH DEMO

See examples/asyncawait-fetch

PROMISES OR ASYNC/AWAIT?

Promise

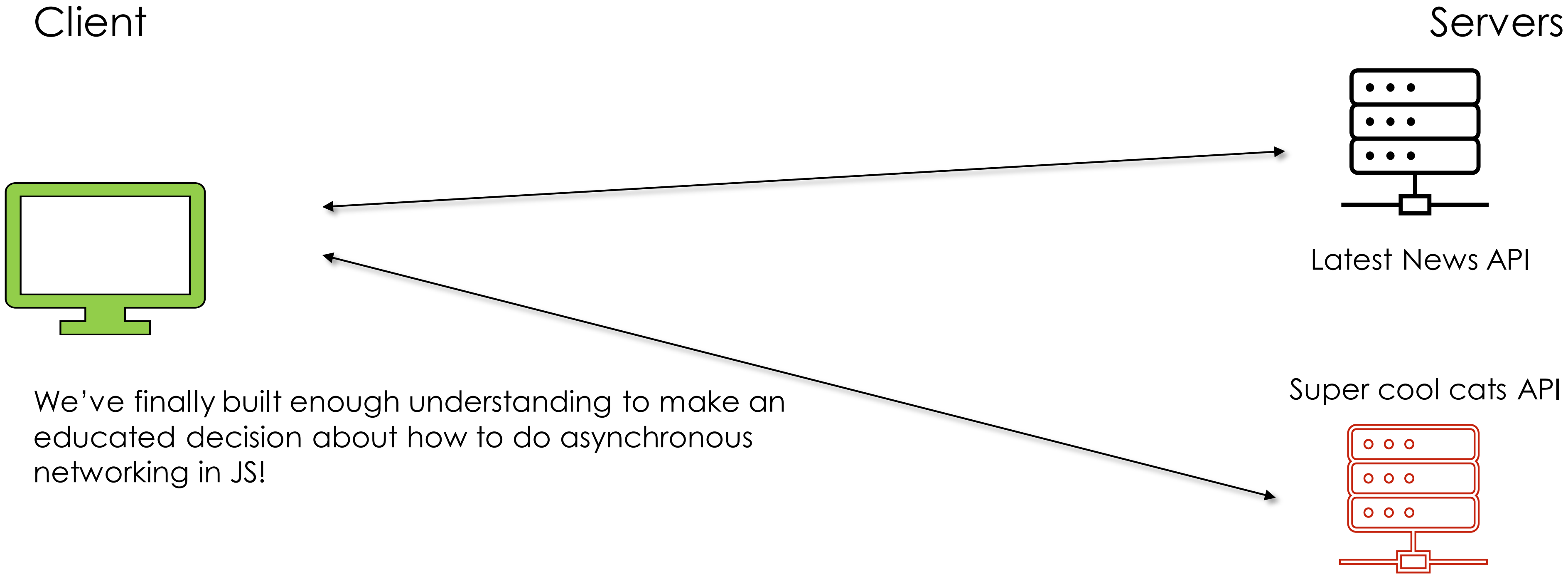
- Can pass around and attach different handlers
- Maintains a stack trace so takes more memory
- Supported more on older browsers

`async/await`

- Easier to read
- Potentially harder to debug
  - Debuggers will jump around source code
- More performant in some cases
- More familiar

Ultimately, which to use comes down to your specific needs / environment / targets, etc.

FINALLY!



We've finally built enough understanding to make an educated decision about how to do asynchronous networking in JS!

SUMMARY

- Today:
  - `async/await`
  - Using `async/await` with `fetch()`

- Asynchronous networking with Javascript accomplished by:
  - `XMLHttpRequest` (legacy)
  - `fetch()` and Promises
  - `fetch()` and `async/await`

- AJAX technologies continue to evolve

- But the fundamentals rarely change