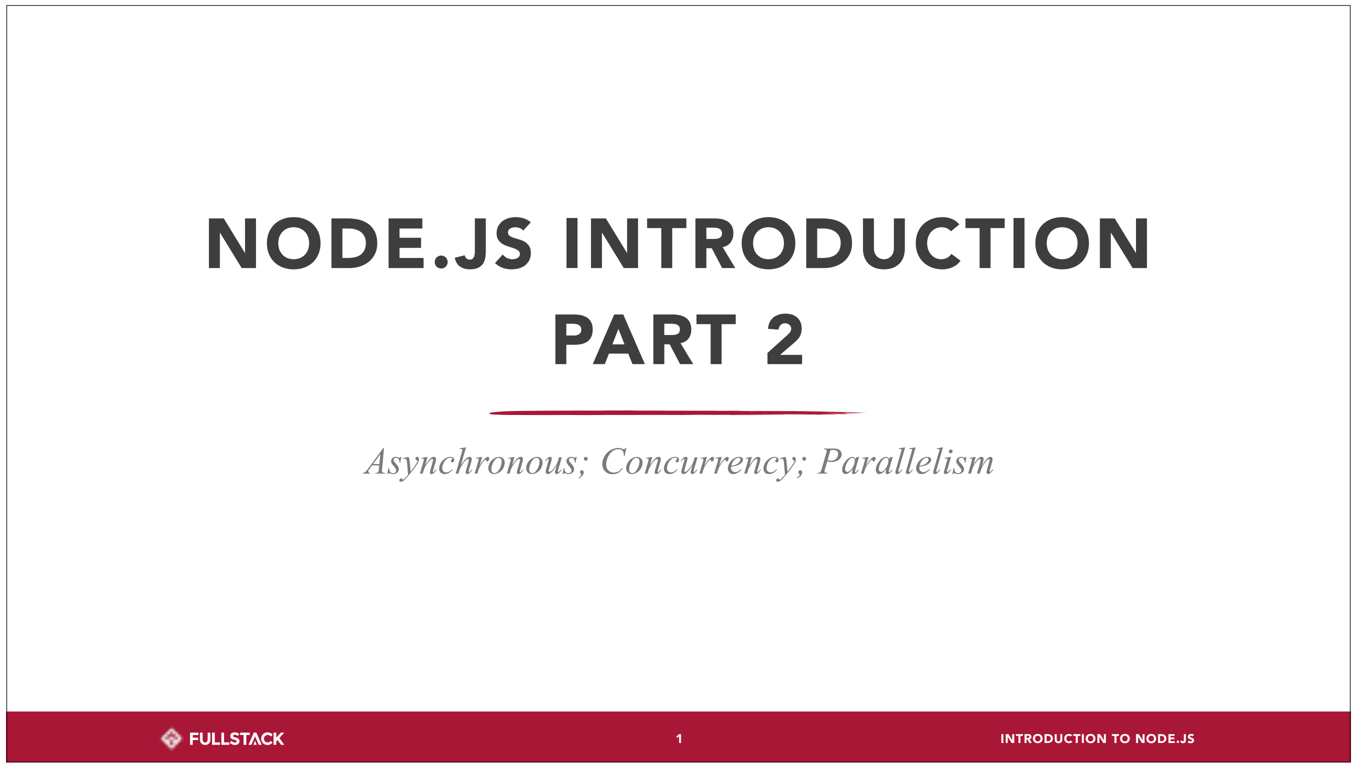


NODE.JS INTRODUCTION PART 2

Asynchronous; Concurrency; Parallelism

In this second part of the node introduction, I want to focus on node.js's very unique place in the world: its characteristic as an asynchronous platform.

RELEASED MAY 27, 2009

node was released to the world in 2009 to a very polarized software community.

Some embraced node.js—mostly people already embedded in front-end Javascript programming—excited to be able to start writing programs that interacted with the operating system, rather than being confined to the browser.

On the other hand, some people approached node with lots of vitriol, citing Javascript as a poor language for serious software and node.js being a poorly implemented idea and, ultimately, a passing fad.

Google NETFLIX

PayPal™ Microsoft®

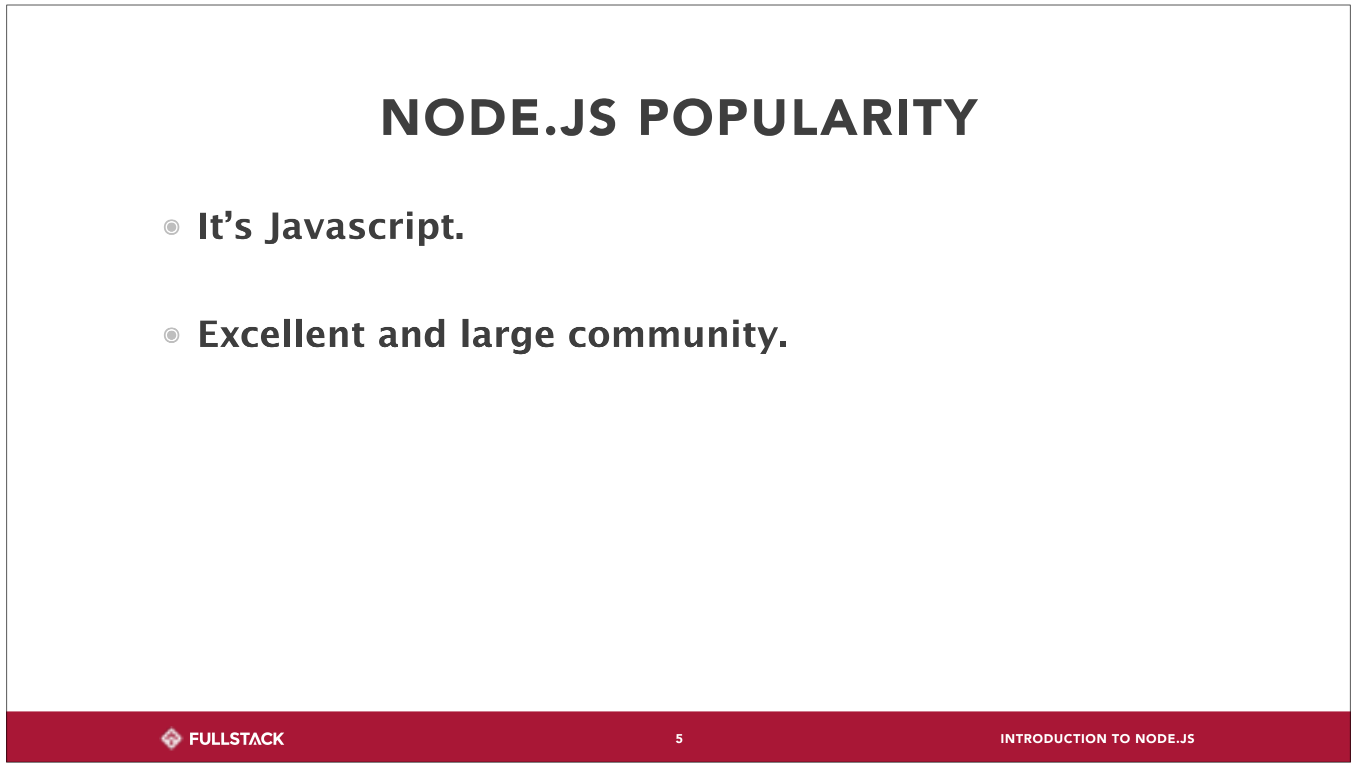
The New York Times Walmart

Almost 6 years later and node.js has proven itself to be important technology. It now powers major software components, including servers and other processes, at very high-profile companies. Here's a shortlist.

NODE.JS POPULARITY

- It’s Javascript.

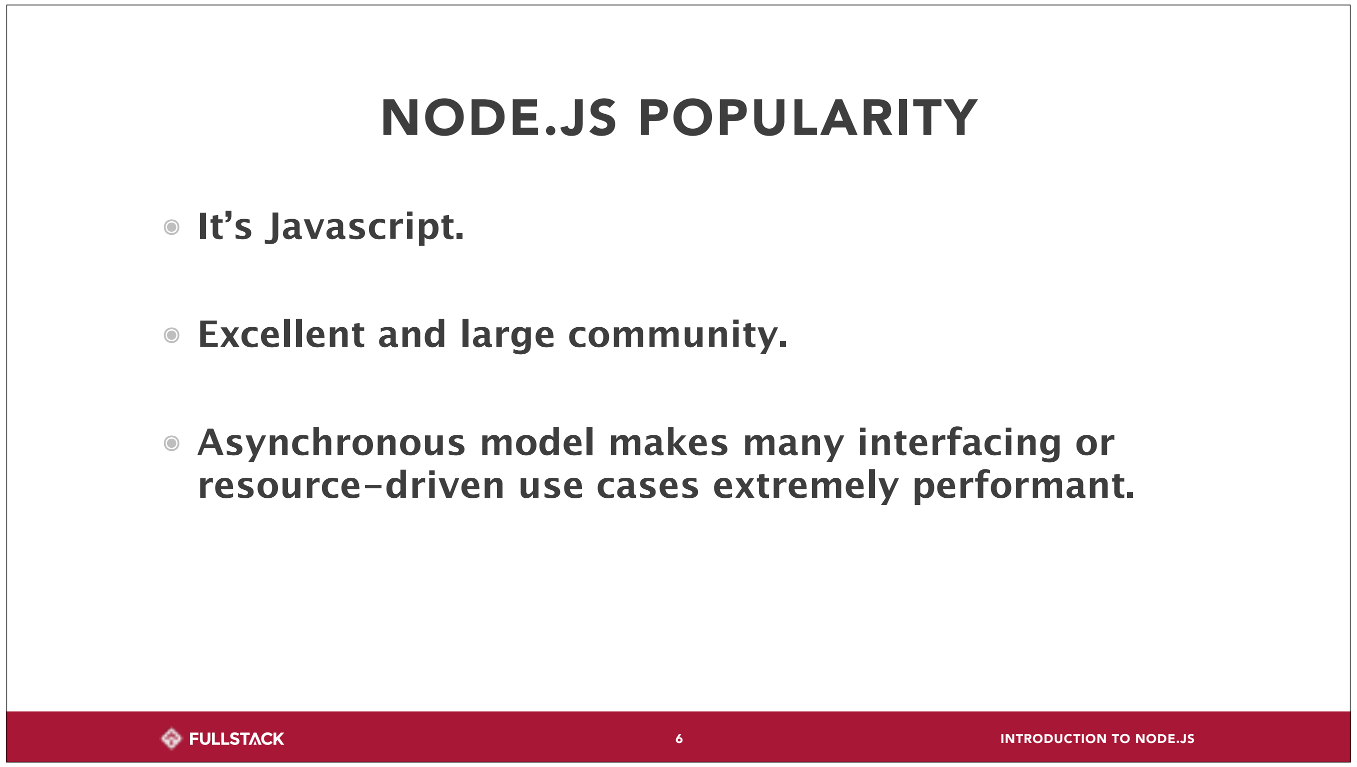
The first I attribute for this climb to popularity and implementation is that it’s Javascript. Many employees at this company were already writing great Javascript on the front-end — making Javascript the language on the server just made sense in order to reuse and focus that talent.



NODE.JS POPULARITY

- It's Javascript.
- Excellent and large community.

The node.js community is large and passionate about open-source, making npm the largest public repository of programming modules ever.



NODE.JS POPULARITY

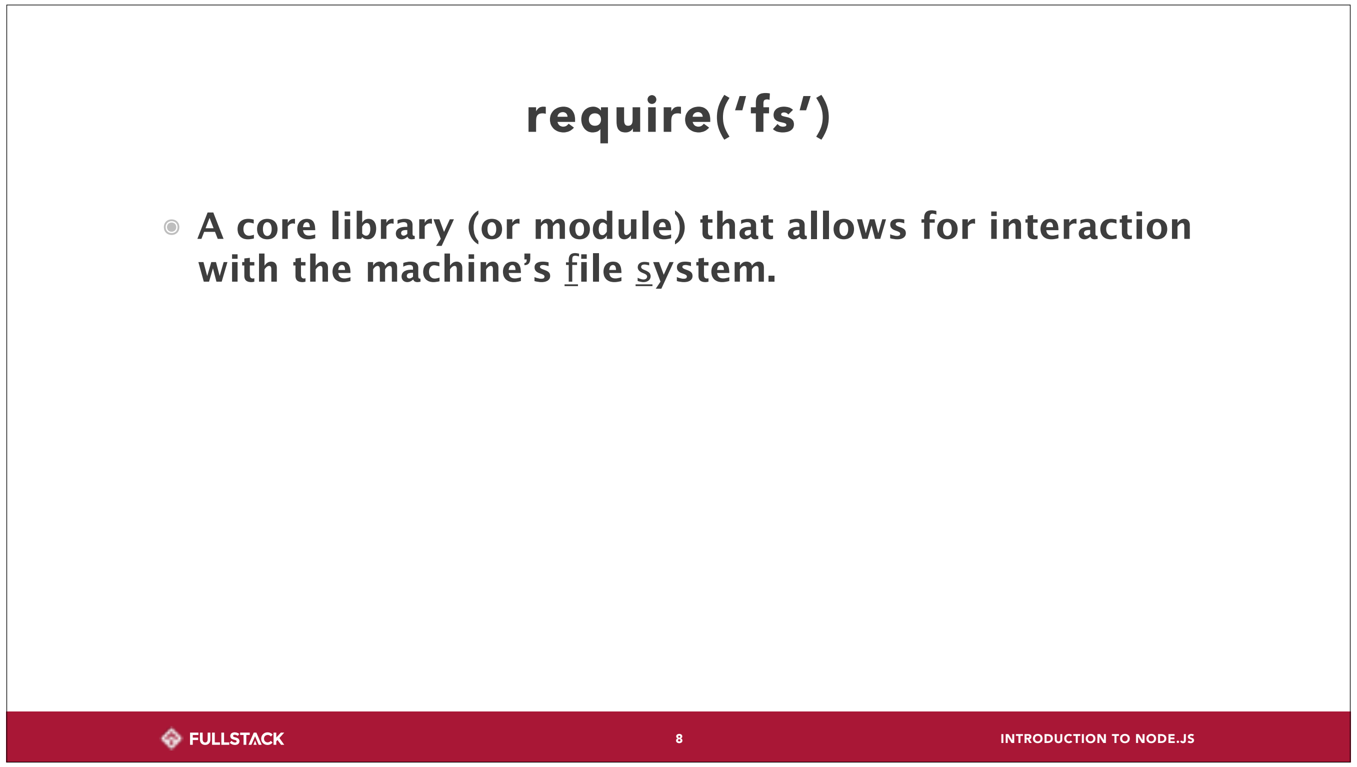
- It’s Javascript.
- Excellent and large community.
- Asynchronous model makes many interfacing or resource-driven use cases extremely performant.

And the third major reason, and the one I want to focus on in this lecture, is node’s unique characteristic as an asynchronous platform, which has shined in the current era of APIs and microservices.

fs

FILE SYSTEM

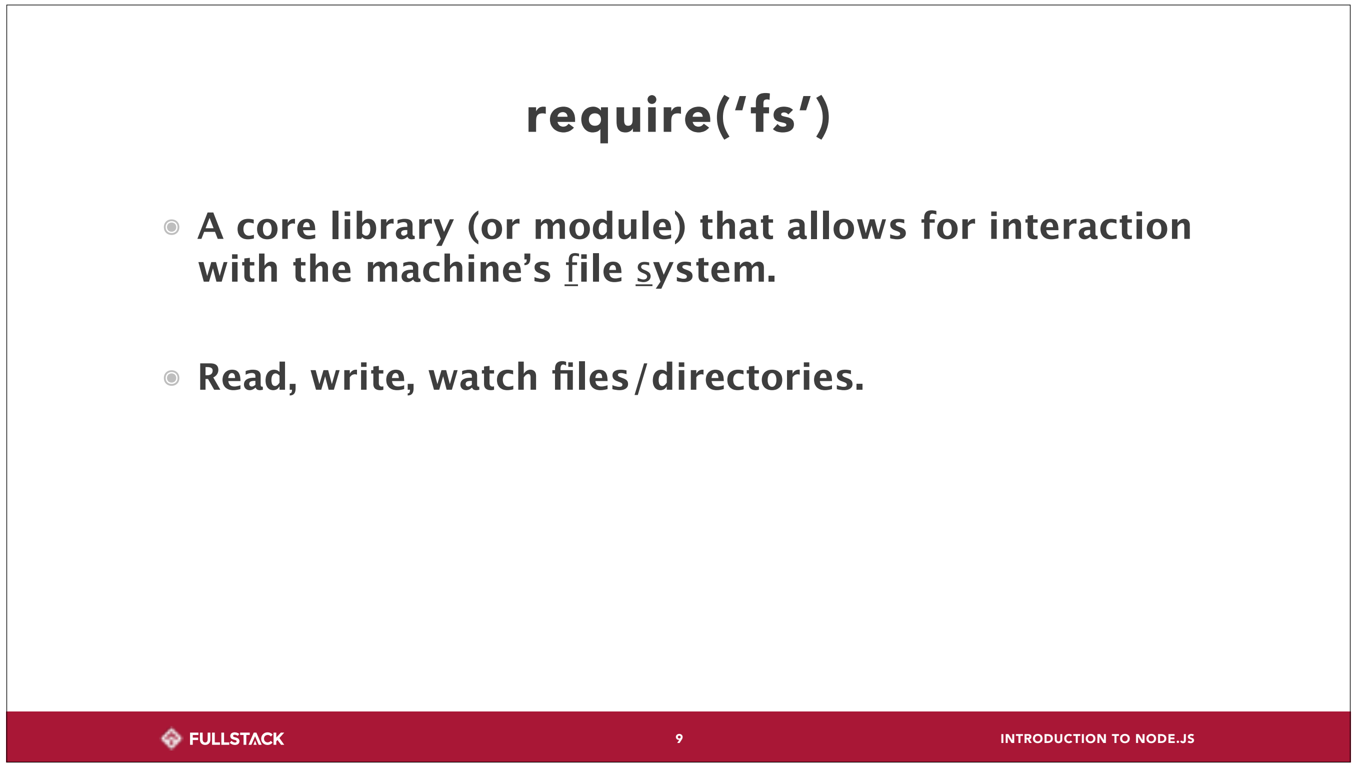
Let’s begin our discussion about the asynchronous characteristic with one of my favorite node.js core libraries: fs.



require('fs')

- A core library (or module) that allows for interaction with the machine's file system.

fs is a core library, built-in with node, that allows for interaction with the files of the machine the process is running on.

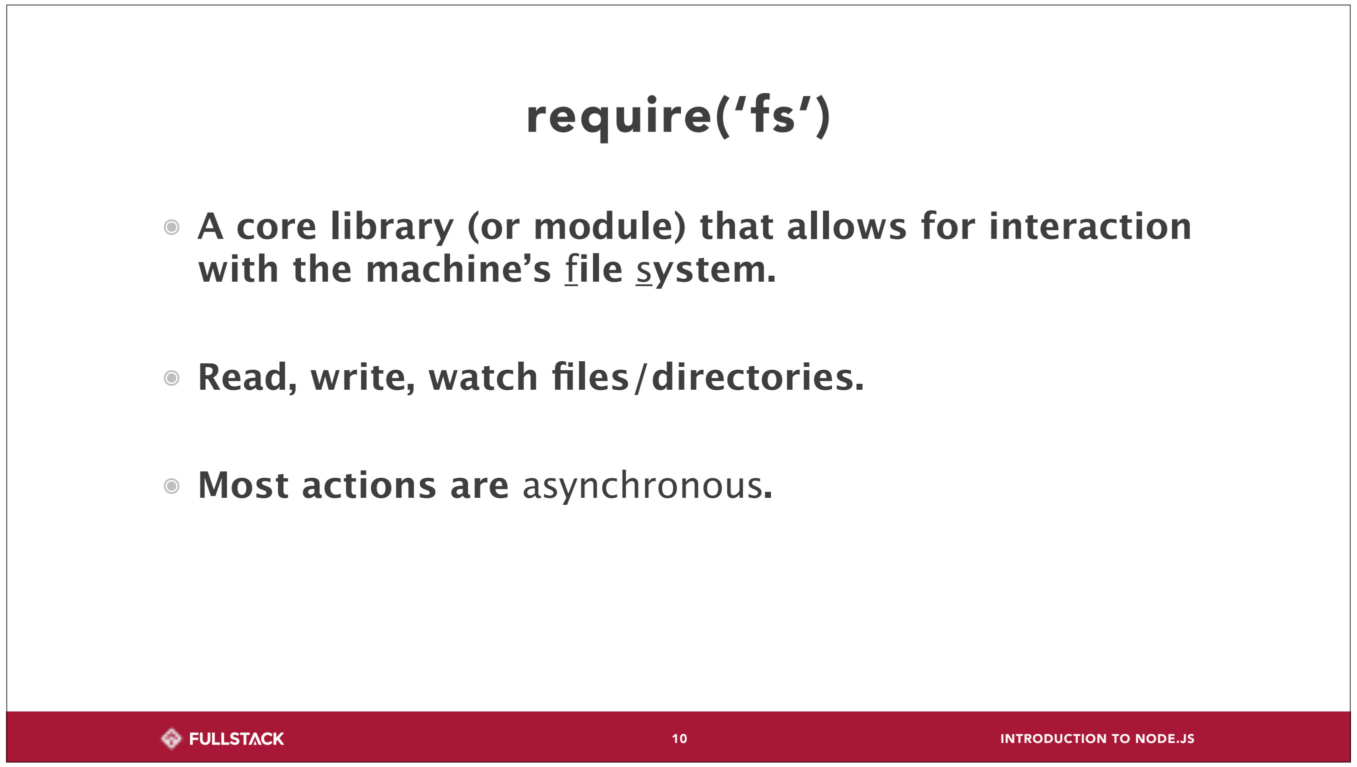


require('fs')

- A core library (or module) that allows for interaction with the machine's file system.
- Read, write, watch files/directories.

This includes reading, writing, watching, renaming files or directories on this machine.

This makes it possible to, for example, read in image files for image processing. Or you can read in CSVs for data. Anything that is a file can be read or written.



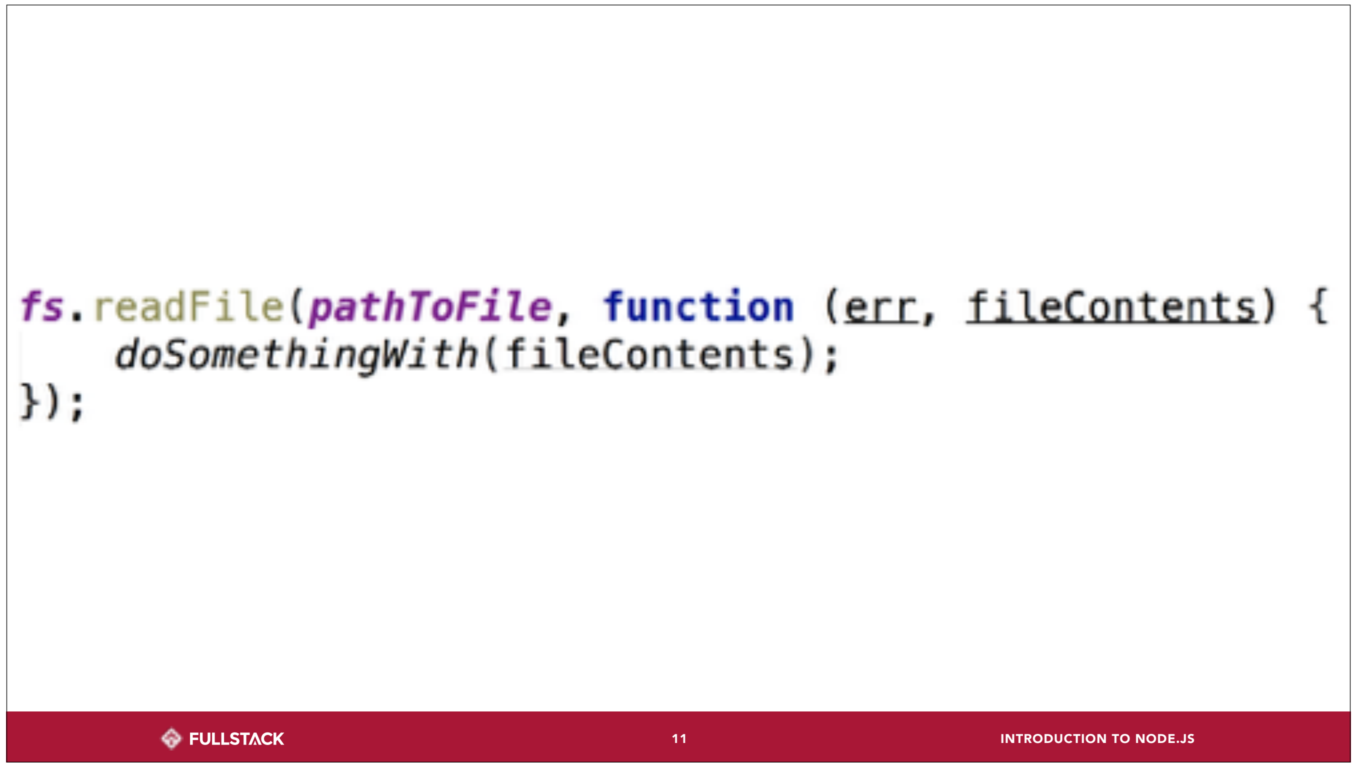
require(‘fs’)

- **A core library (or module) that allows for interaction with the machine’s <u>f</u>ile <u>s</u>ystem.**
- **Read, write, watch files/directories.**
- **Most actions are** asynchronous**.**

FULLSTACK 10 INTRODUCTION TO NODE.JS

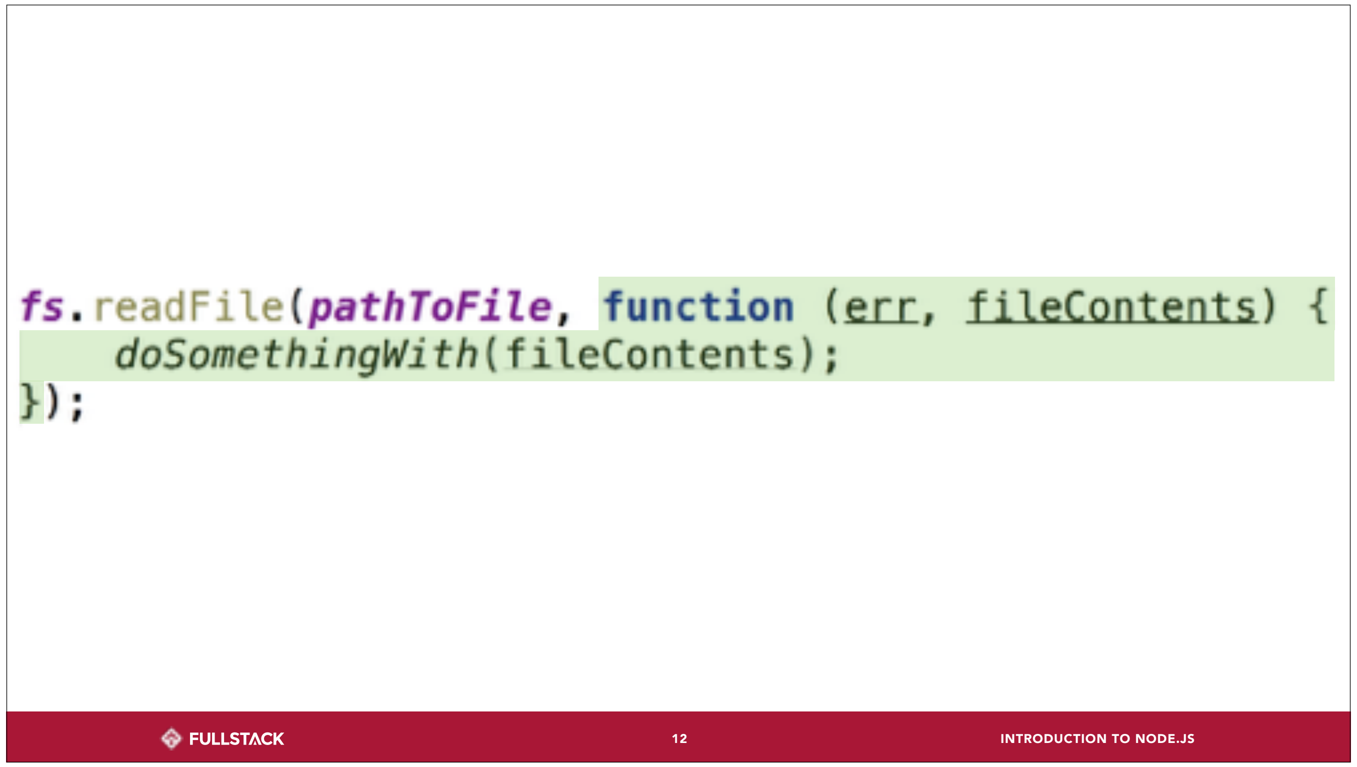
But one thing that makes fs really cool (and relevant) is that nearly every method you can call on this module is asynchronous.

[EXAMPLE WITH fs.readFile]



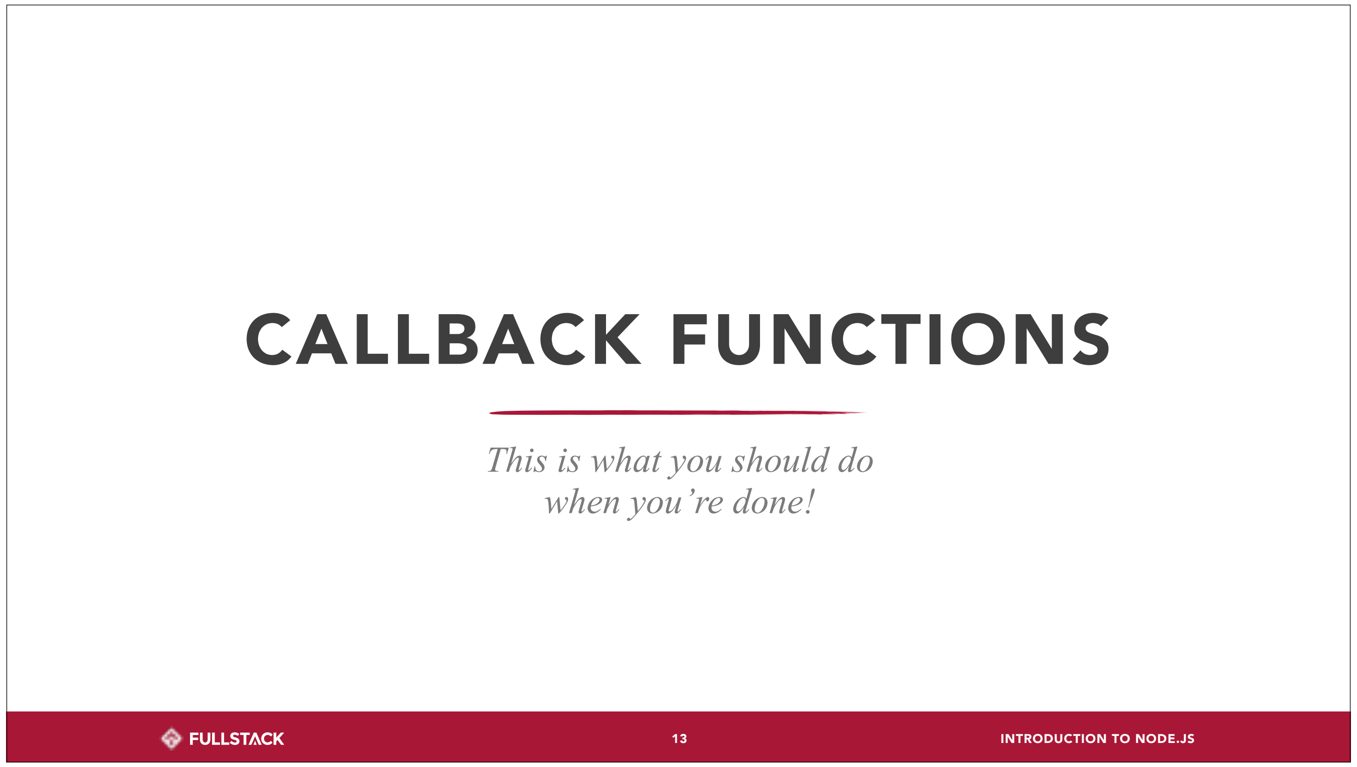
```
fs.readFile(pathToFile, function (err, fileContents) {
    doSomethingWith(fileContents);
});
```

So, as we've seen, these are the semantics. Read file, this is the path to the file, and when the file has been finally accessed (or not accessed), give me the file contents or the error if the access failed.



```
fs.readFile(pathToFile, function (err, fileContents) {
    doSomethingWith(fileContents);
});
```

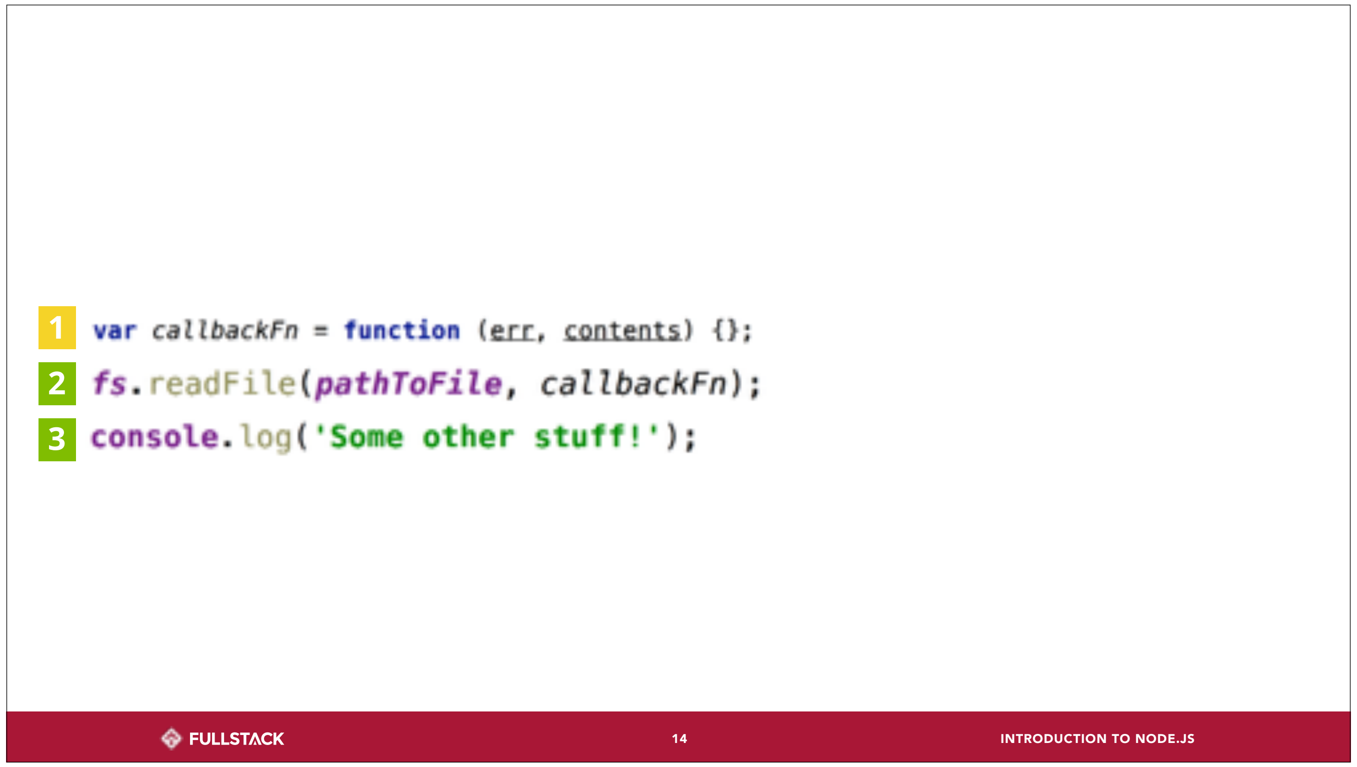
This second parameter, the function that receives our error or our file contents, is called . . .



CALLBACK FUNCTIONS

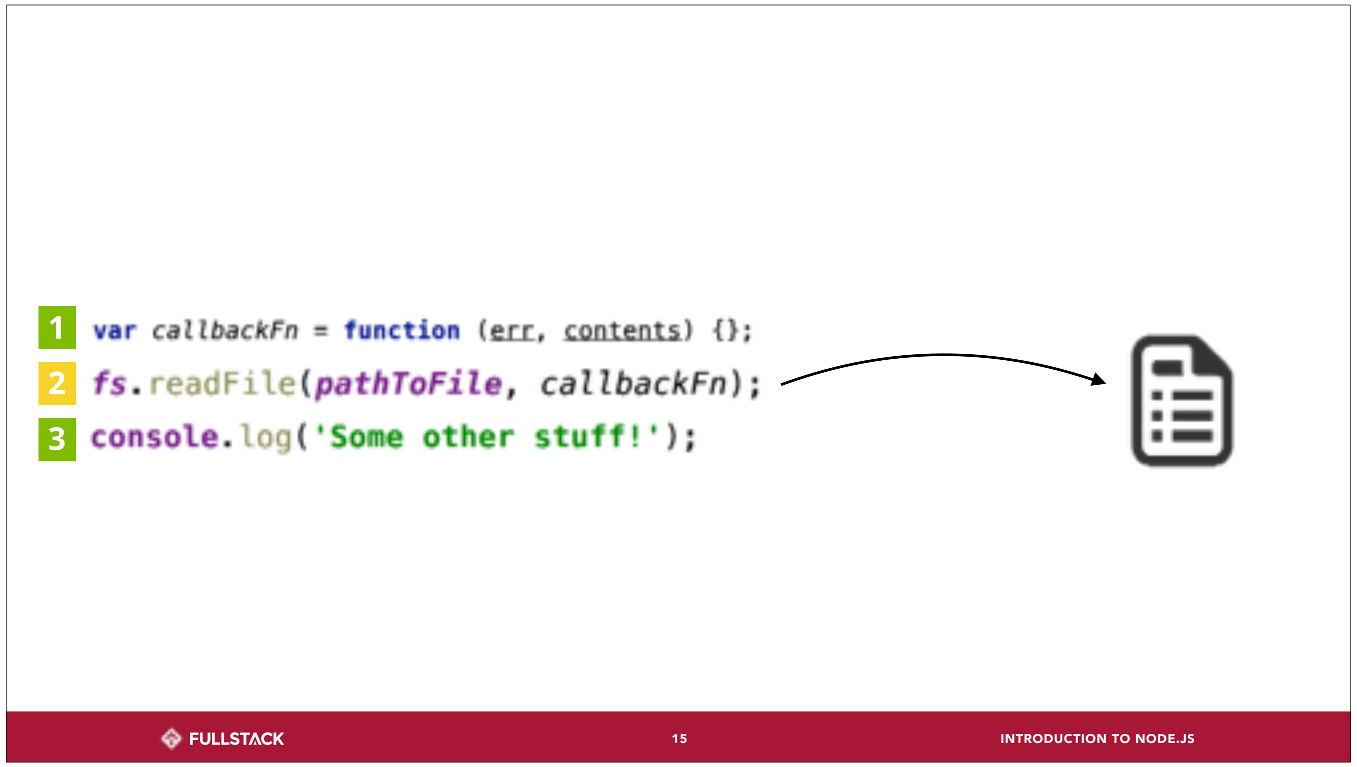
This is what you should do when you're done!

A callback function. Callback functions are the foundation of asynchronous operations in node.



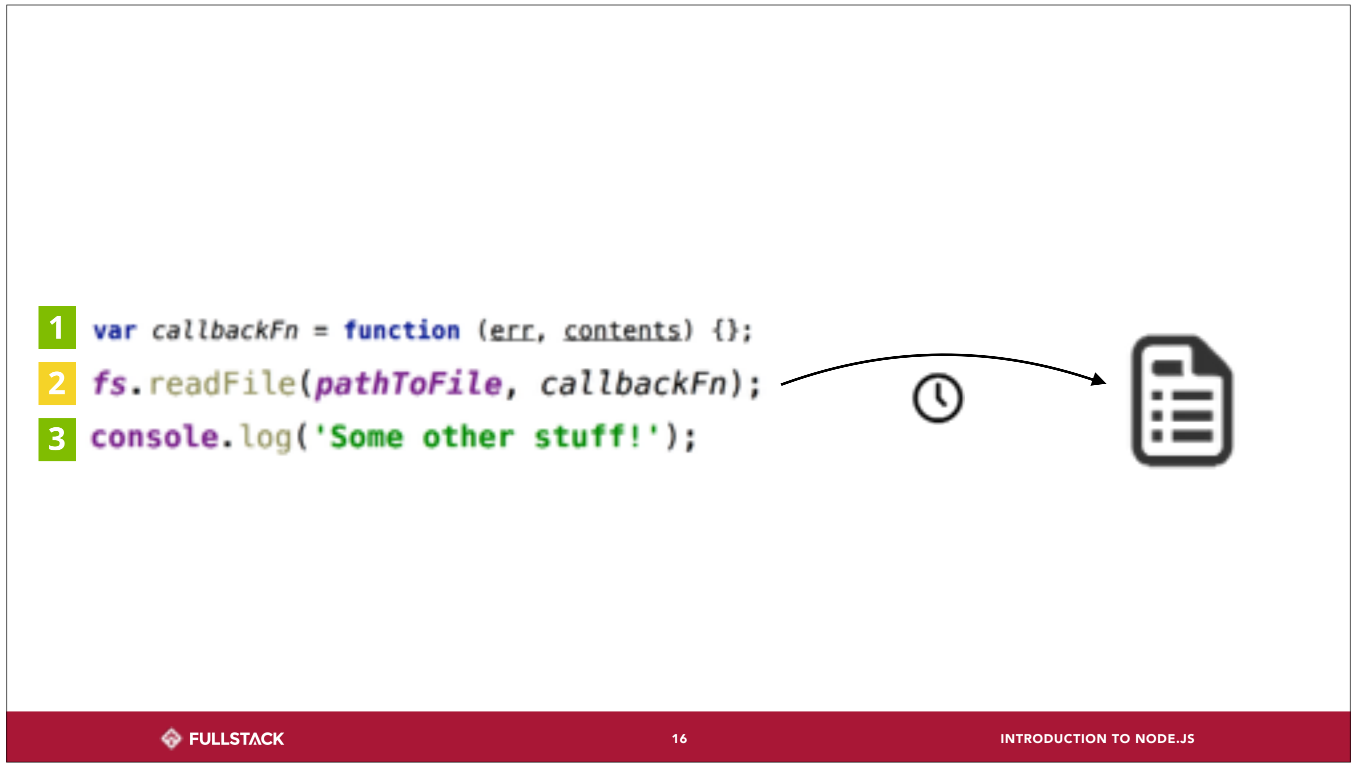
```
var callbackFn = function (err, contents) {};
fs.readFile(pathToFile, callbackFn);
console.log('Some other stuff!');
```

Here’s an example node program. We are going to run through the execution of this program, line-by-line. The line currently being evaluated is yellow, like our first line here. Right now, a function is being stored in the variable callbackFn.



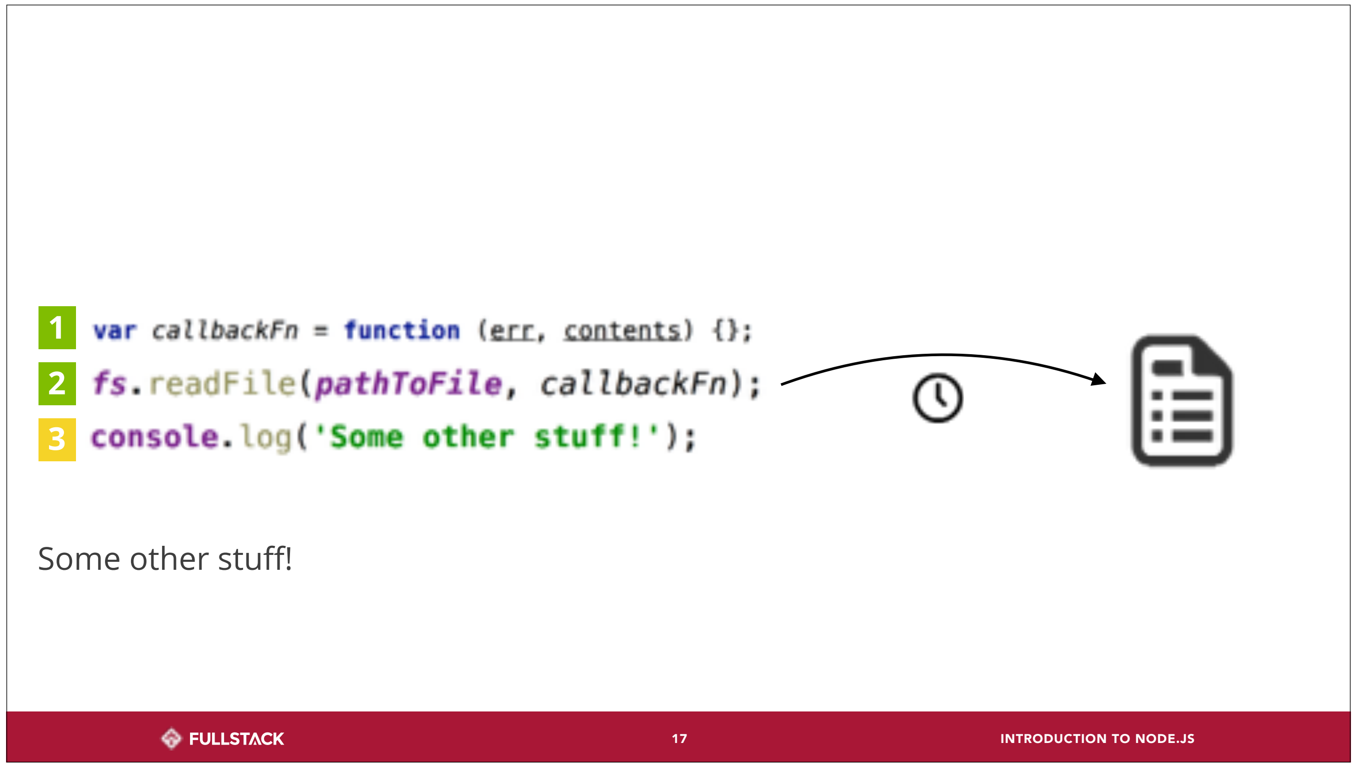
```
1 var callbackFn = function (err, contents) {};
2 fs.readFile(pathToFile, callbackFn);
3 console.log('Some other stuff!');
```

Next, the program evaluates line 2. fs.readFile. At this point, a background operation is initiated to go and fetch the file.



```
1 var callbackFn = function (err, contents) {};
2 fs.readFile(pathToFile, callbackFn);
3 console.log('Some other stuff!');
```

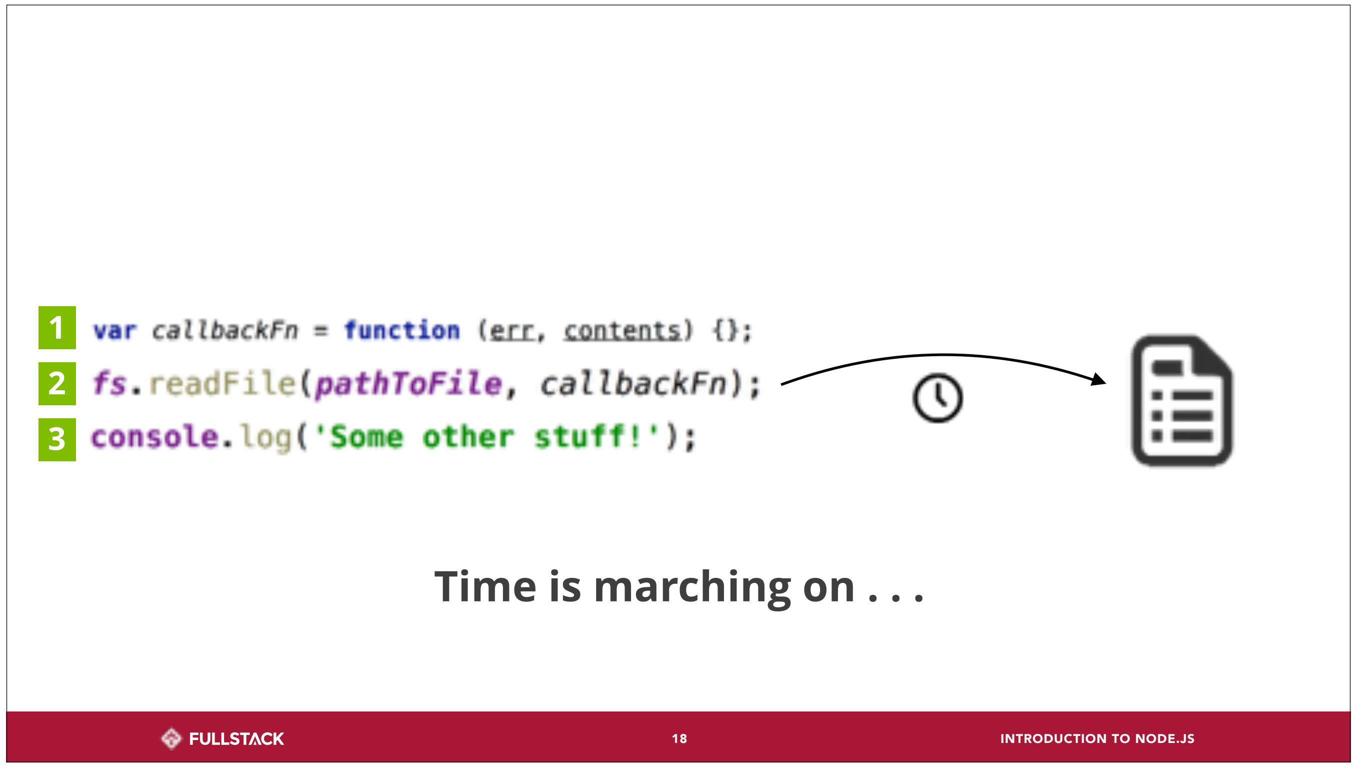
This will take some time. There is no assumption made about how fast or slow the hard drive on this machine is, or how large the file. It could take nanoseconds to read the file, it could take minutes. In any case, node allows the file access and read to happen in the background . . .



```
1 var callbackFn = function (err, contents) {};
2 fs.readFile(pathToFile, callbackFn);
3 console.log('Some other stuff!');
```

```
Some other stuff!
```

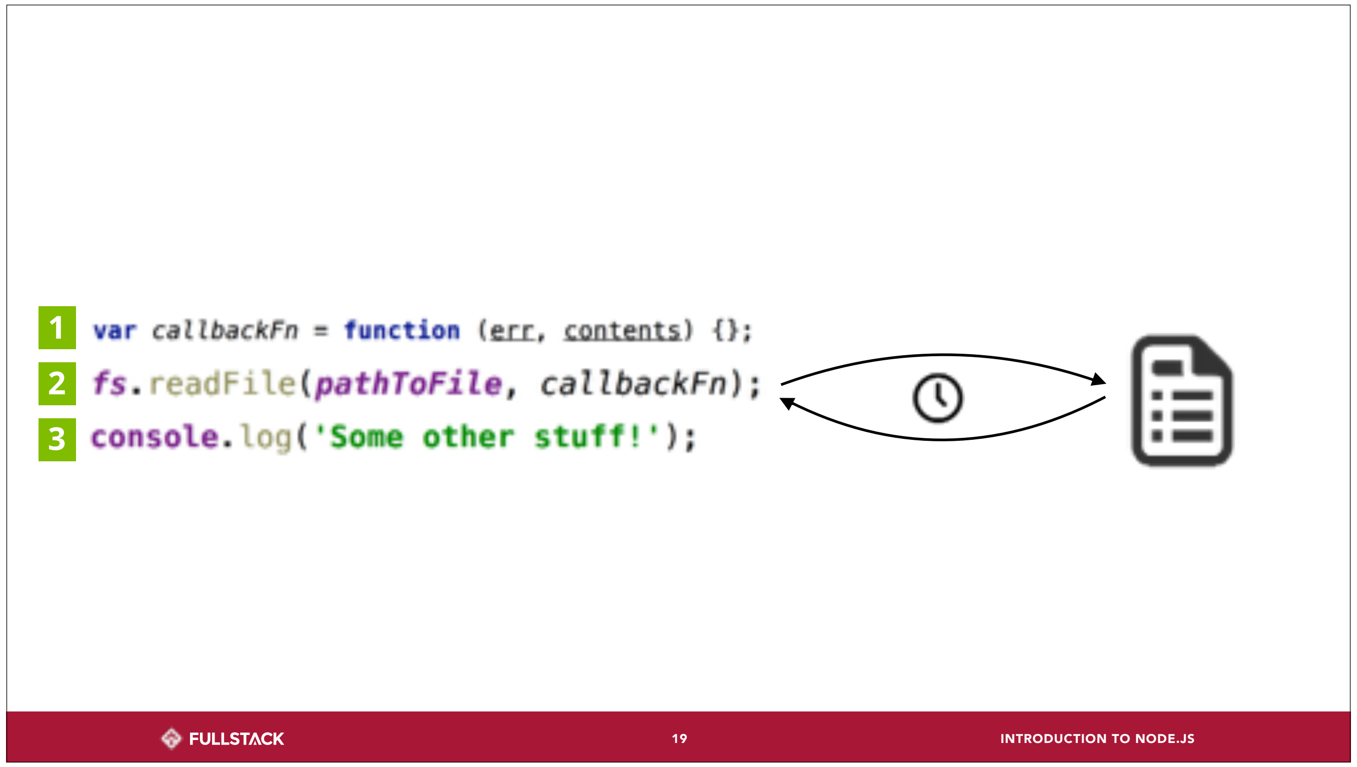
And continues on to line 3, where we evaluate a simple console log. This is very important. node works in a way that the program knows that the file read will take a while and just resumes program execution. It will continue this execution until there are no more lines to evaluate.



```
var callbackFn = function (err, contents) {};
fs.readFile(pathToFile, callbackFn);
console.log('Some other stuff!');
```

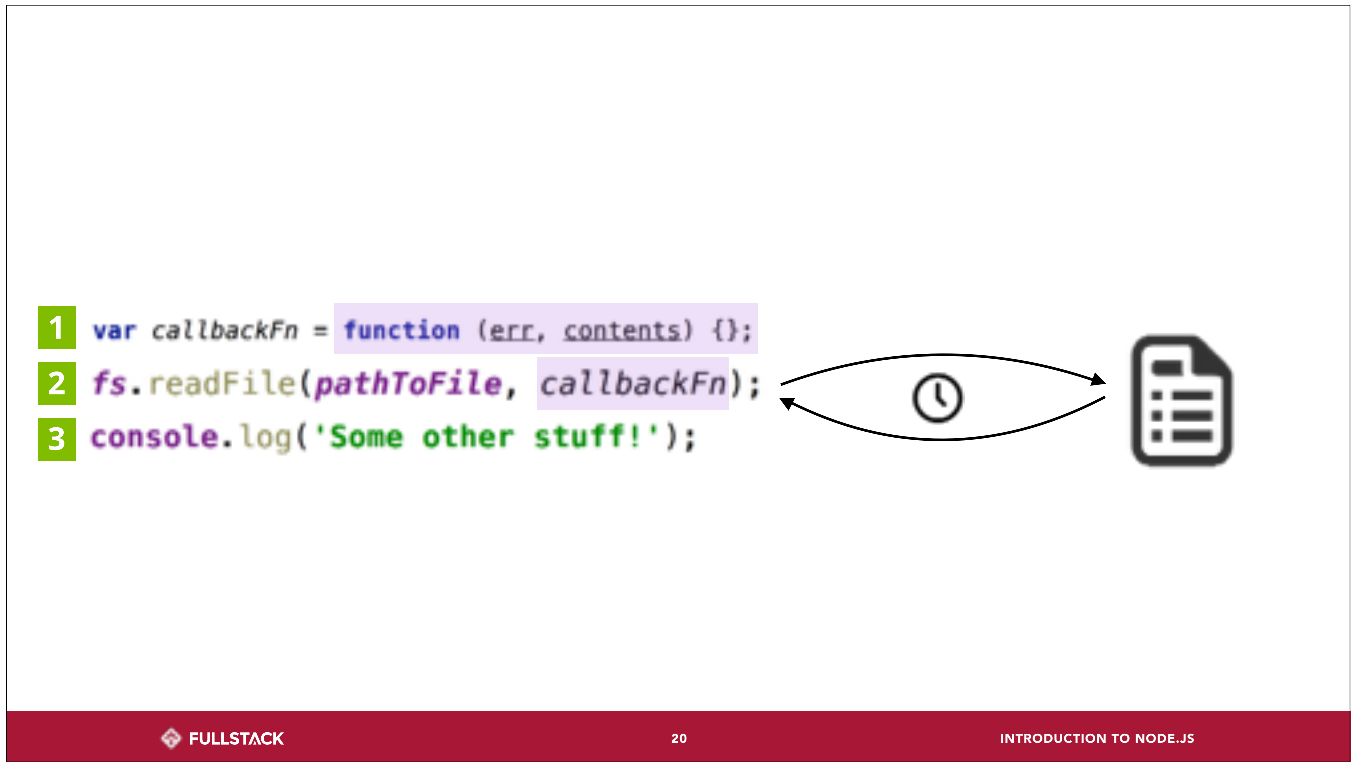
Time is marching on . . .

Even when it’s done evaluating lines, we may still have to wait.

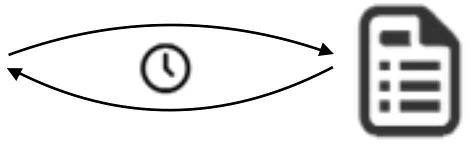


```
1 var callbackFn = function (err, contents) {};
2 fs.readFile(pathToFile, callbackFn);
3 console.log('Some other stuff!');
```

But finally, the background operation lets our program’s execution know that the file has been accessed and read . . .



```
1 var callbackFn = function (err, contents) {};
2 fs.readFile(pathToFile, callbackFn);
3 console.log('Some other stuff!');
```

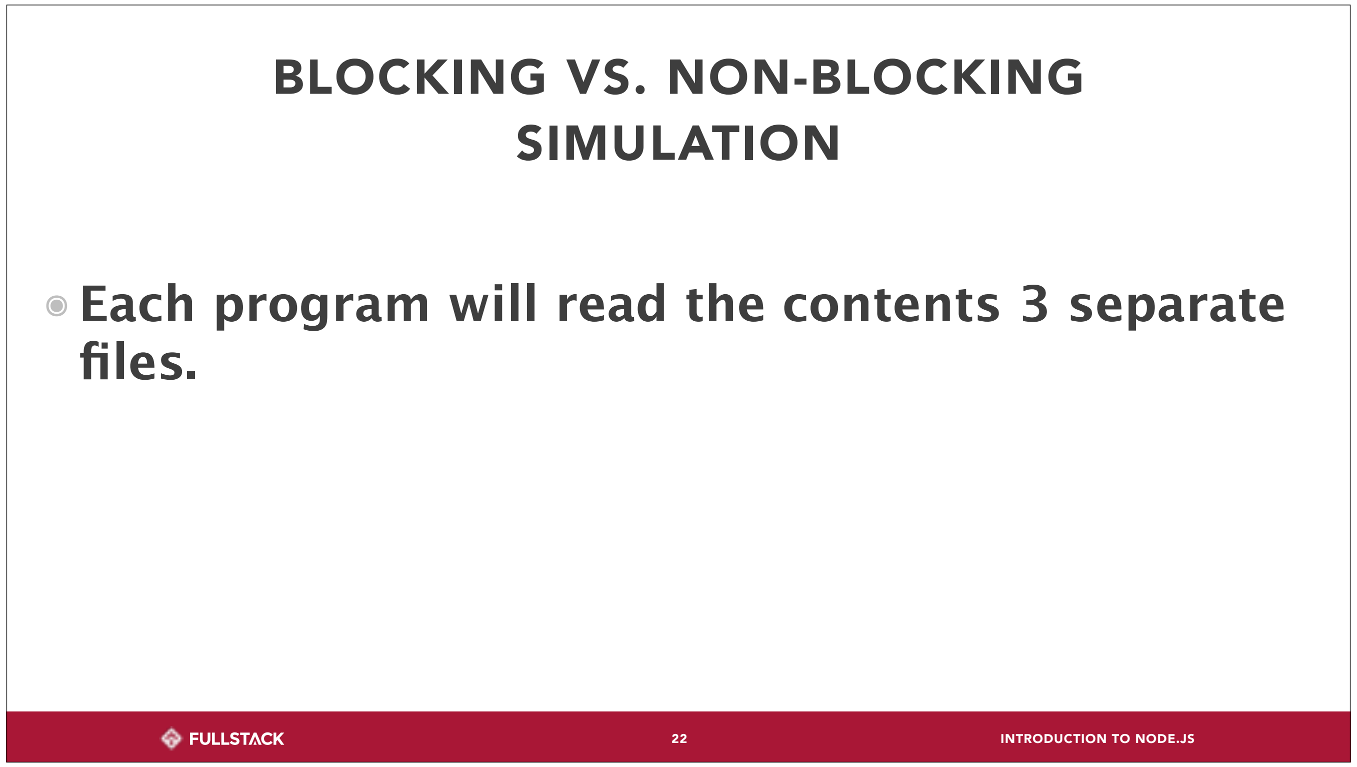


And calls our callback function with any possible error and the contents of the file.

[EXAMPLE]

NON-BLOCKING EXECUTION

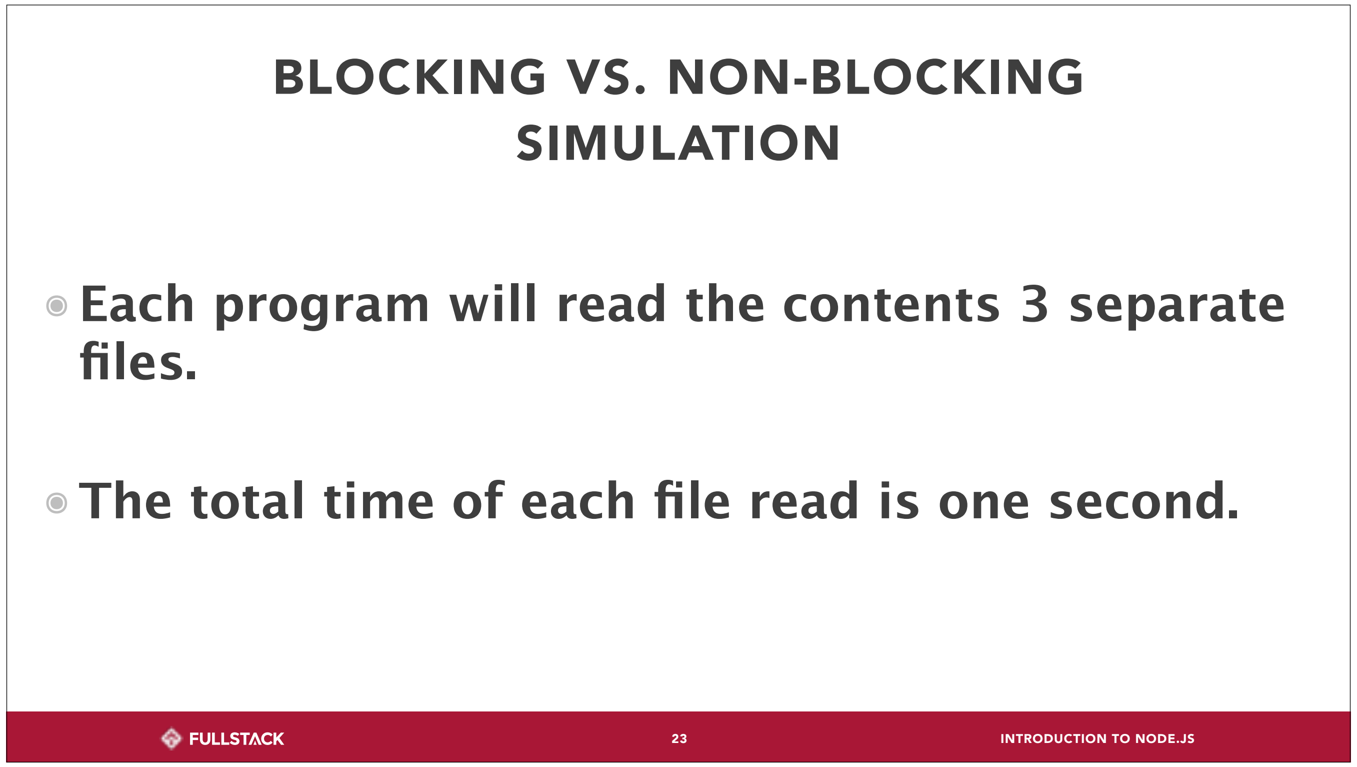
This model is called non-blocking execution. Think of blocking as the program pausing to wait for some work being done. Node is a non-blocking model.



BLOCKING VS. NON-BLOCKING SIMULATION

- Each program will read the contents 3 separate files.

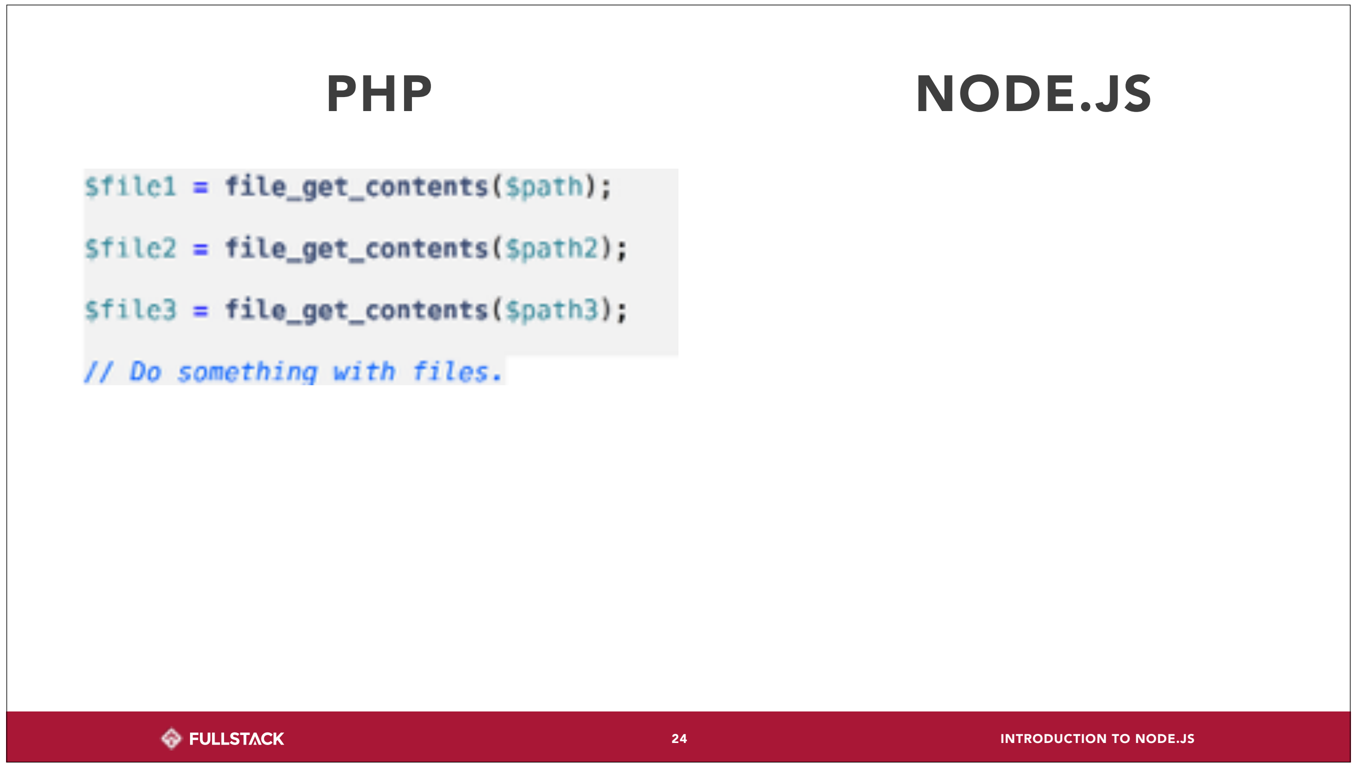
Let's run a bit of a simulation to see how these two models match up. In this simulation, each program is simply going to read the contents of 3 separate files.



BLOCKING VS. NON-BLOCKING SIMULATION

- Each program will read the contents 3 separate files.
- The total time of each file read is one second.

The total time to read each file, because the files are pretty large or we have a super old hard drive, is one second per file.



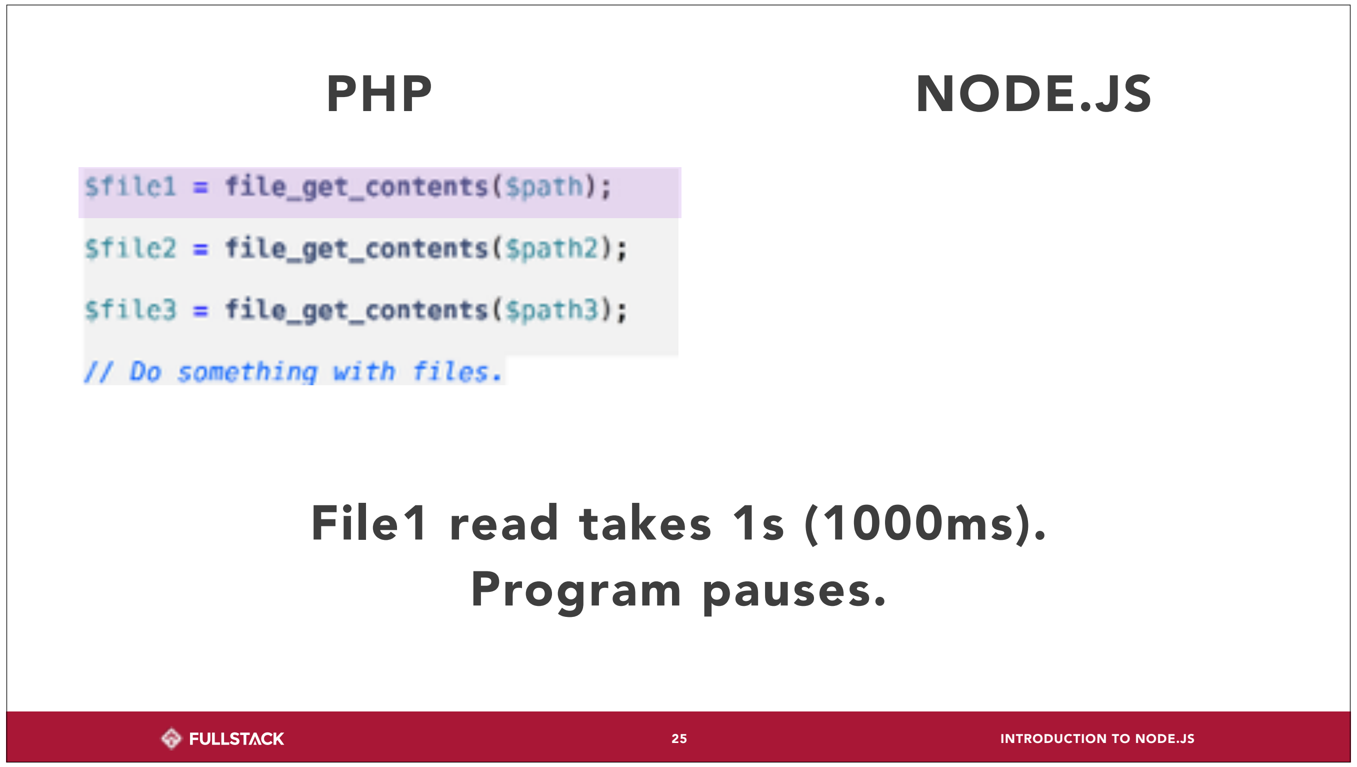
PHP

NODE.JS

```
$file1 = file_get_contents($path);
$file2 = file_get_contents($path2);
$file3 = file_get_contents($path3);
// Do something with files.
```

FULLSTACK 24 INTRODUCTION TO NODE.JS

Here’s the program in PHP.



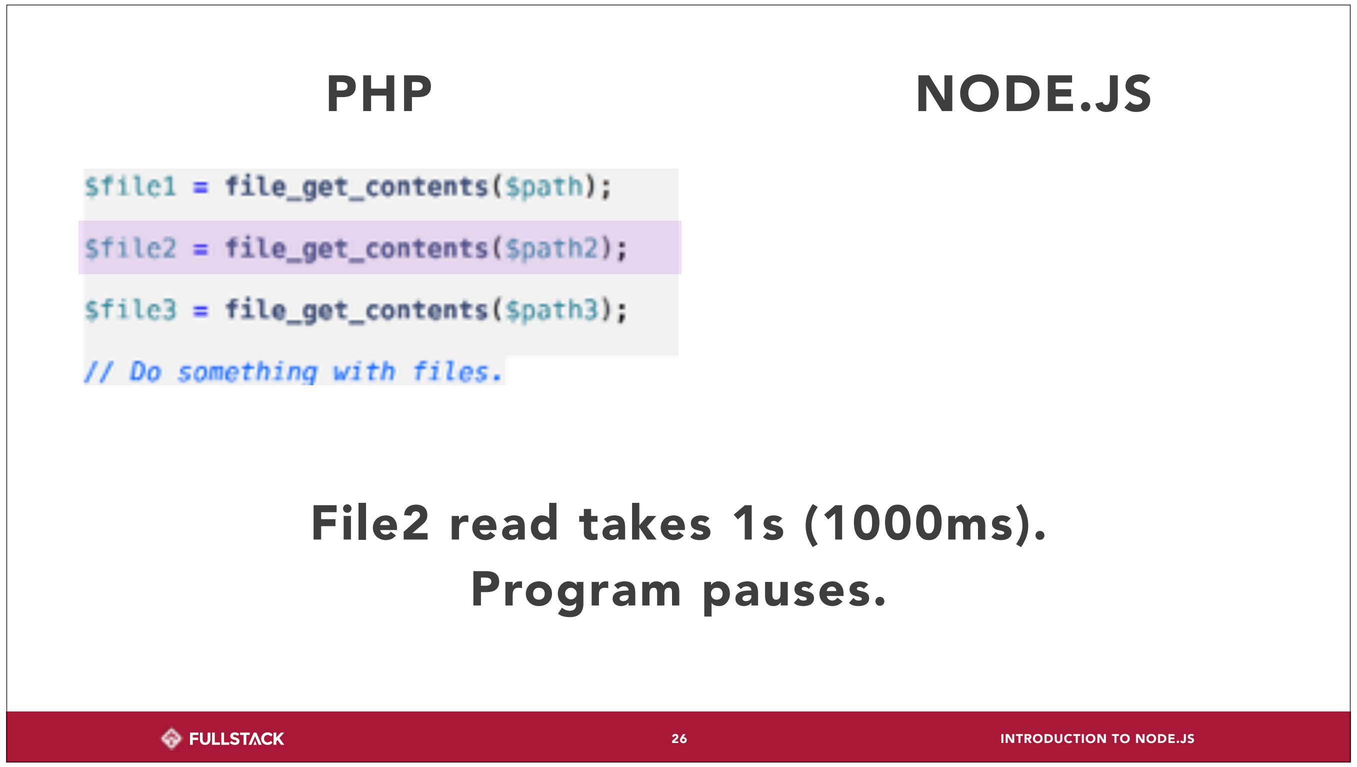
PHP

NODE.JS

```
$file1 = file_get_contents($path);
$file2 = file_get_contents($path2);
$file3 = file_get_contents($path3);
// Do something with files.
```

File1 read takes 1s (1000ms).
Program pauses.

We evaluate the first line, and the program pauses until the file read is completed.



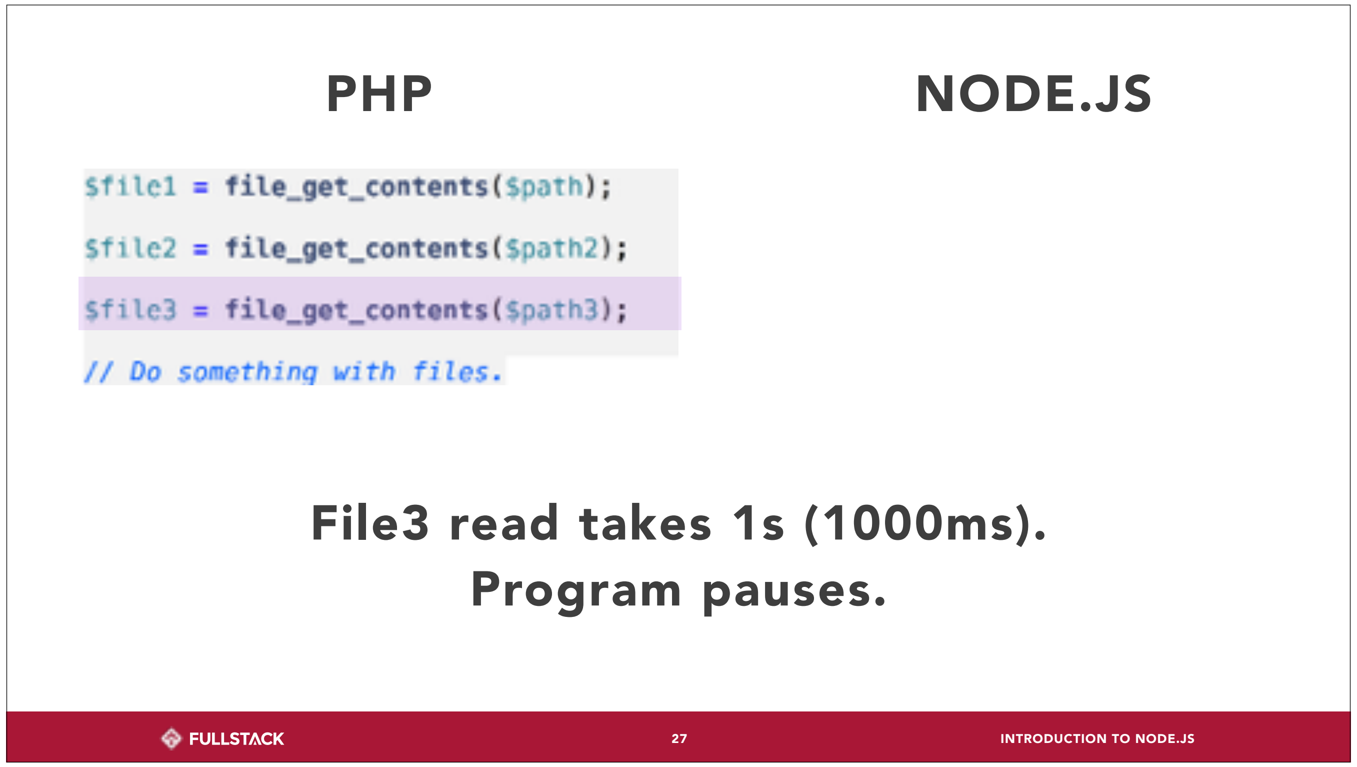
PHP

```
$file1 = file_get_contents($path);
$file2 = file_get_contents($path2);
$file3 = file_get_contents($path3);
// Do something with files.
```

NODE.JS

File2 read takes 1s (1000ms).
Program pauses.

Then, line 2 of the program is evaluated. Another file read is initiated, and the program pauses while the read completes.



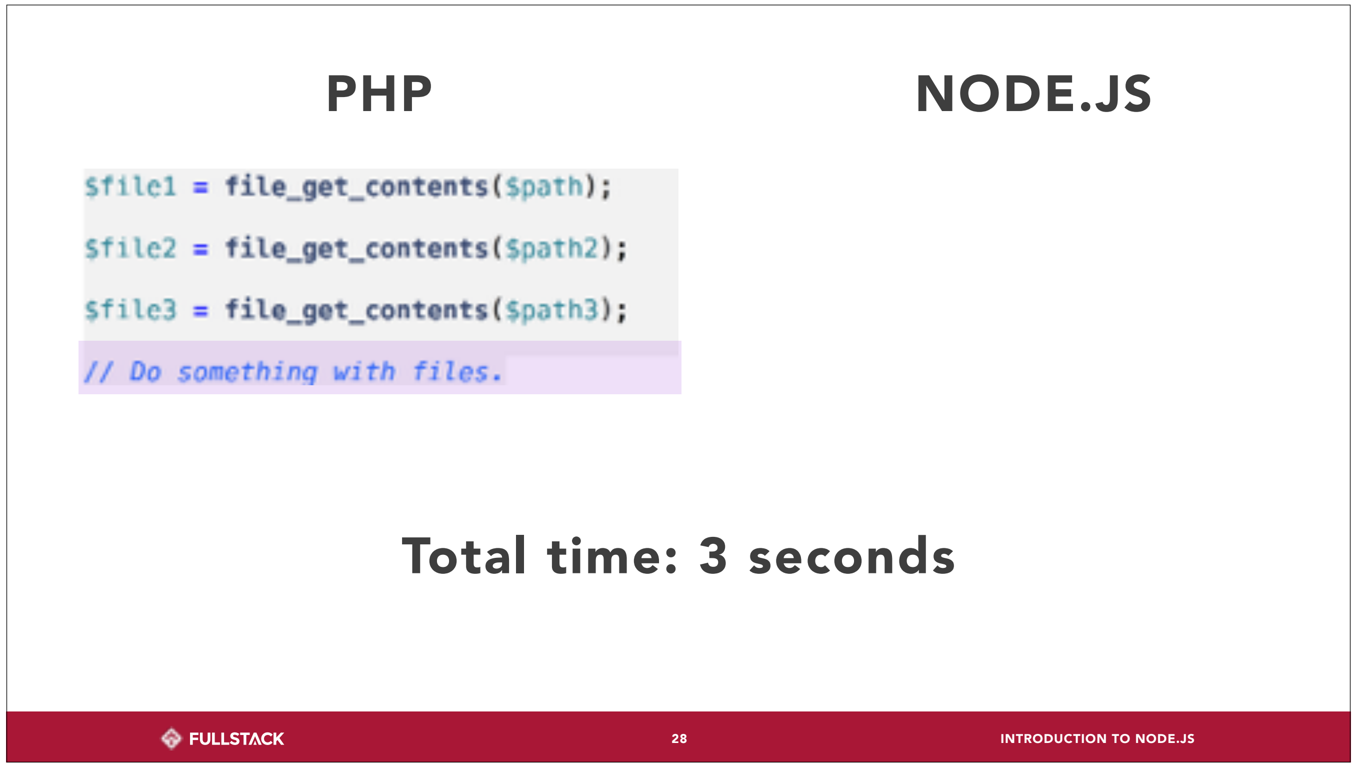
PHP

NODE.JS

```
$file1 = file_get_contents($path);
$file2 = file_get_contents($path2);
$file3 = file_get_contents($path3);
// Do something with files.
```

File3 read takes 1s (1000ms).
Program pauses.

Finally, line 3 of the program starts and, as you guessed it, the program pauses for one second. And the total time of our program’s execution is . . .



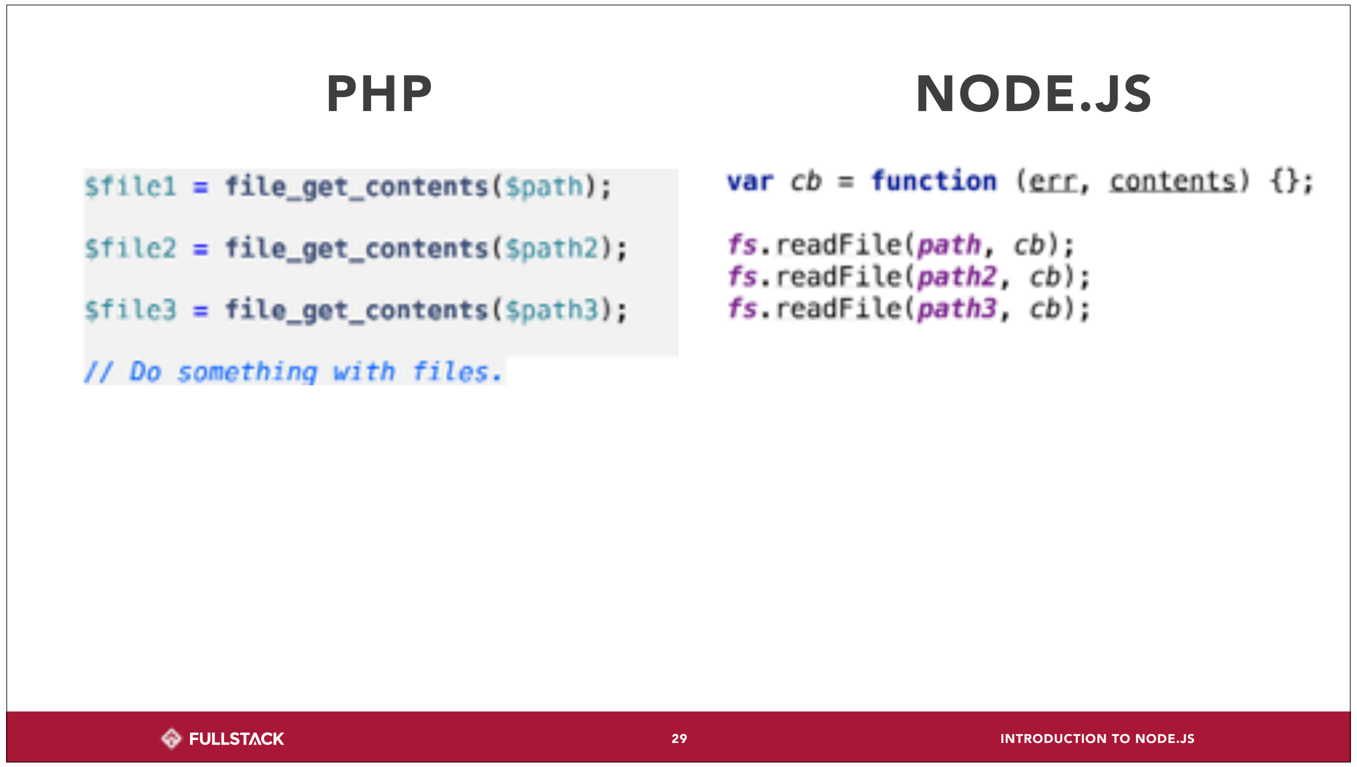
PHP

NODE.JS

```php
$file1 = file_get_contents($path);
$file2 = file_get_contents($path2);
$file3 = file_get_contents($path3);
// Do something with files.
```

Total time: 3 seconds

Right, 3 seconds.



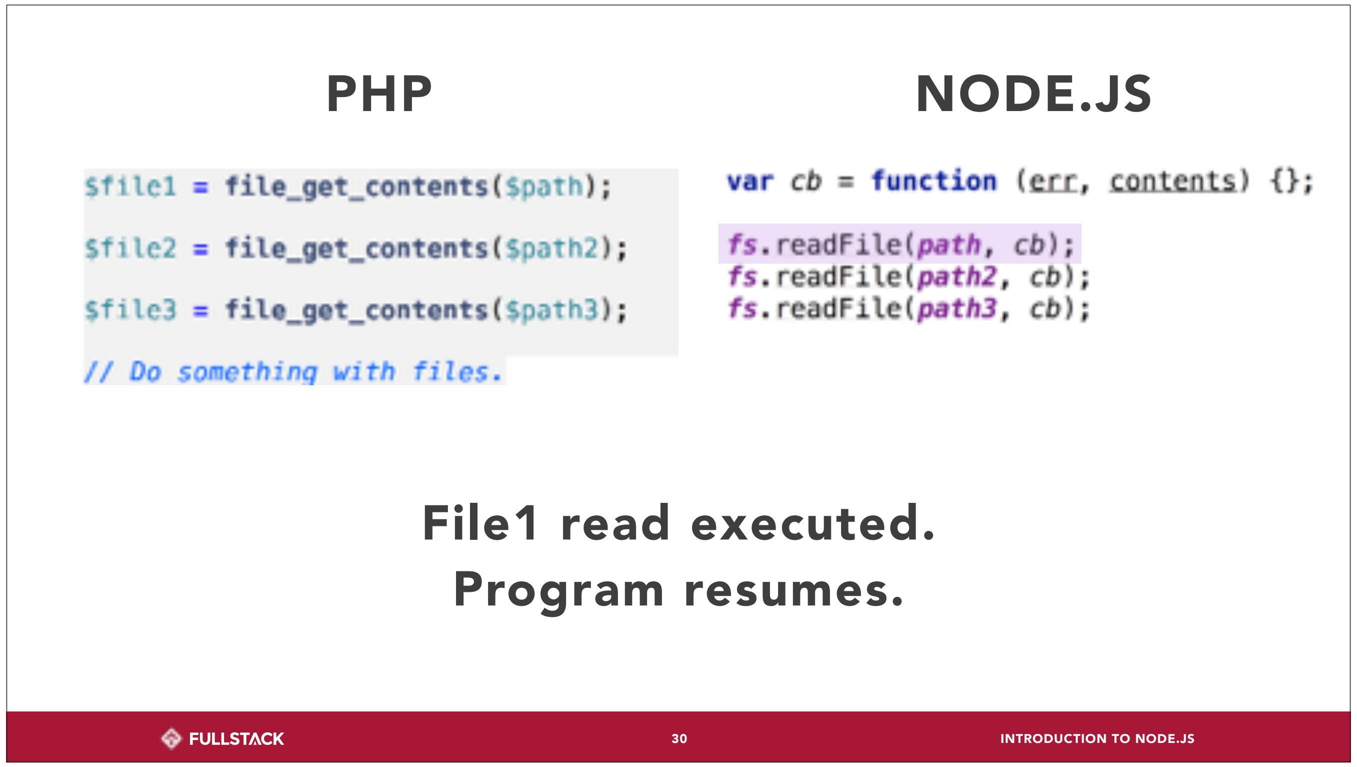
PHP

```
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

Now, we have a simple node.js program accomplishing the same task. Let’s skip the evaluation of our cb variable being assigned, that’s just for clarity.


PHP

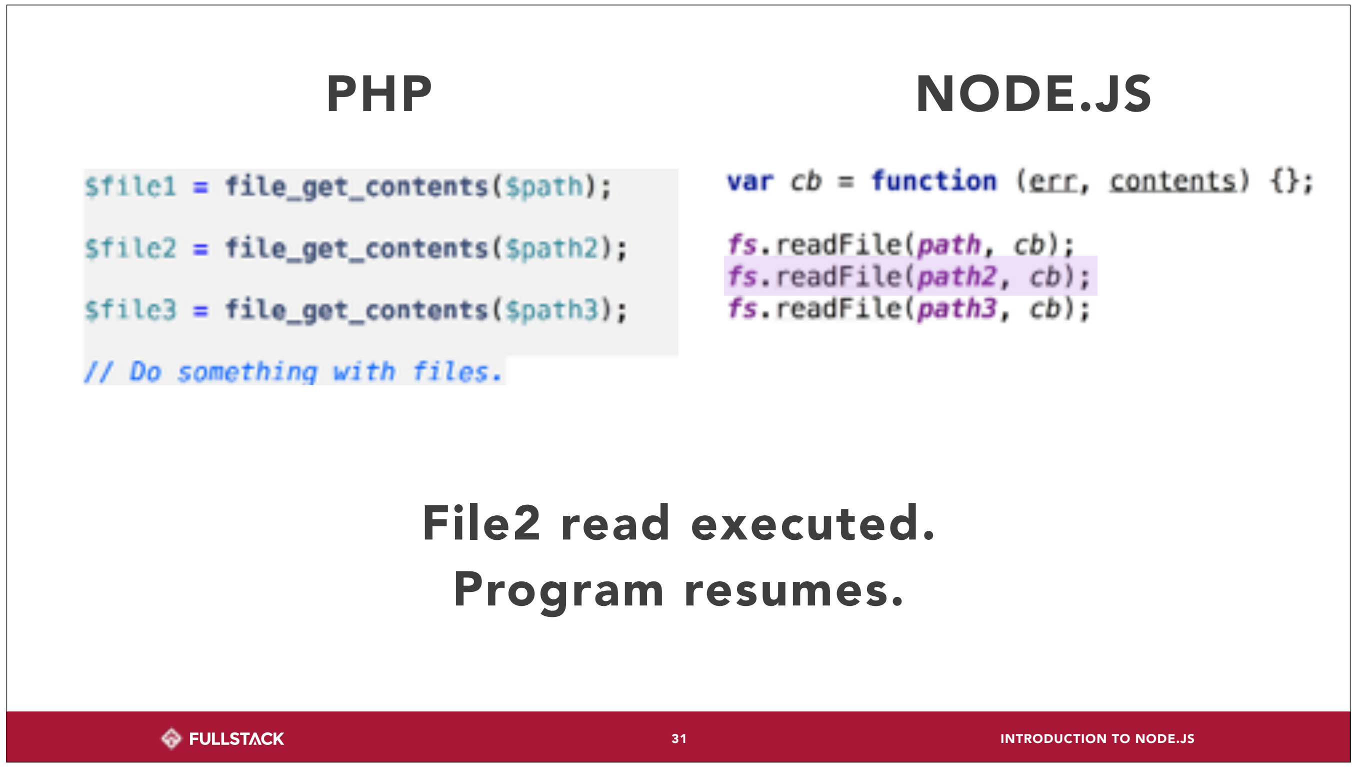
```
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

File1 read executed.
Program resumes.

First read file line is executed. A background process is initiated to retrieve the file. Unlike how our blocking program worked, the program allows for the background process to handle the file read and moves on.



PHP

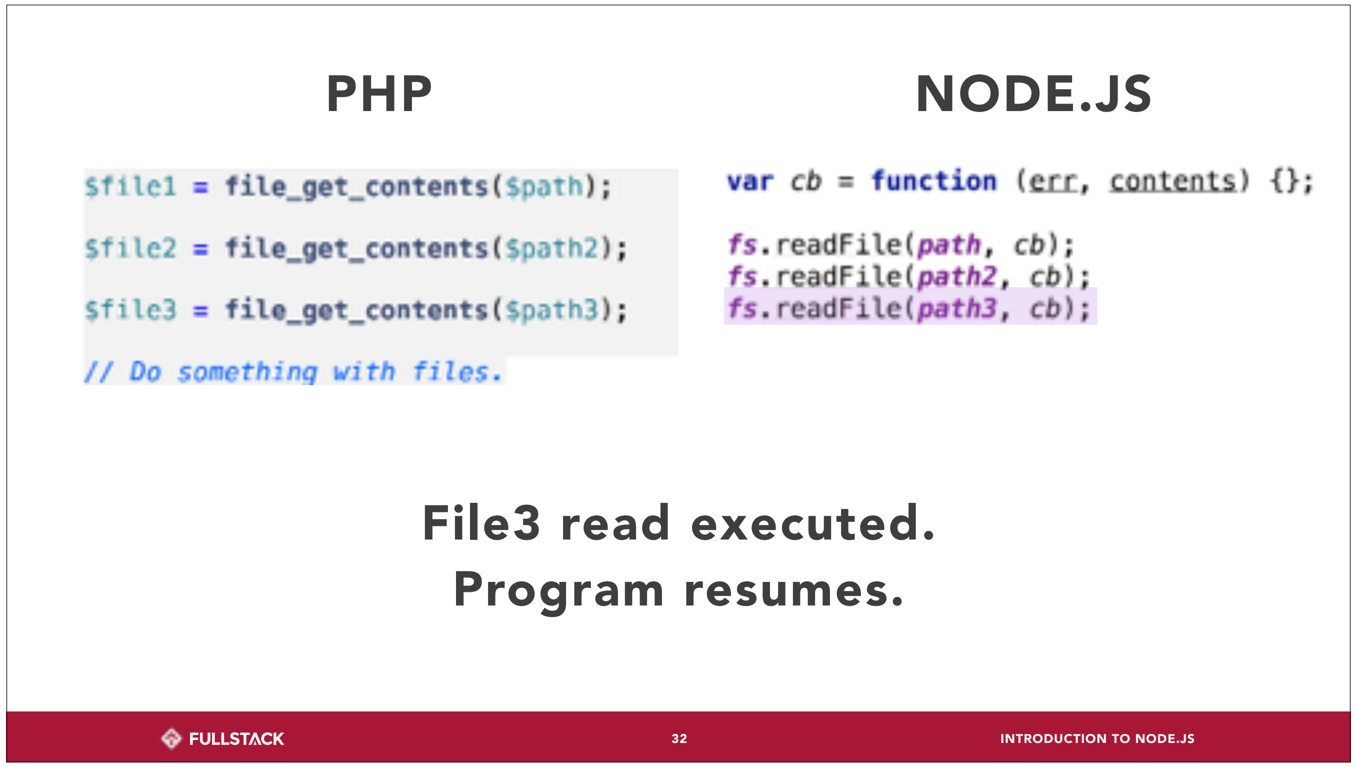
```
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

File2 read executed.
Program resumes.

Same for our second read file. Background process initiated, we move on. Any predictions for the third?



PHP

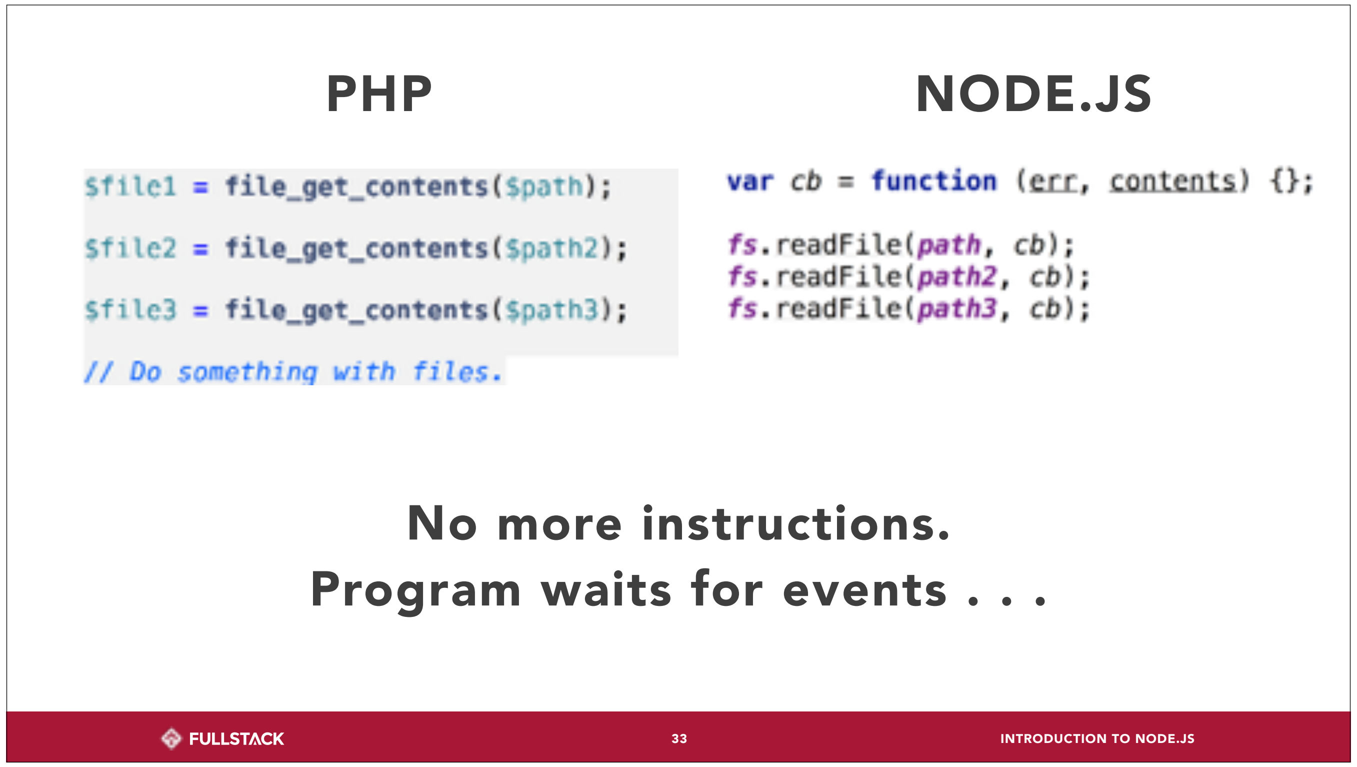
```php
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```javascript
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

File3 read executed.
Program resumes.

Yup, go get file, but program execution resumes.


PHP

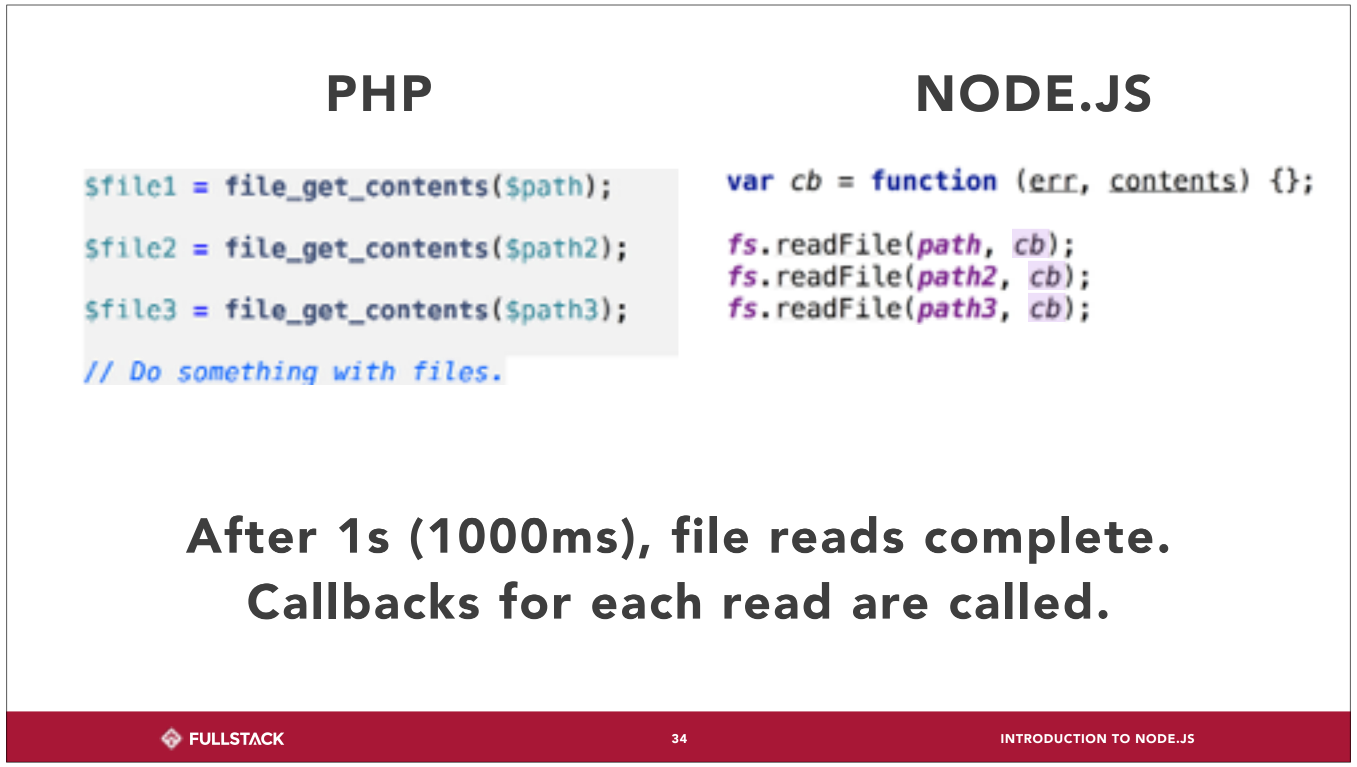
```
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

No more instructions.
Program waits for events . . .

We have no more code to immediately evaluate, so our program waits for events to happen.



PHP

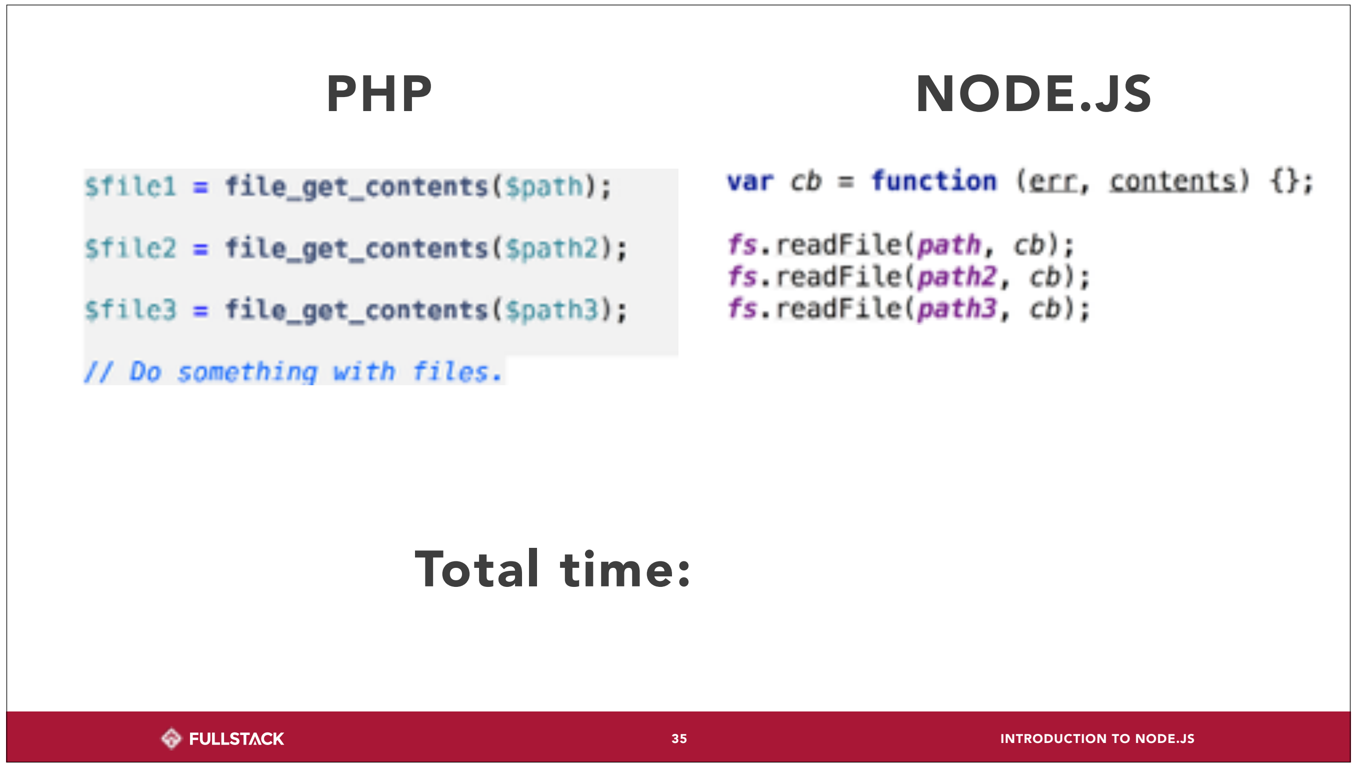
```
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

After 1s (1000ms), file reads complete. Callbacks for each read are called.

But after that 1 second, the background operations dispatched to retrieve our files are now sending signals to our main program with the file contents in tow. These signals activate our callbacks, which receive the file contents.



PHP

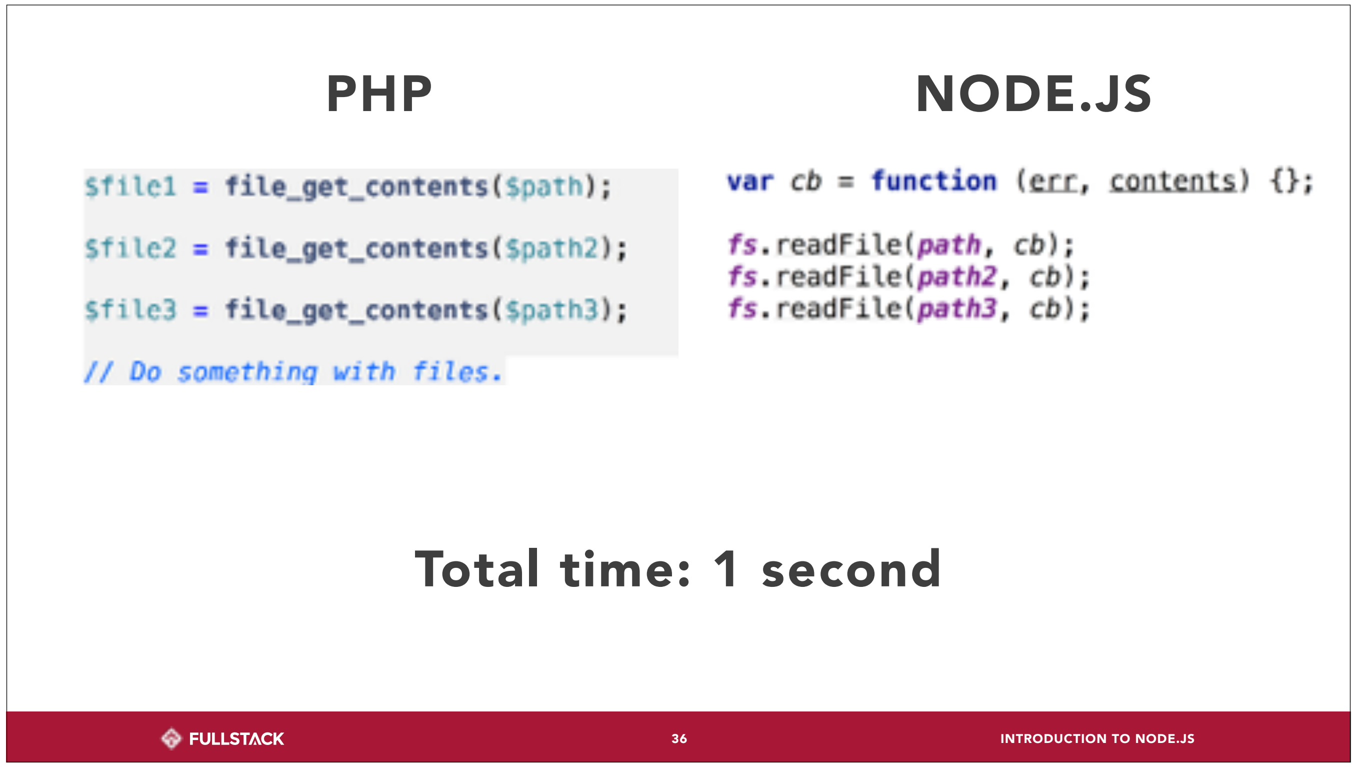
```php
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```javascript
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

Total time:

So, how much time did our program take to execute?



PHP

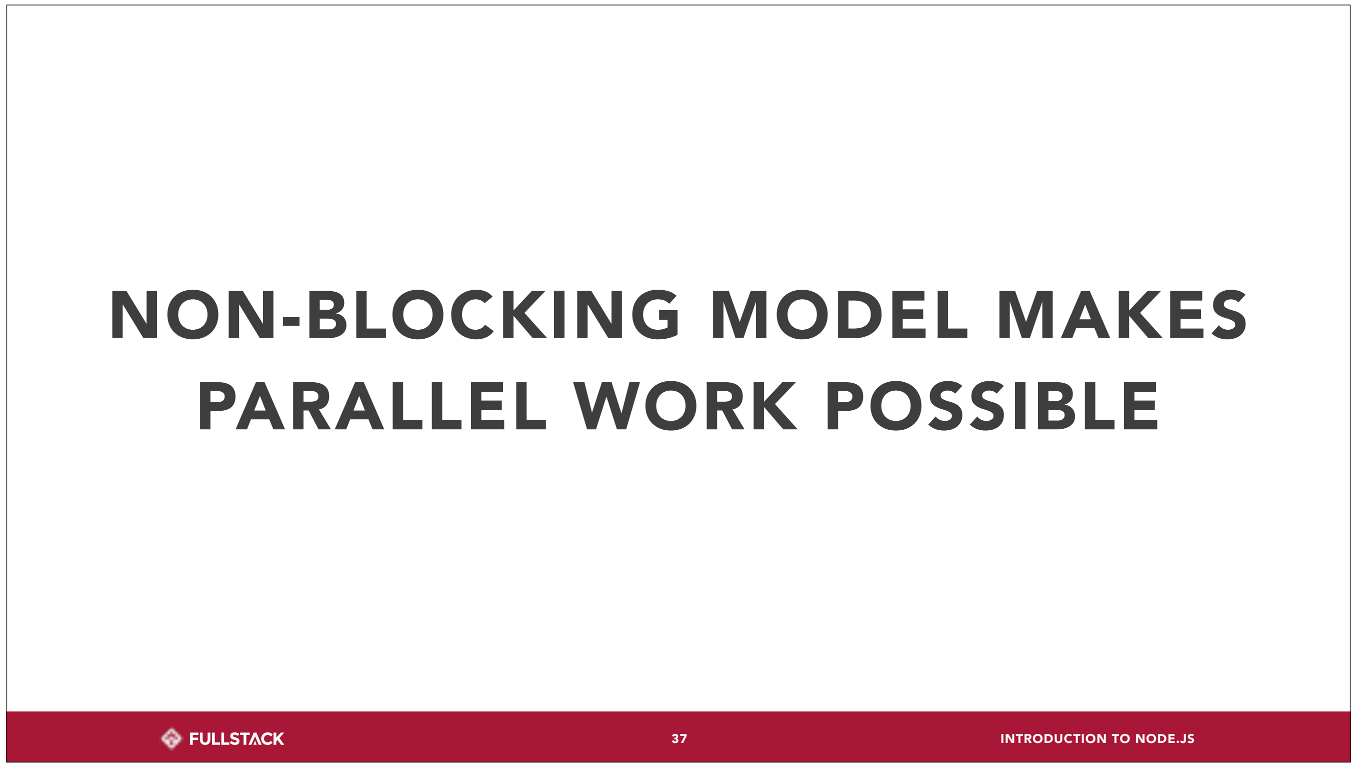
```php
$file1 = file_get_contents($path);

$file2 = file_get_contents($path2);

$file3 = file_get_contents($path3);

// Do something with files.
```

NODE.JS

```js
var cb = function (err, contents) {};

fs.readFile(path, cb);
fs.readFile(path2, cb);
fs.readFile(path3, cb);
```

Total time: 1 second

1 second. To do the same work that the PHP process did in 3.



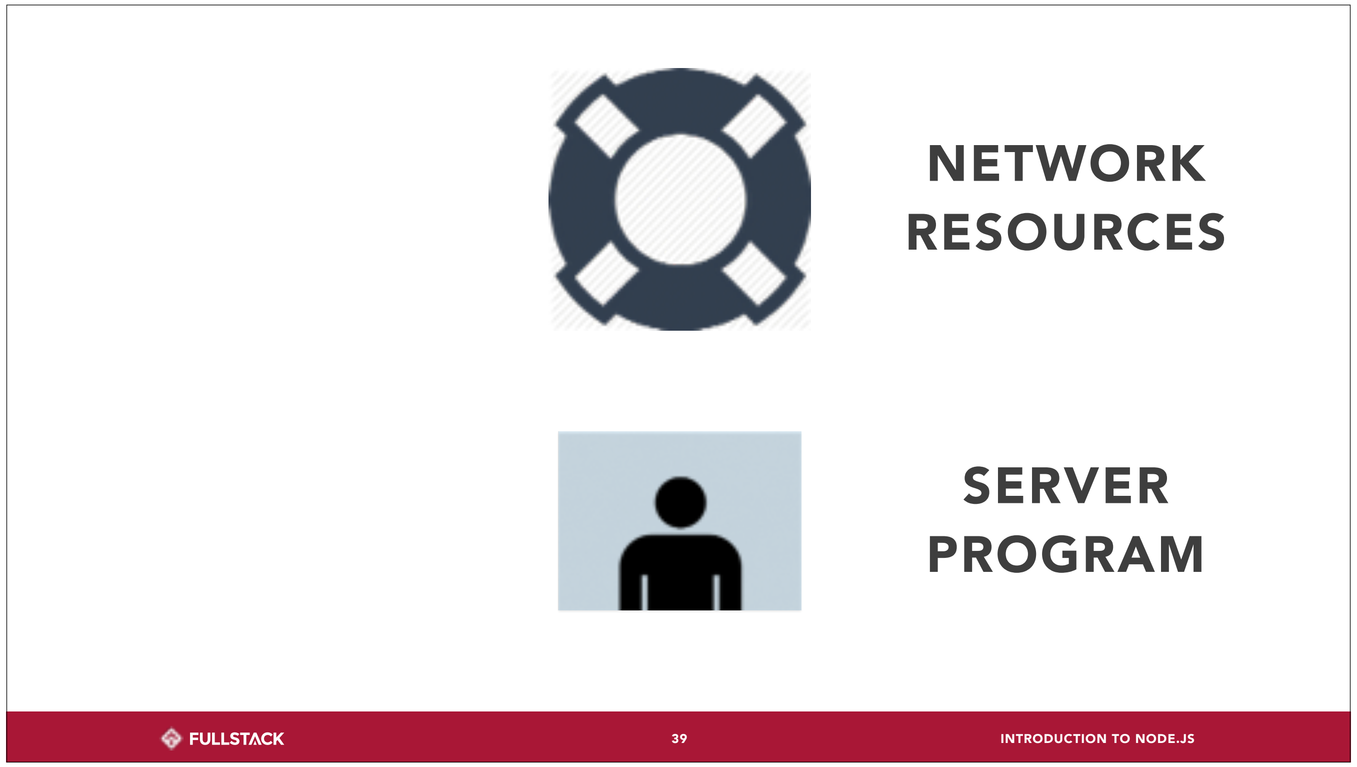
NON-BLOCKING MODEL MAKES PARALLEL WORK POSSIBLE

This is because the non-blocking model makes parallel work possible.

[QUESTIONS]

BLOCKING BANK

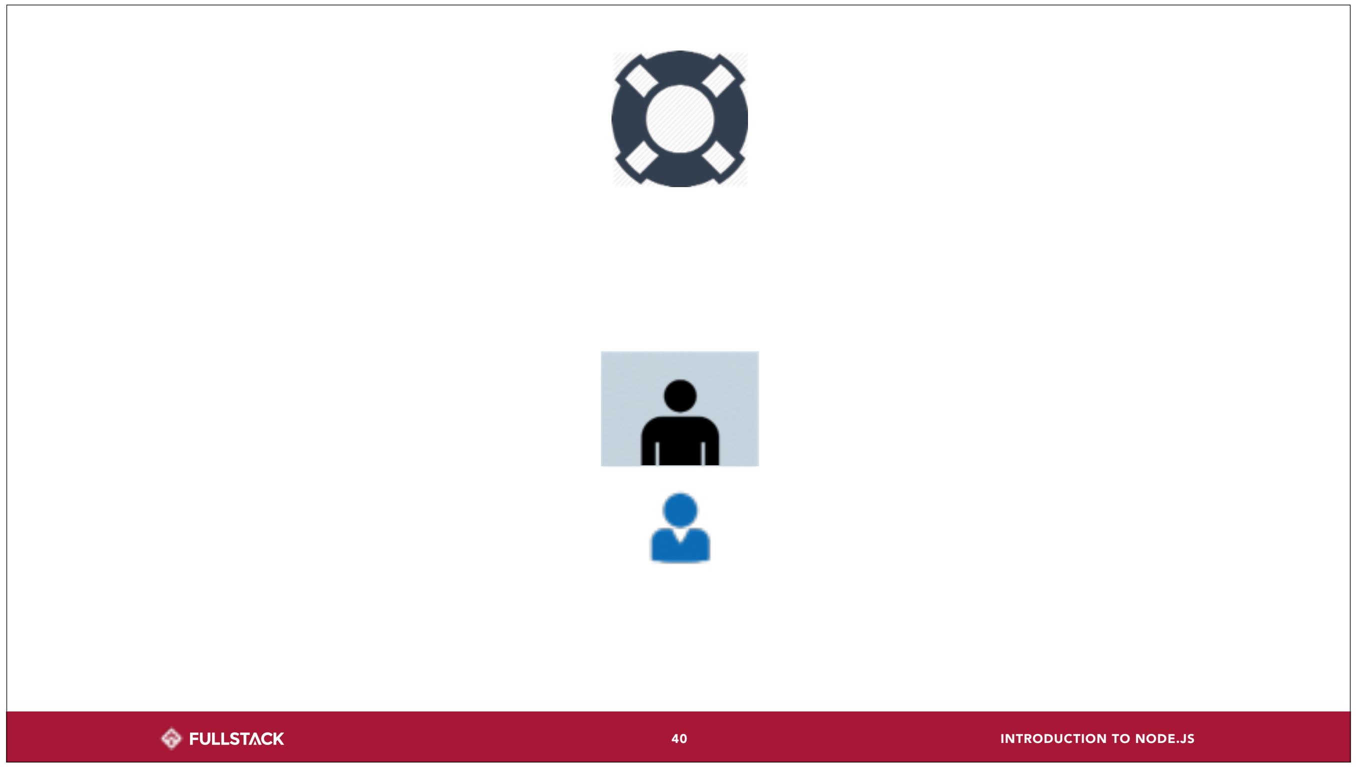
Okay, I’m about to jump into a huge analogy here. This is the Blocking Bank. The bank has a teller, and it has a vault that holds all of the customers assets.



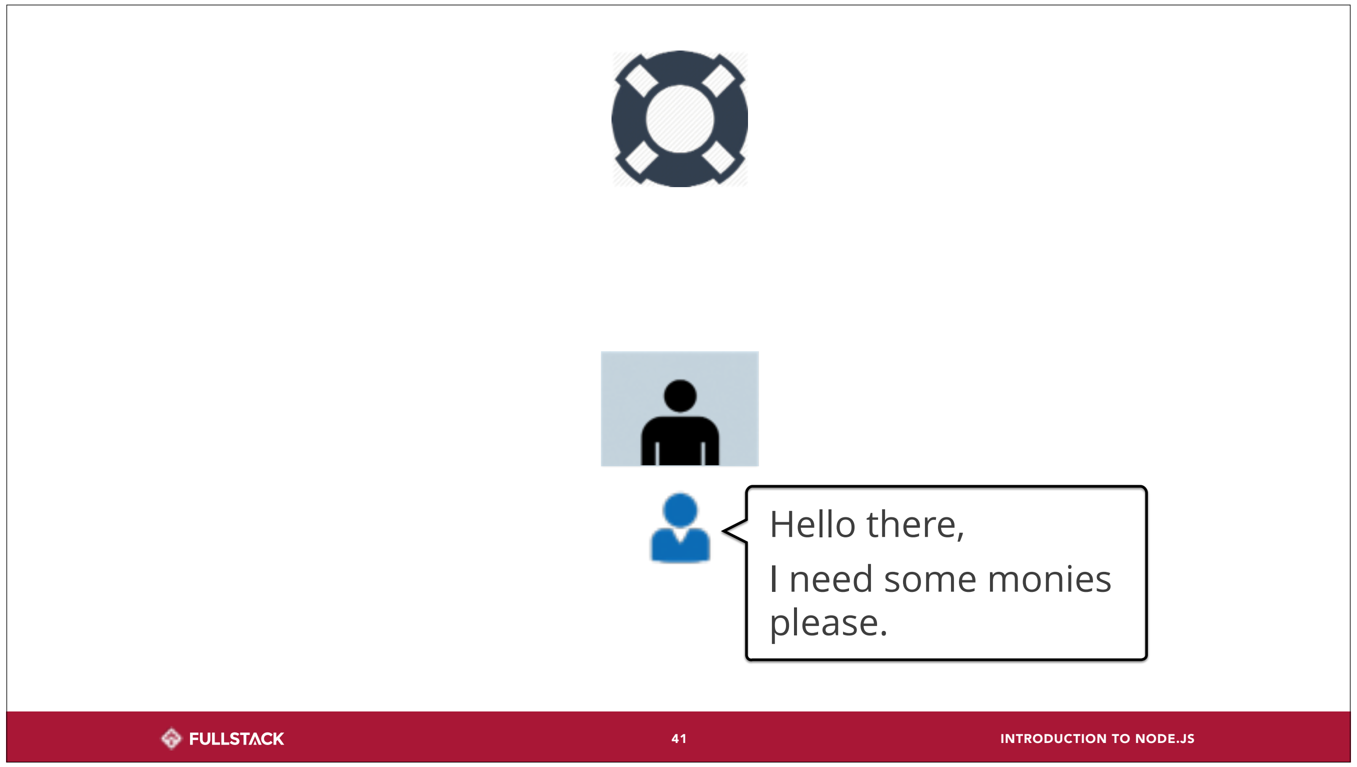
NETWORK RESOURCES

SERVER PROGRAM

The bank teller is our server program, and the vault is our network resources. This is our file system, our database, third-party APIs like Google Maps, what have you.



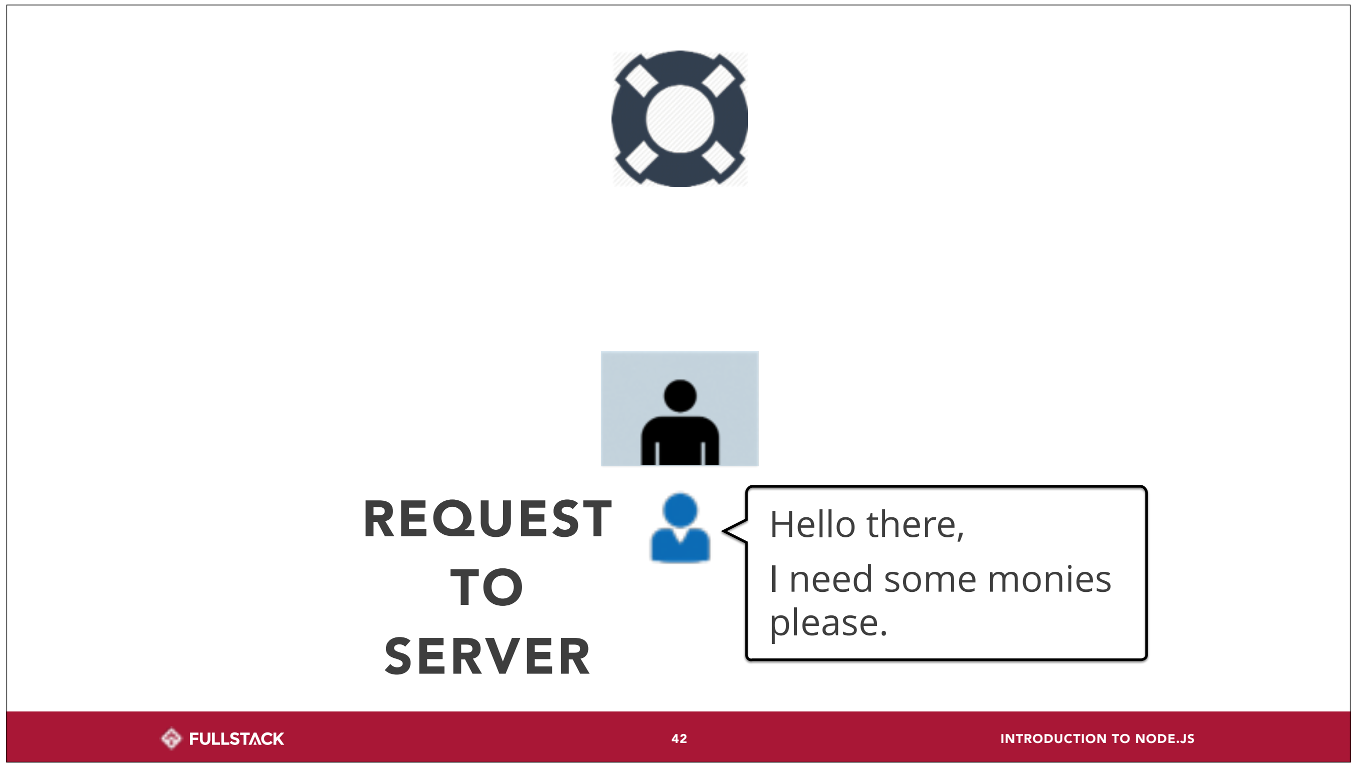
A custom comes into our bank . . .



Hello there,

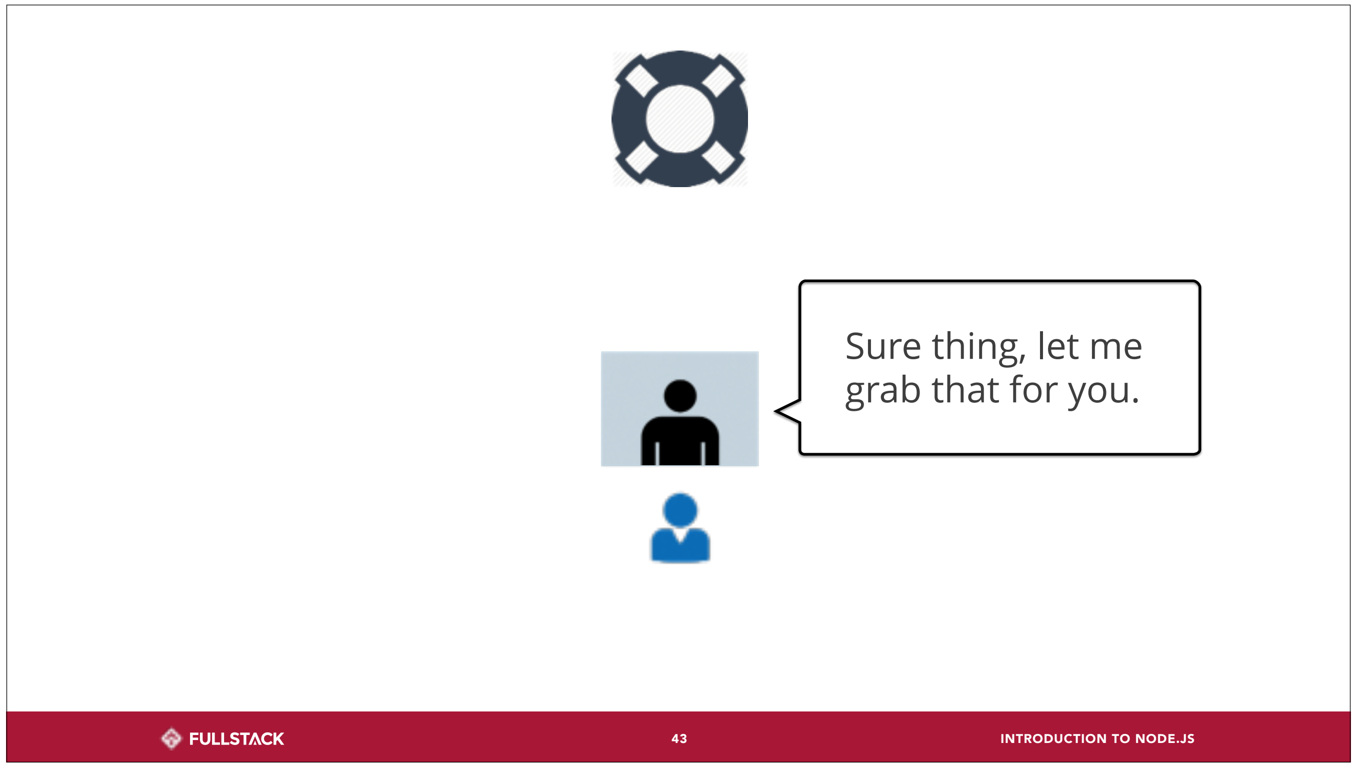
I need some monies please.

And says, [...].

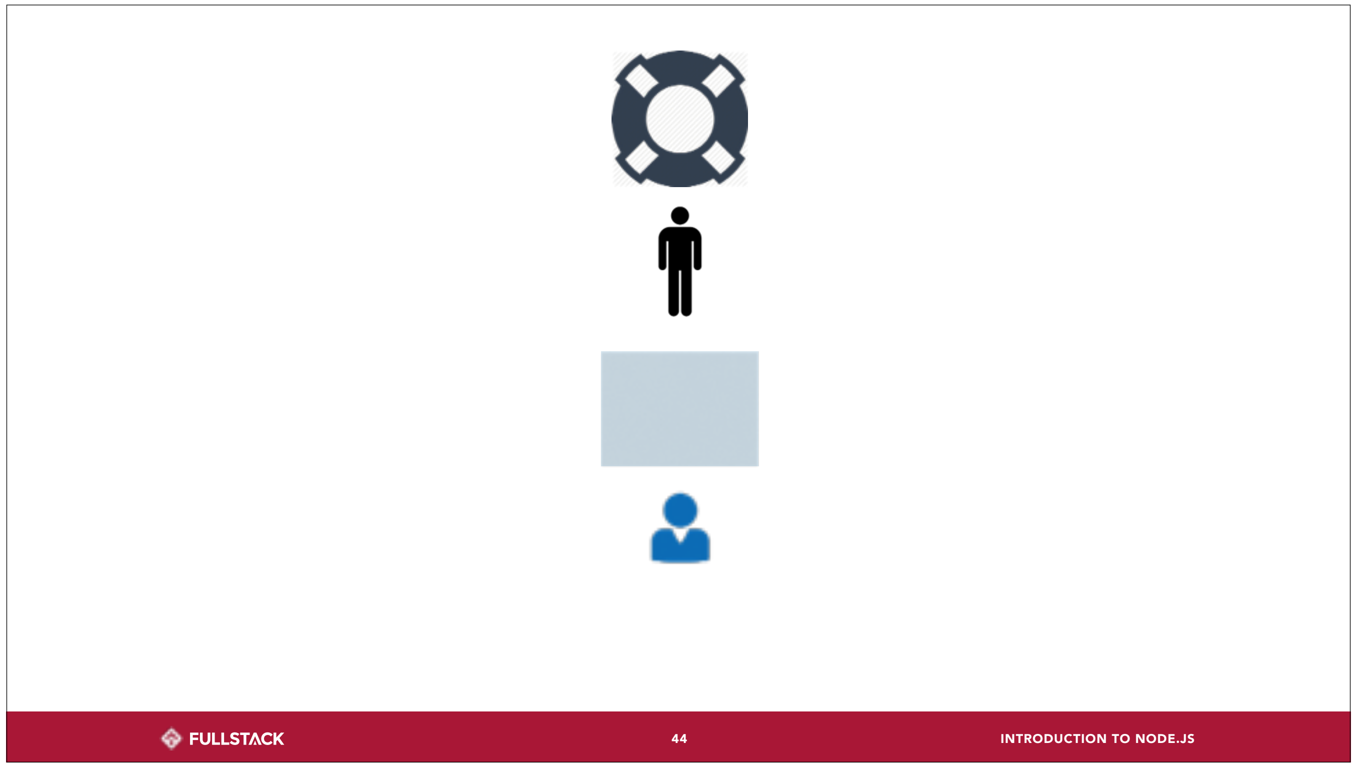


REQUEST TO SERVER

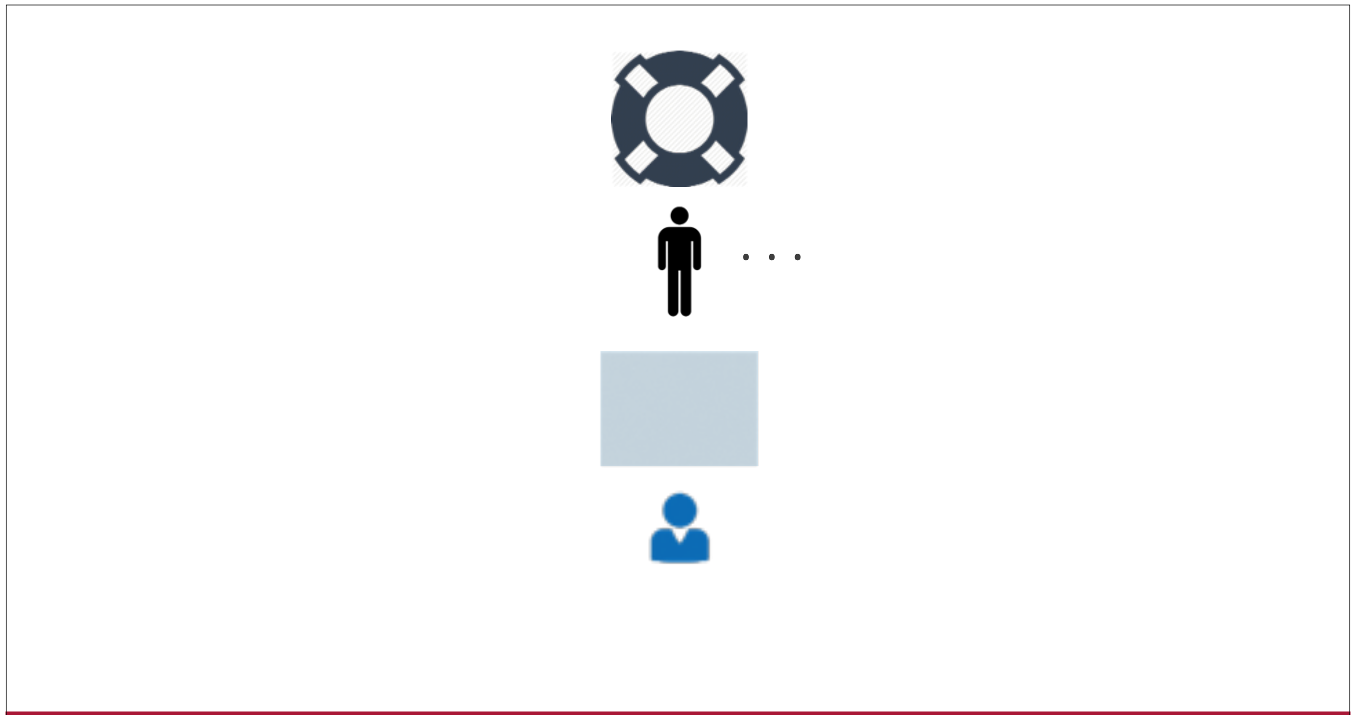
This customer, and subsequent customers, represent requests to the server program.



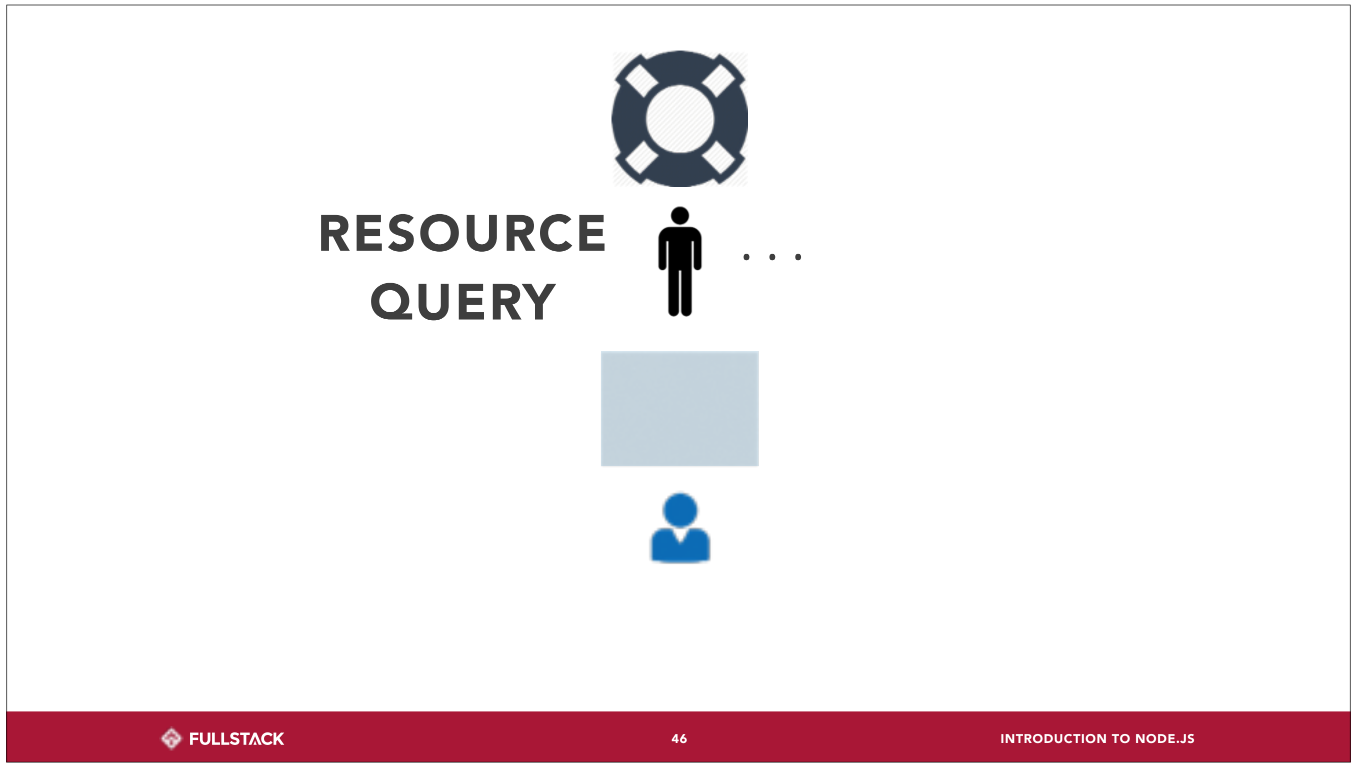
Our bank teller receives the request and acknowledges it.



The teller walks to the back of the bank where the vault is located, opens it and searches for our customer’s monies.

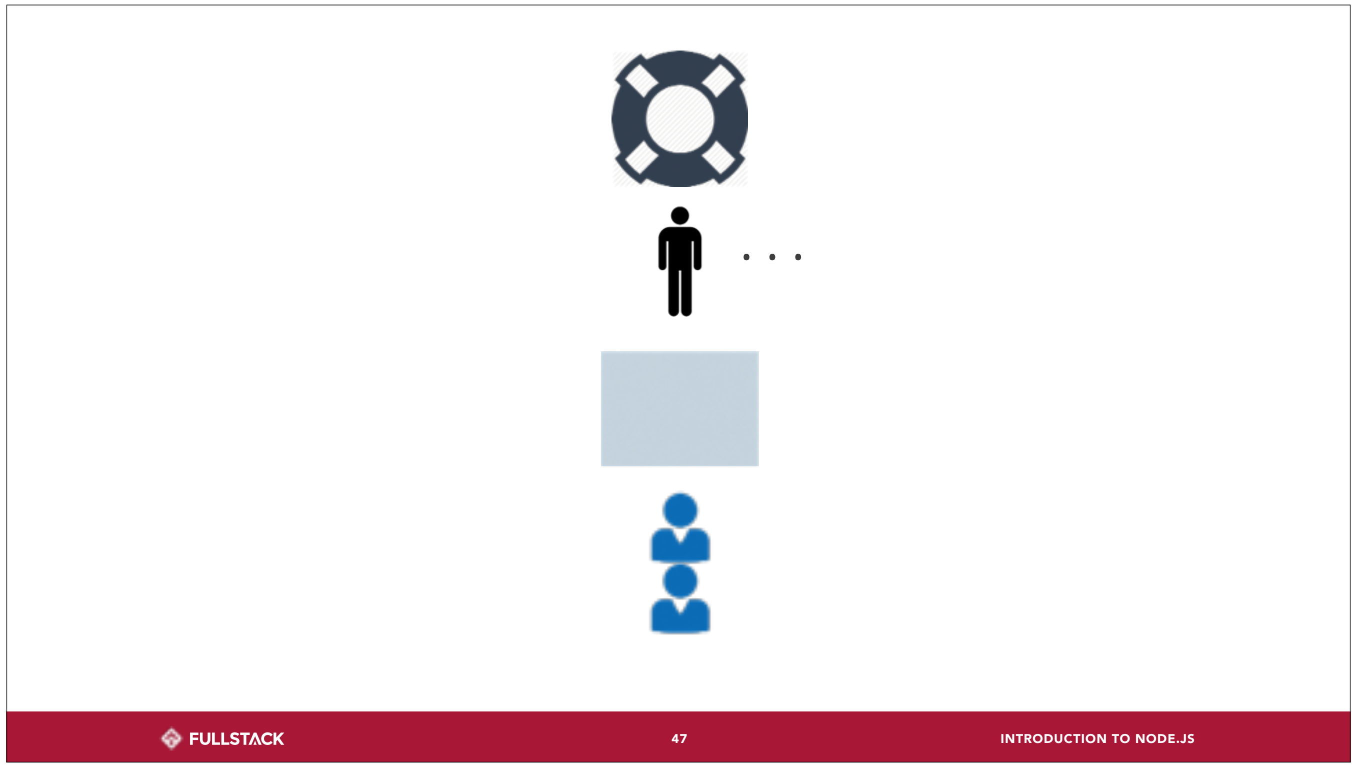


This takes a while, the vault is huge.

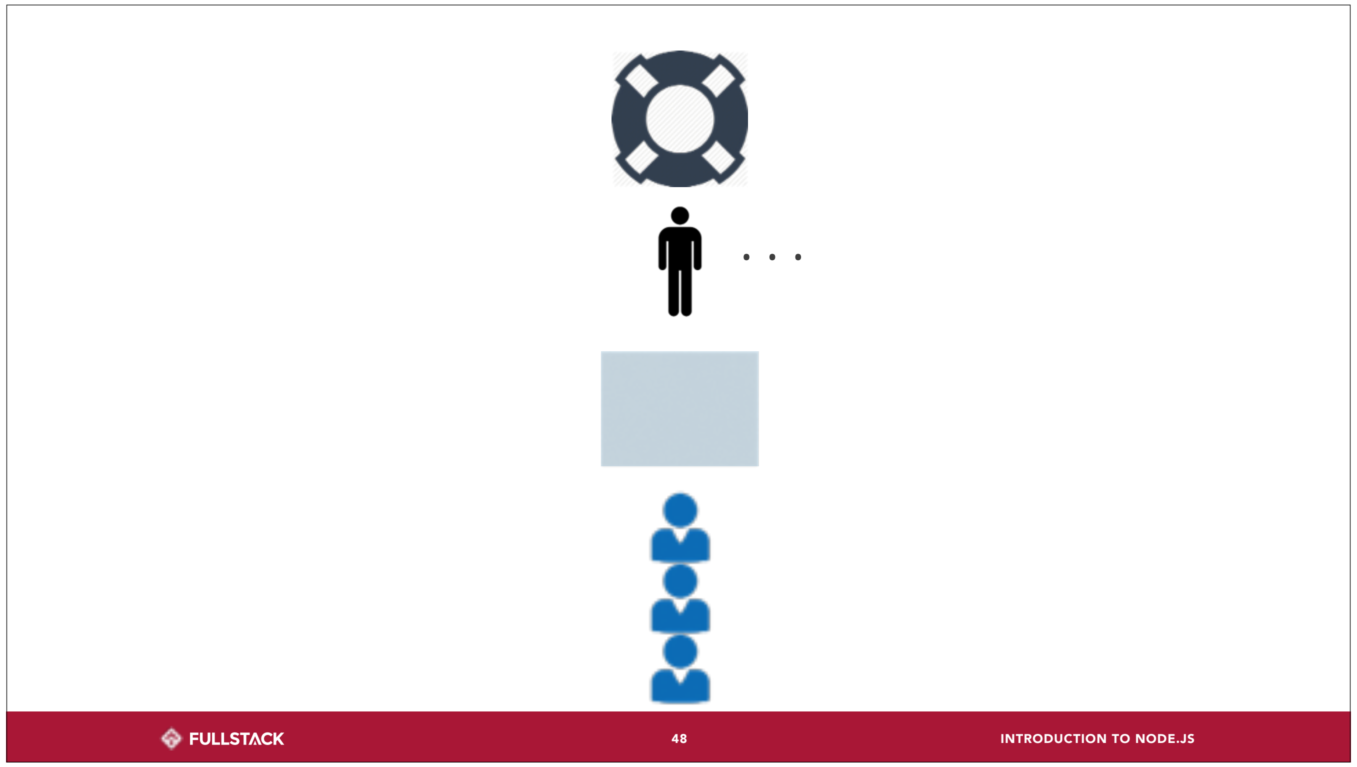


RESOURCE QUERY

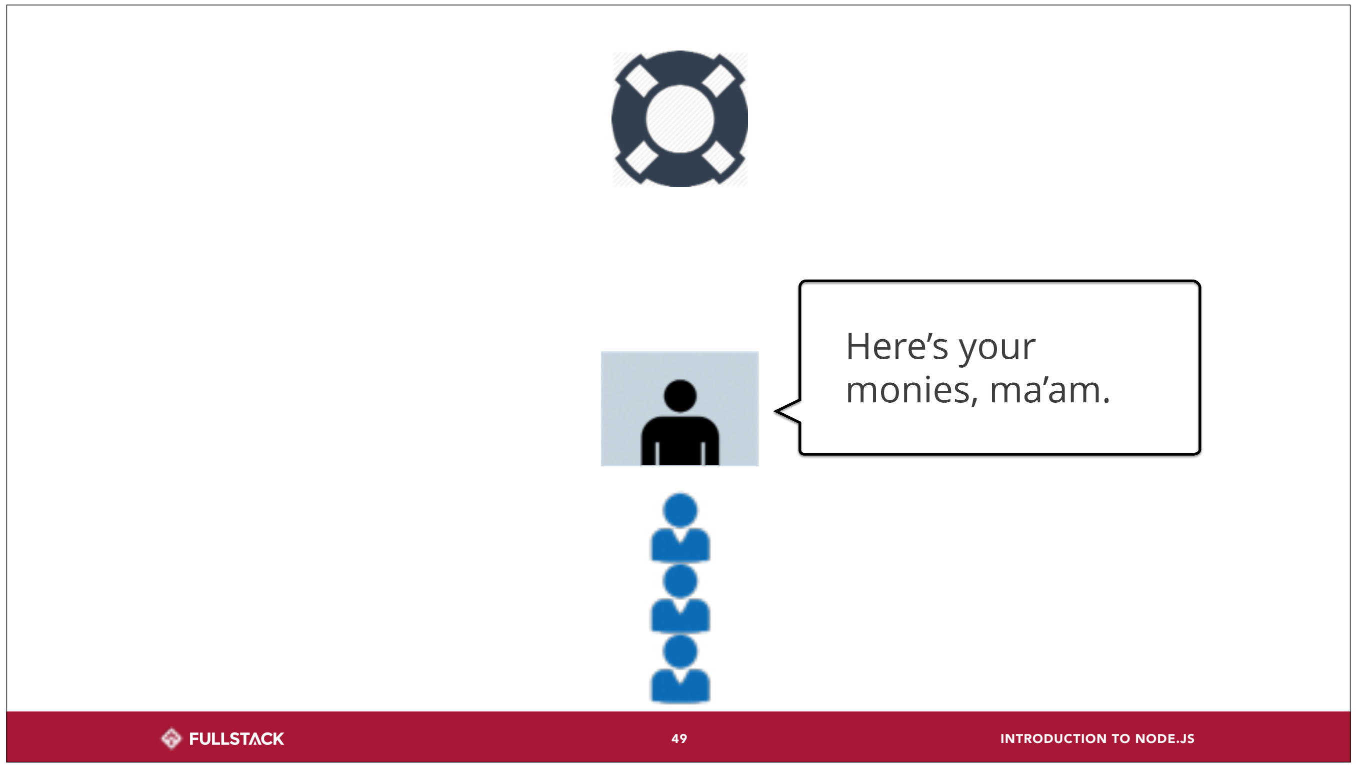
This is a resource query. This is like our server program accessing the file system, or a database. And it takes a while.



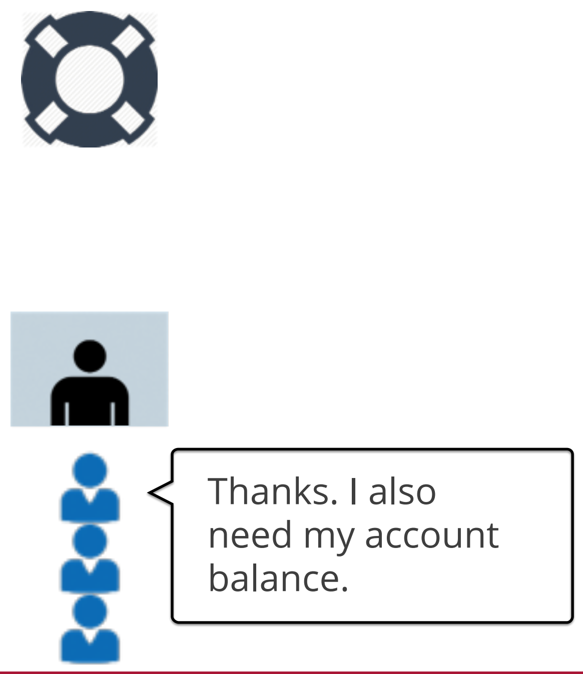
And while the teller is busy accessing the vault for our first customer, more customers are lining up, and are unattended.



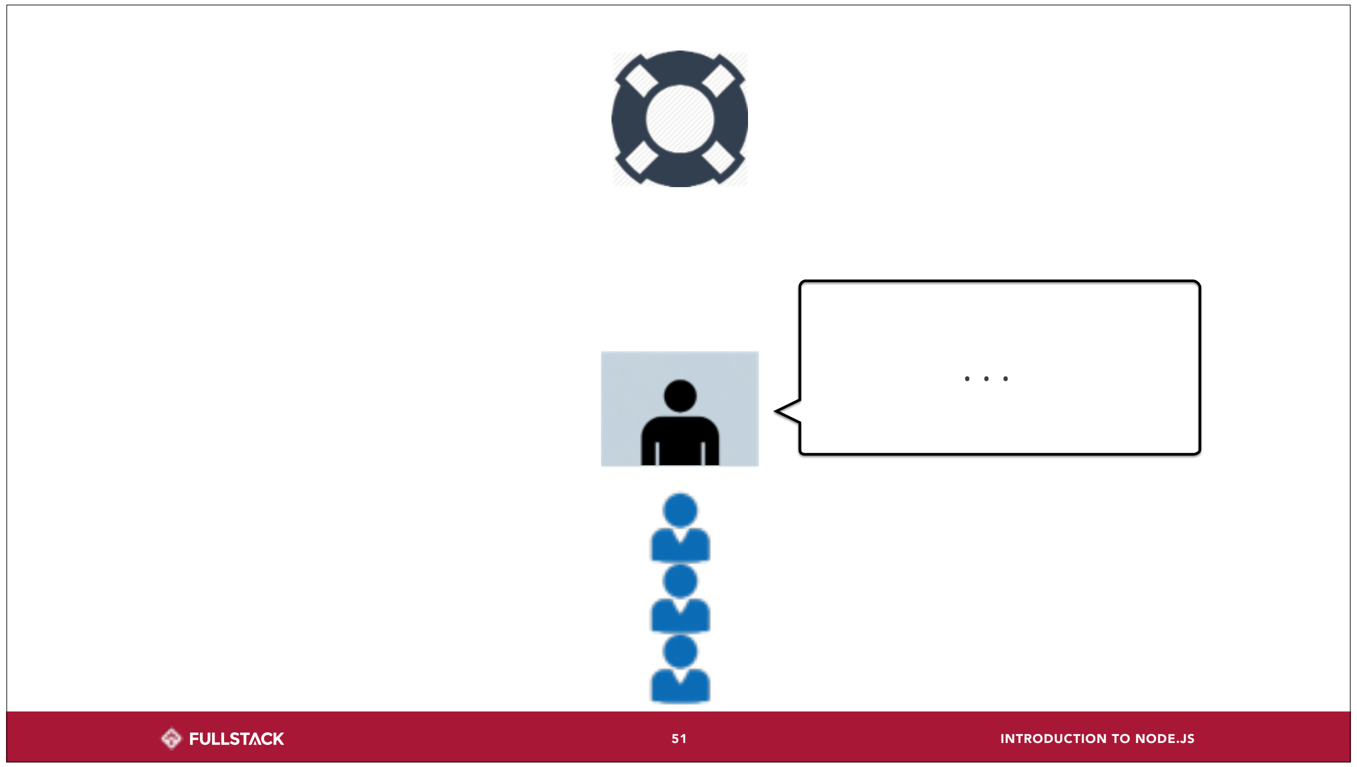
A queue builds and customers/requests are not handled.



Now the teller has the monies ready to hand to our customer.

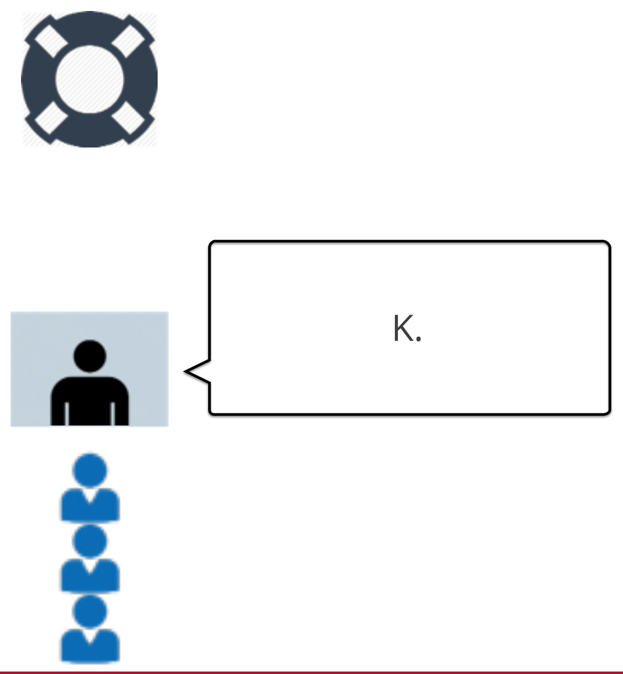


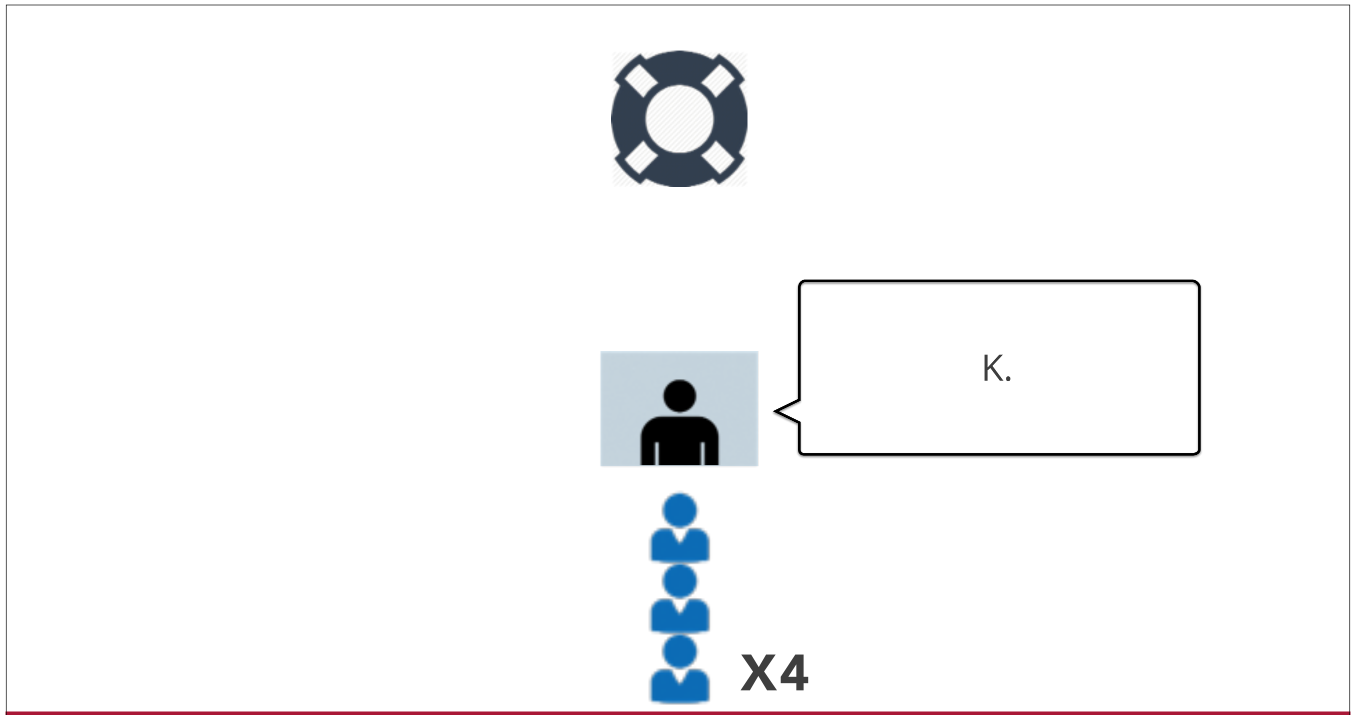
But the customer also needs her account balance.



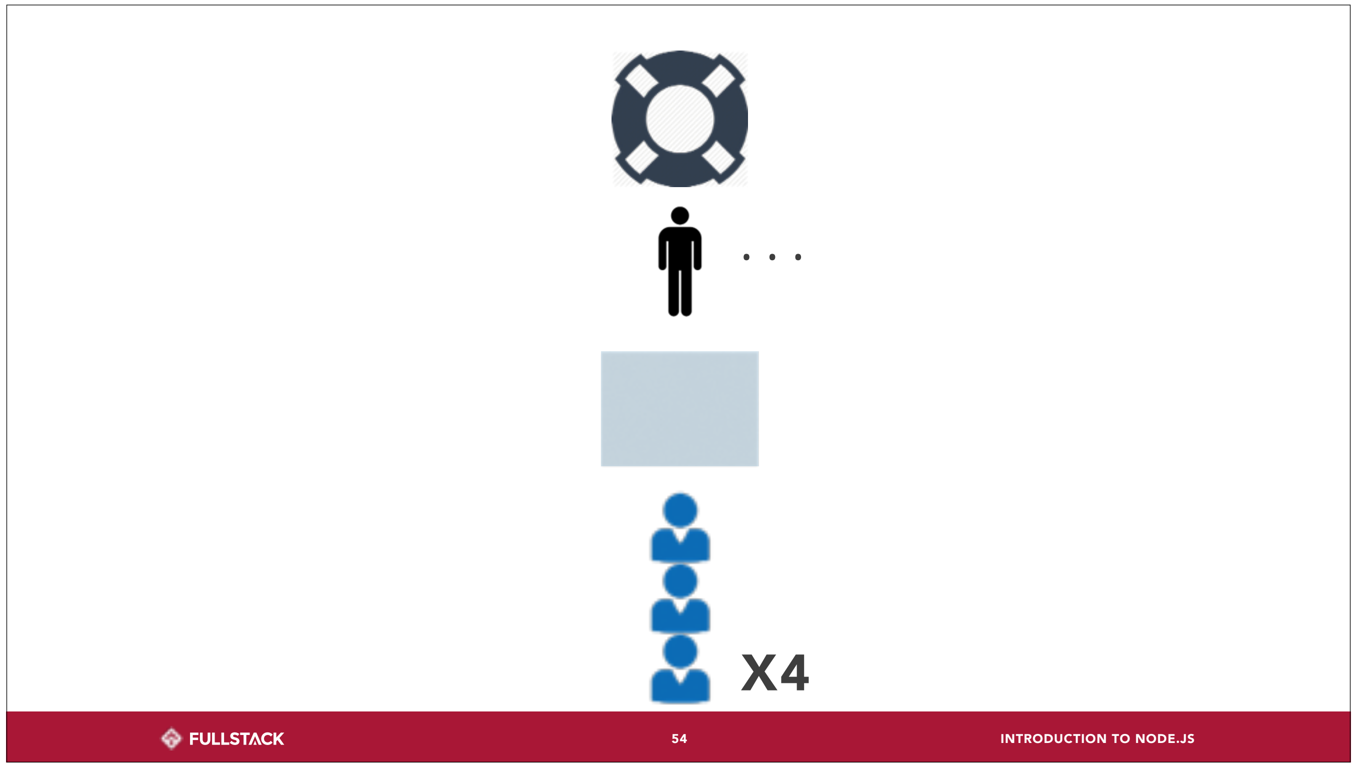
FULLSTACK 51 INTRODUCTION TO NODE.JS

Our teller begrudgingly acknowledges.


K.

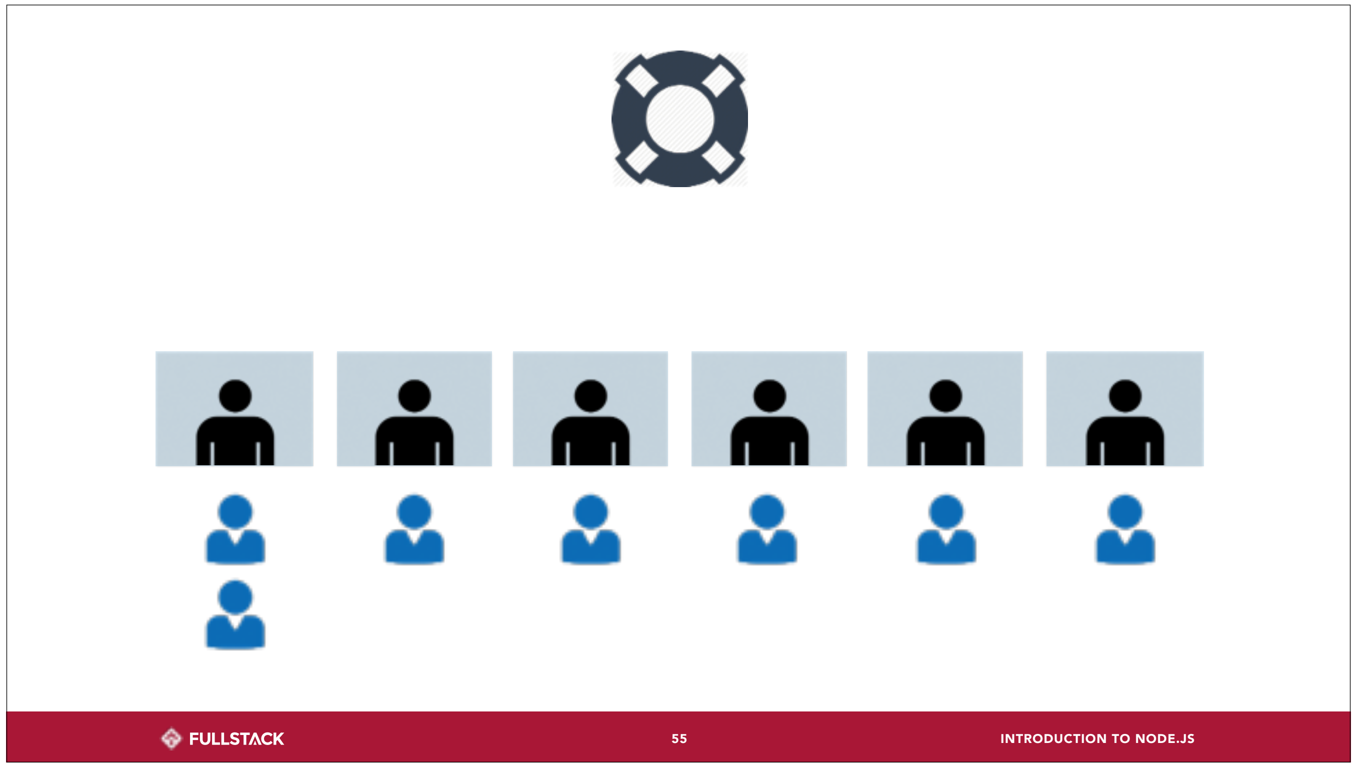


The queue continues to build.

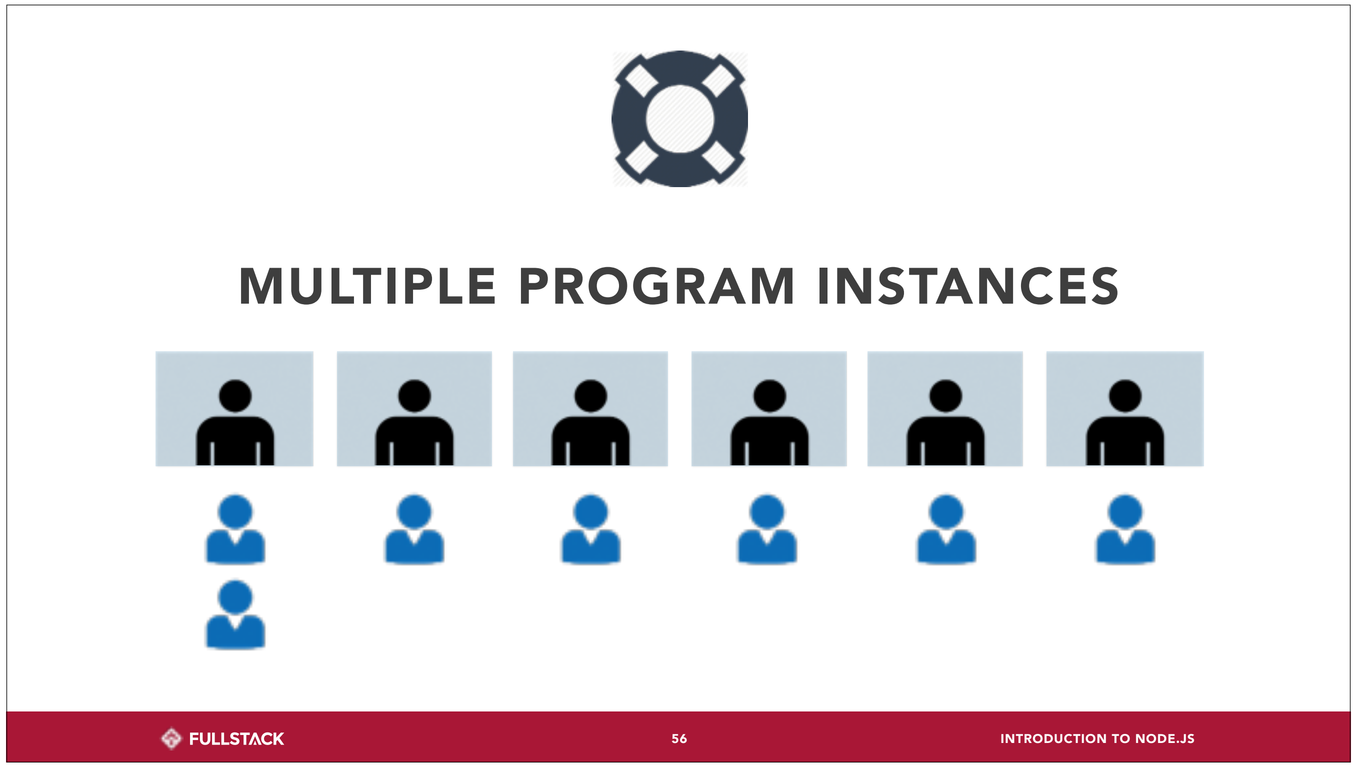


And our teller walks back to the vault to count the customer's monies.

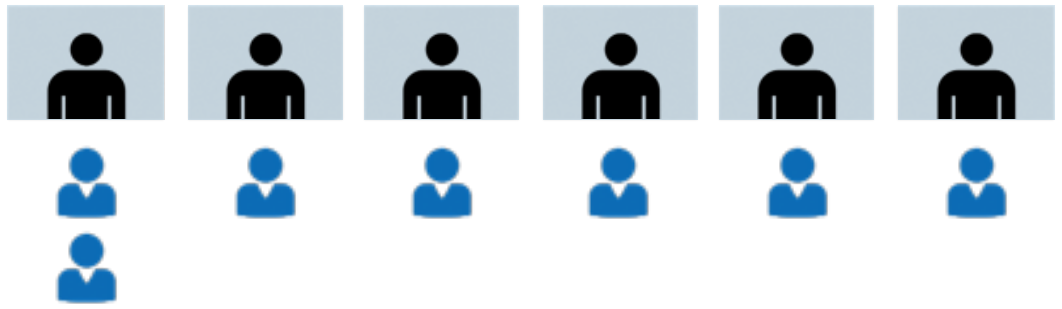
Now, in this banking scenario, what is the obvious way for the bank to handle increased traffic?



Right, they hire more tellers. This is similar to how blocking server technologies handle concurrent, or at the same time, requests.

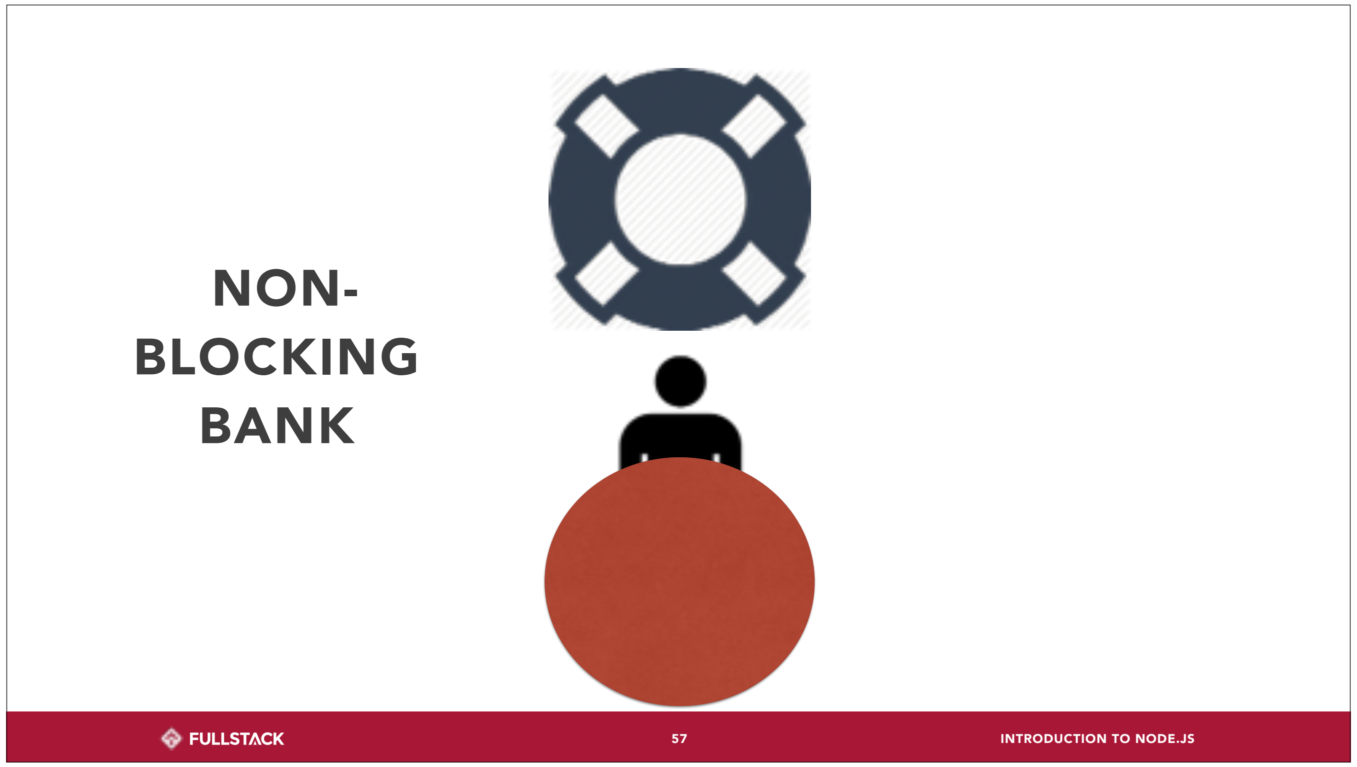


MULTIPLE PROGRAM INSTANCES

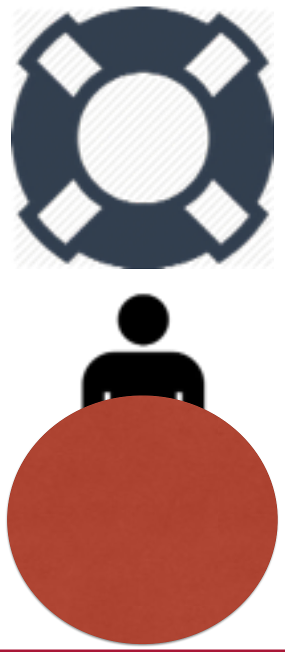


For each request, they spin up another instance of the server program. This allows for each request, or customer, to be handled by that instance.

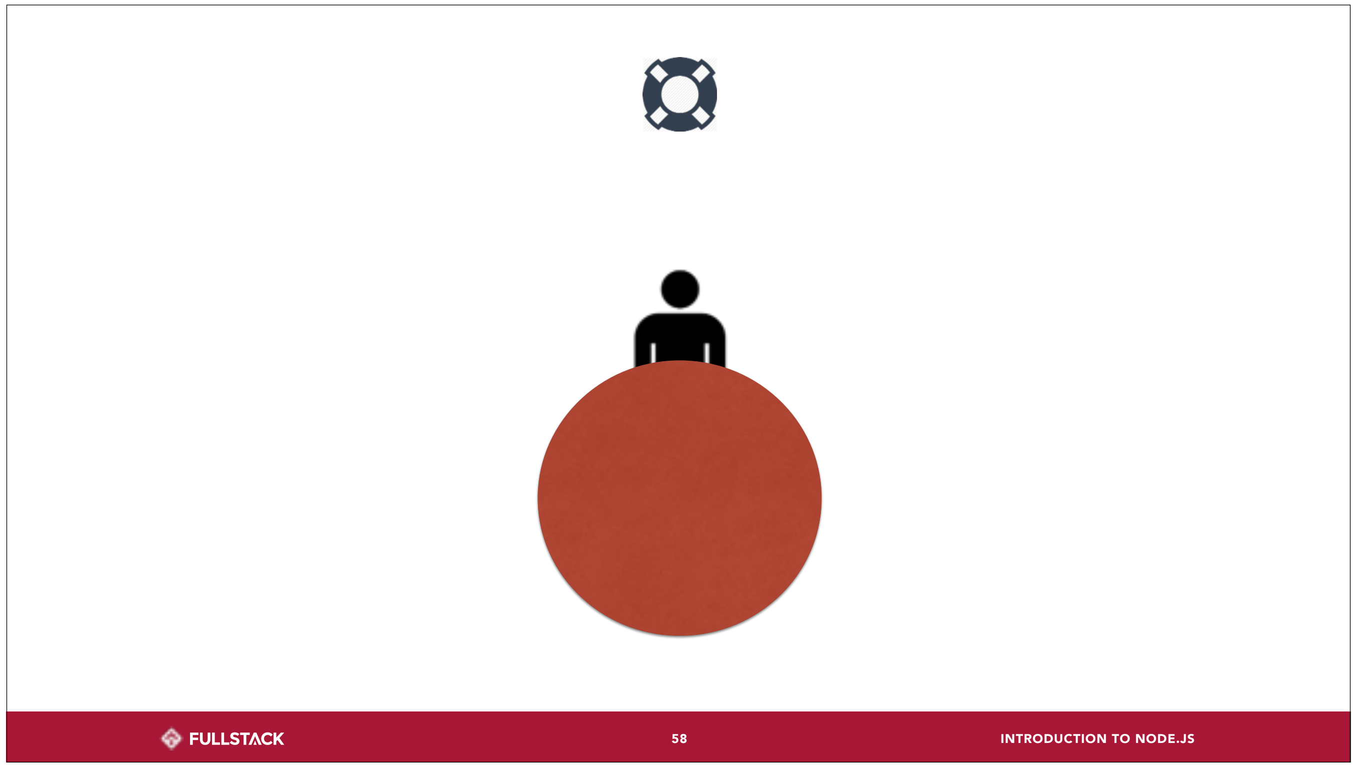
There are many drawbacks to this model. The overhead to instantiate the program—think of it as training a new teller—is high. Program instances take up a lot of memory and time. And these tellers are too focused on their customer to talk to each other. They only share information through the vault.



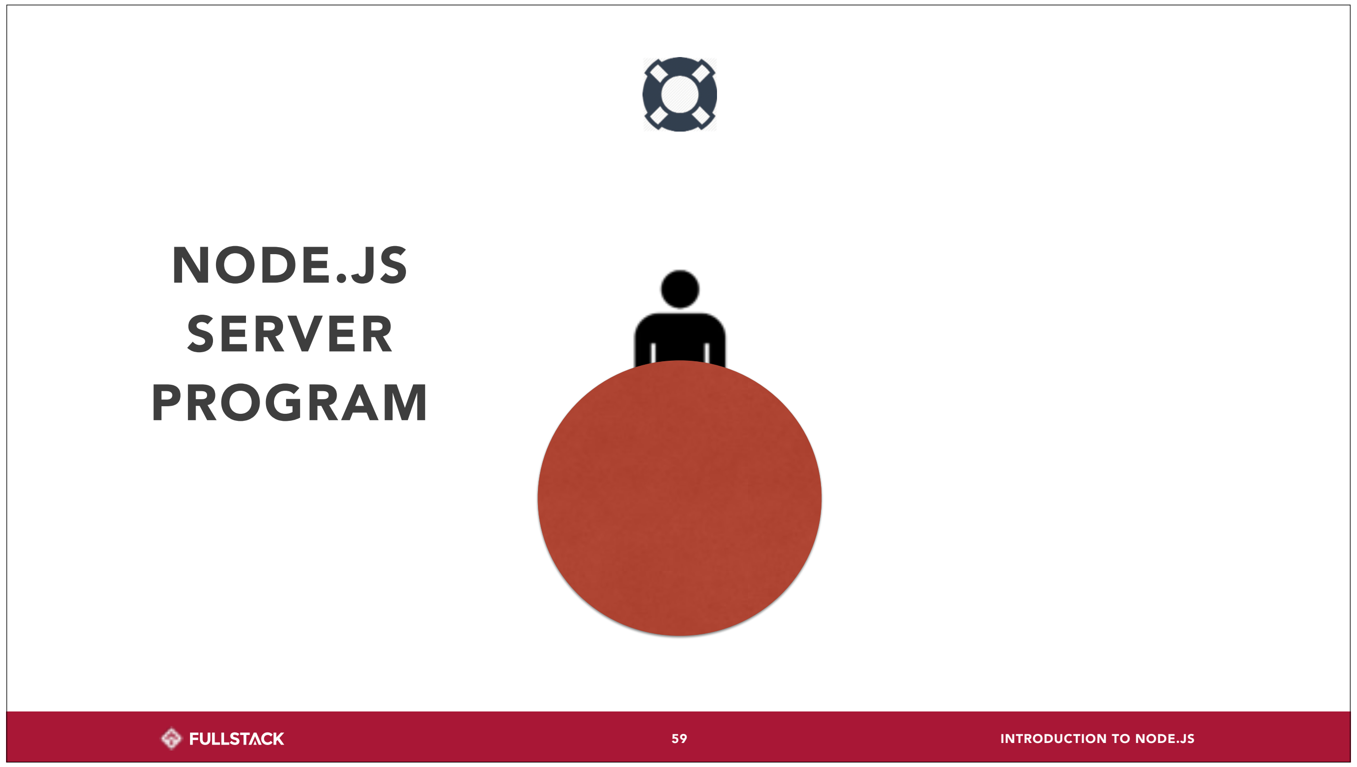
NON-BLOCKING BANK



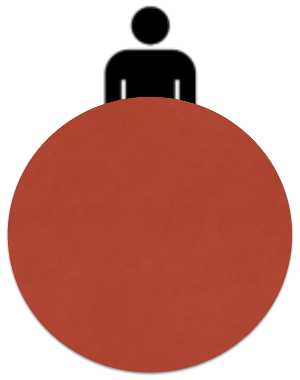
So, here's our non-blocking bank.



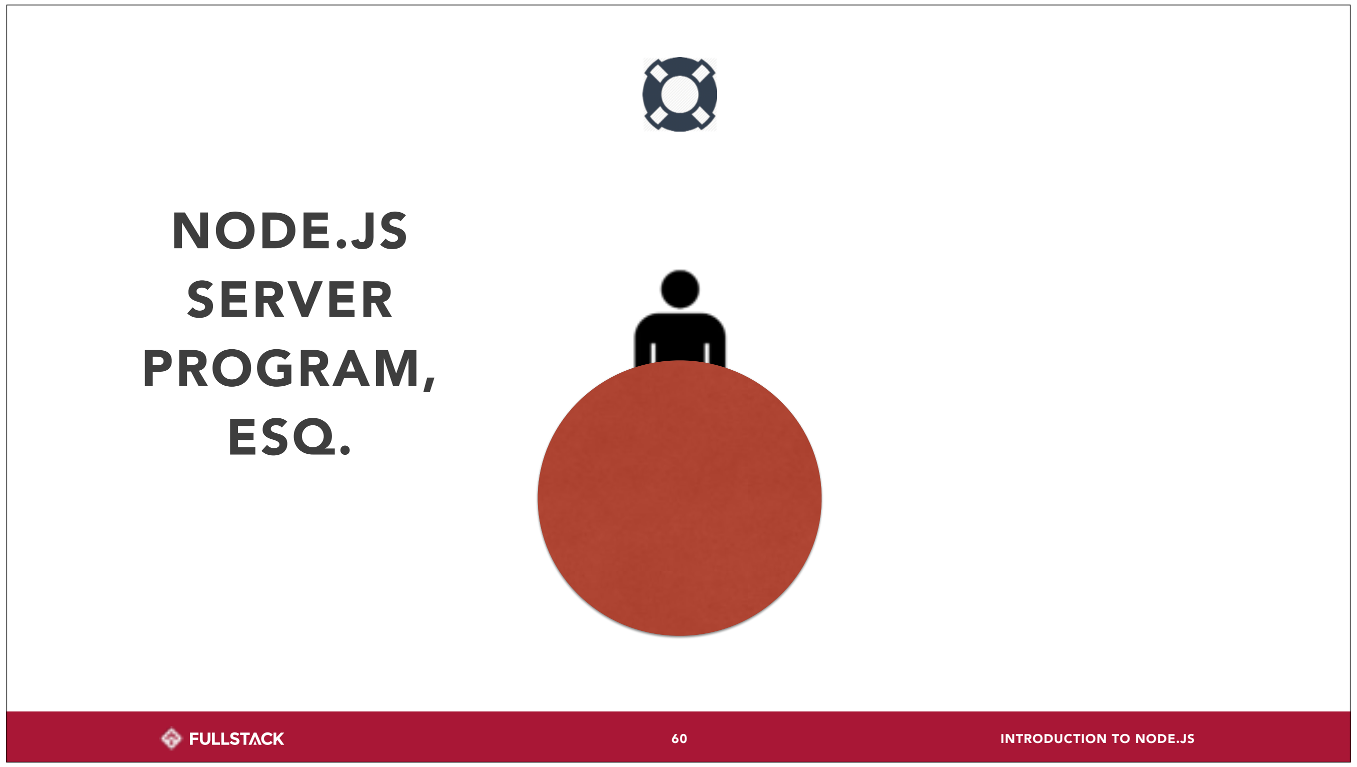
Our non-blocking bank once again has one teller.



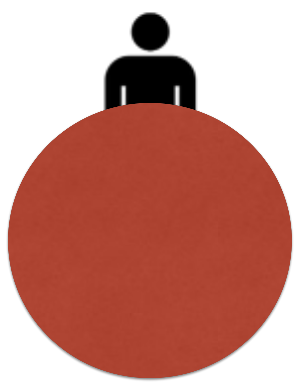
NODE.JS SERVER PROGRAM



This teller’s name is Mr. Node.js Server Program.

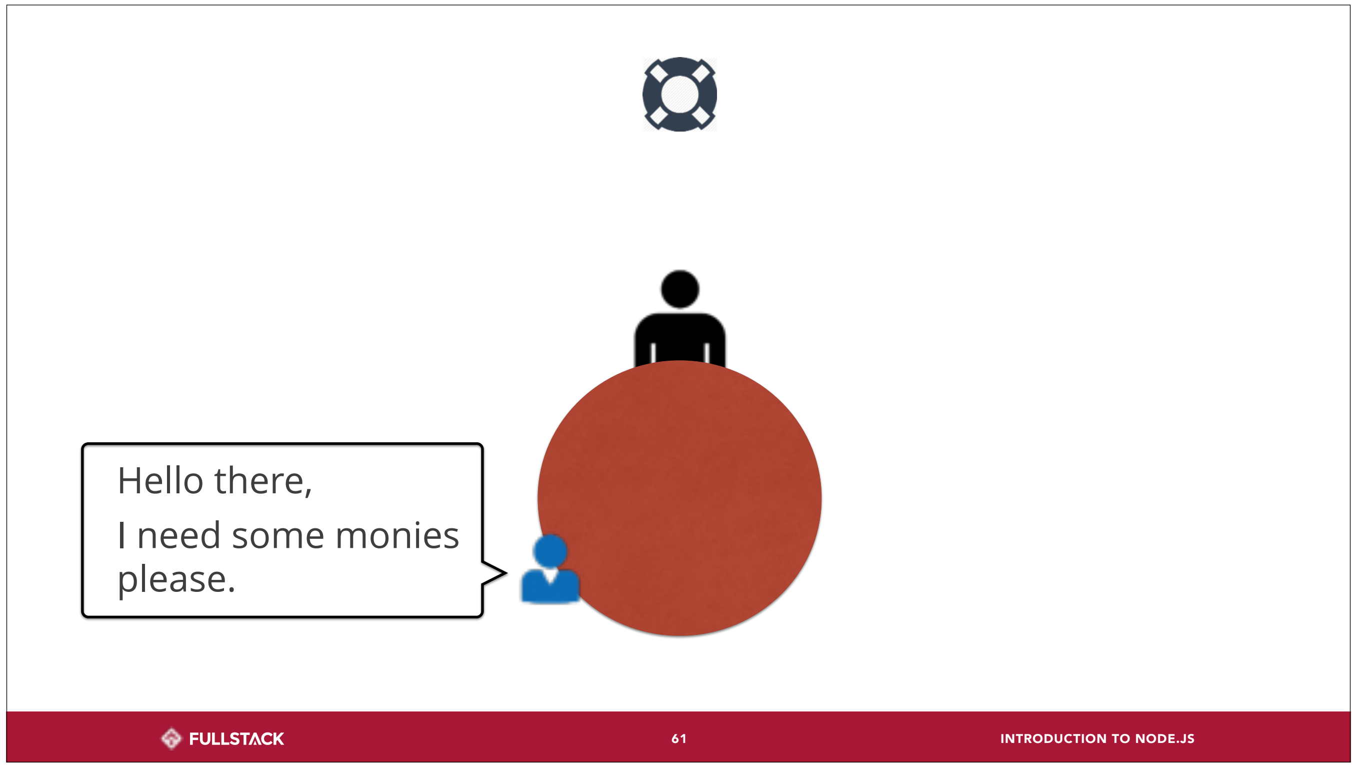


NODE.JS SERVER PROGRAM, ESQ.

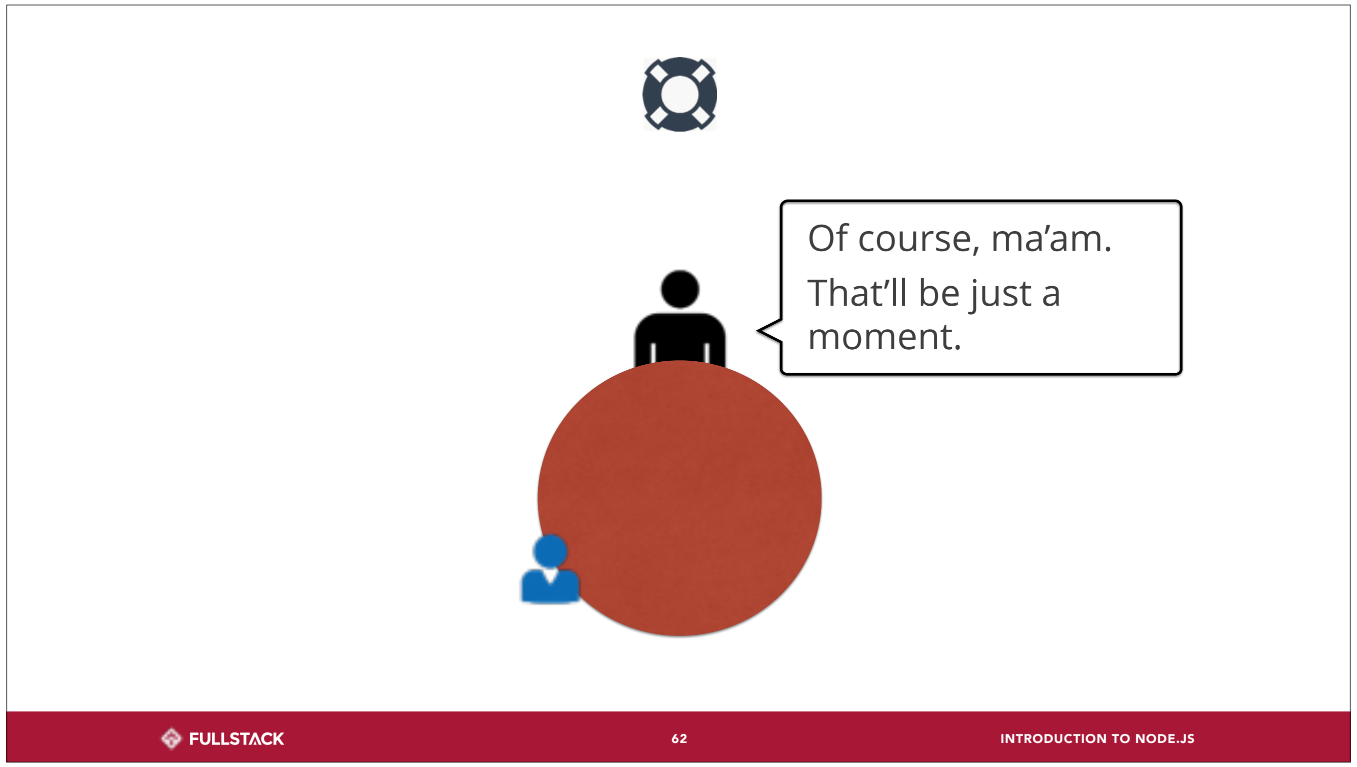


Well, Mr. Node.js Server Program, Esquire. He’s a pretty fancy guy.

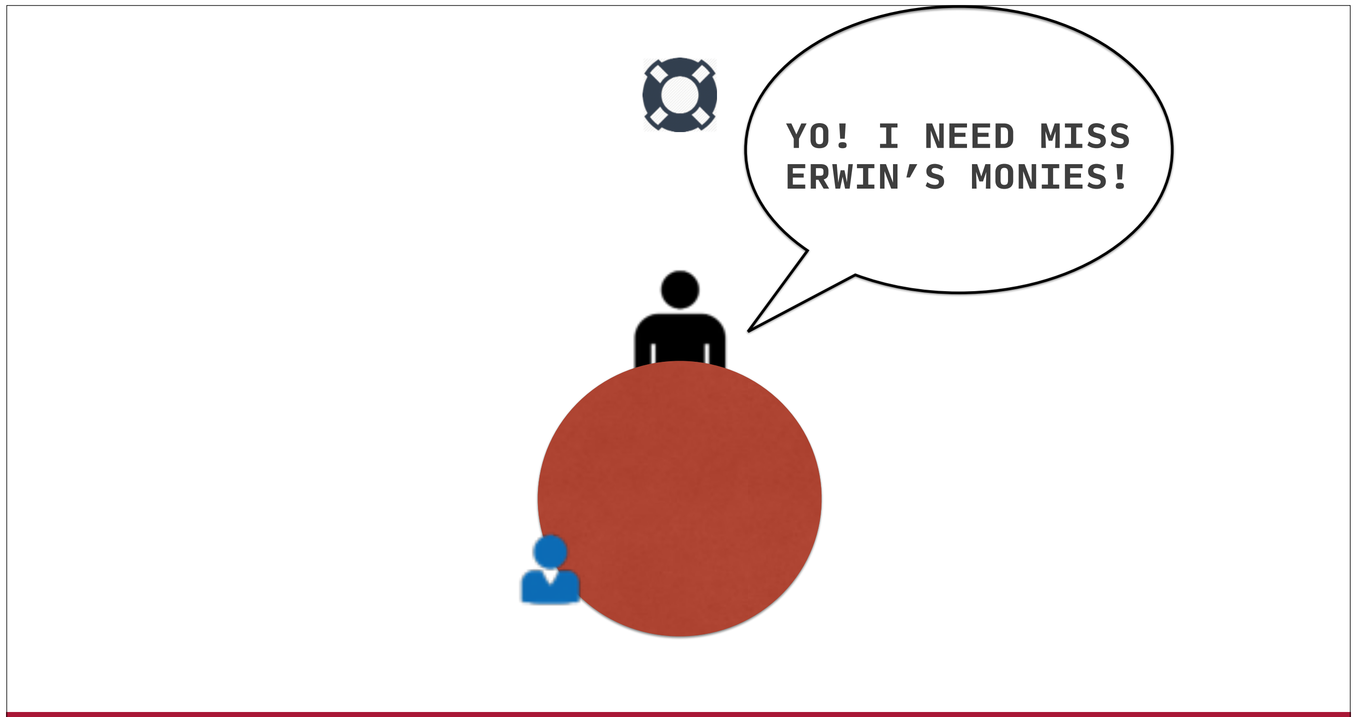
Mr. Node.js Server Program is a smooth talker, and very fast talker, and he sits at a large table, waiting for customers.



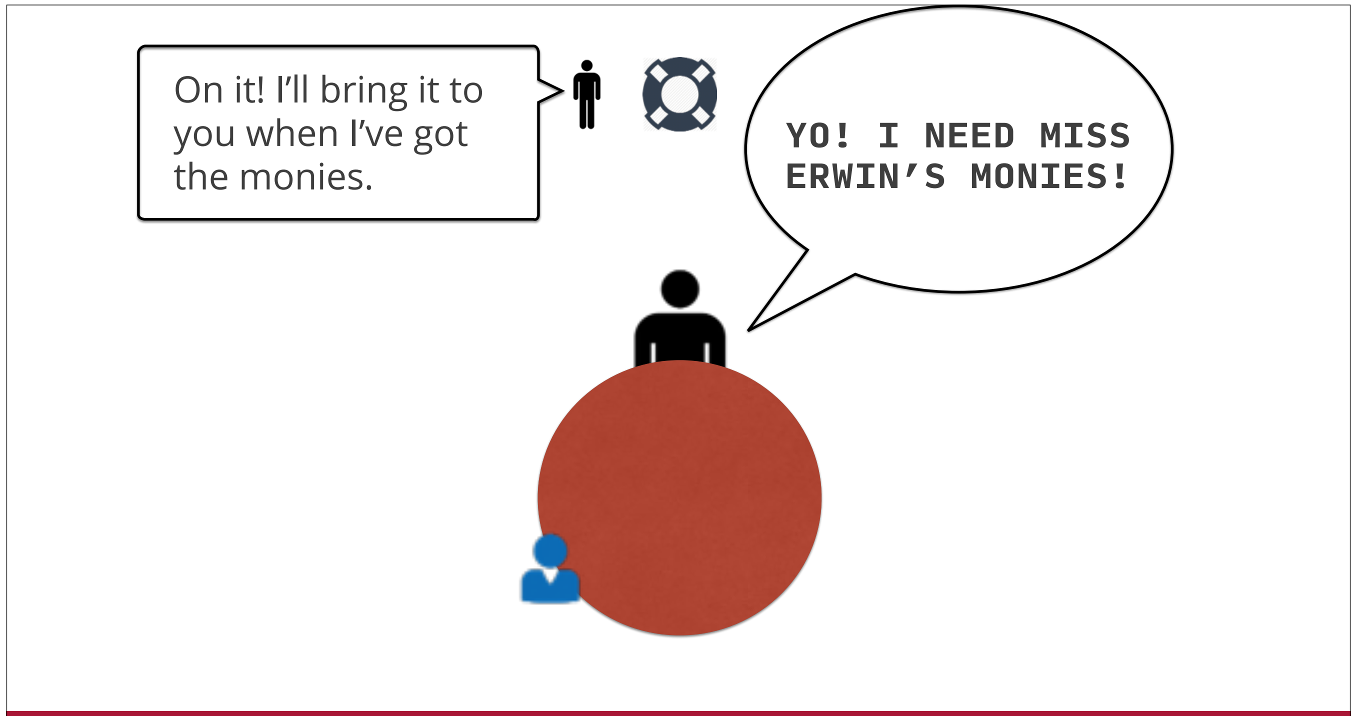
Like this one. This is Miss Erwin and she’s just switched banks. She, once again, wants her monies.



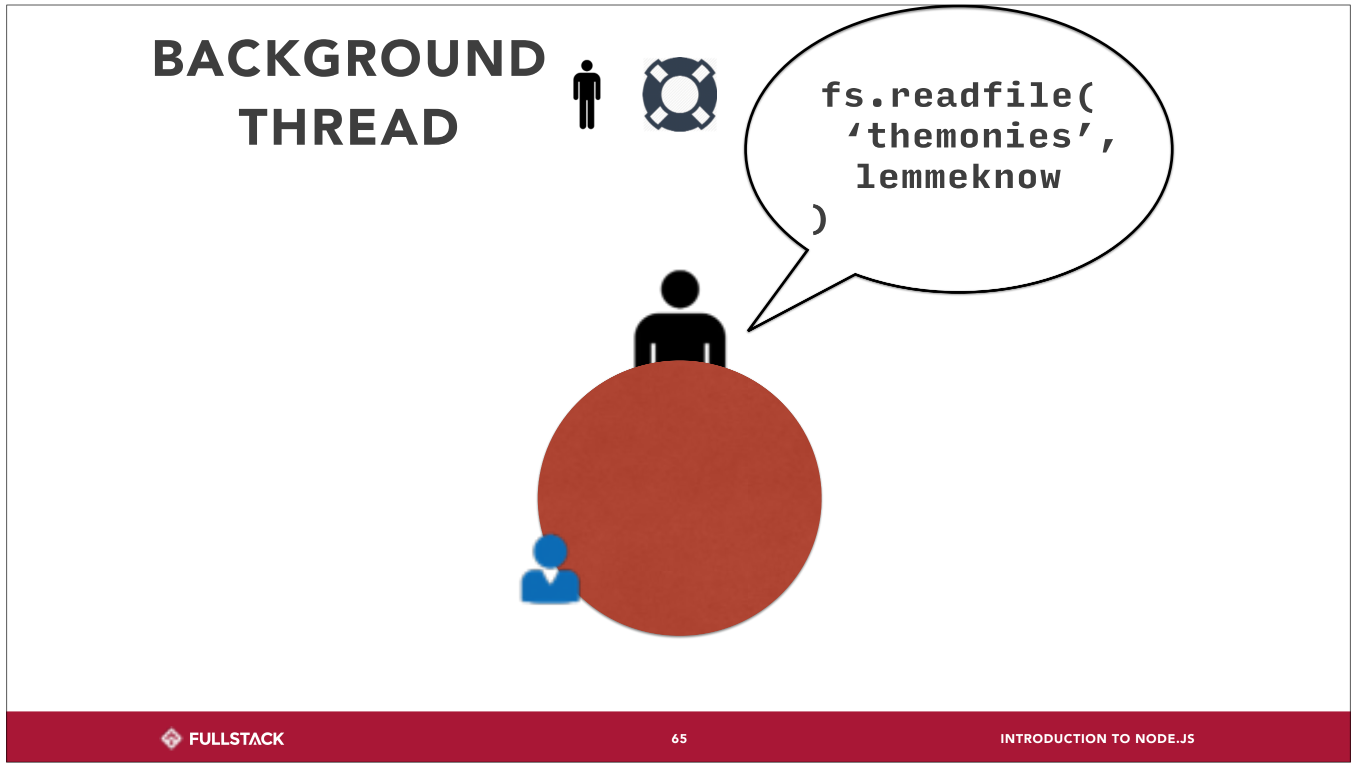
Mr. Node.js Server Program acknowledges her request for monies. But, contrary to our blocking bank . . .



He yells out to one of the workers in the background near the vault, [...].


On it! I'll bring it to you when I've got the monies.
YO! I NEED MISS ERWIN'S MONIES!

Brian, who works in the back, hears the request for resources and informs Mr. Node.js Server Program that he'll bring Mr. Program the resource once he's found it.



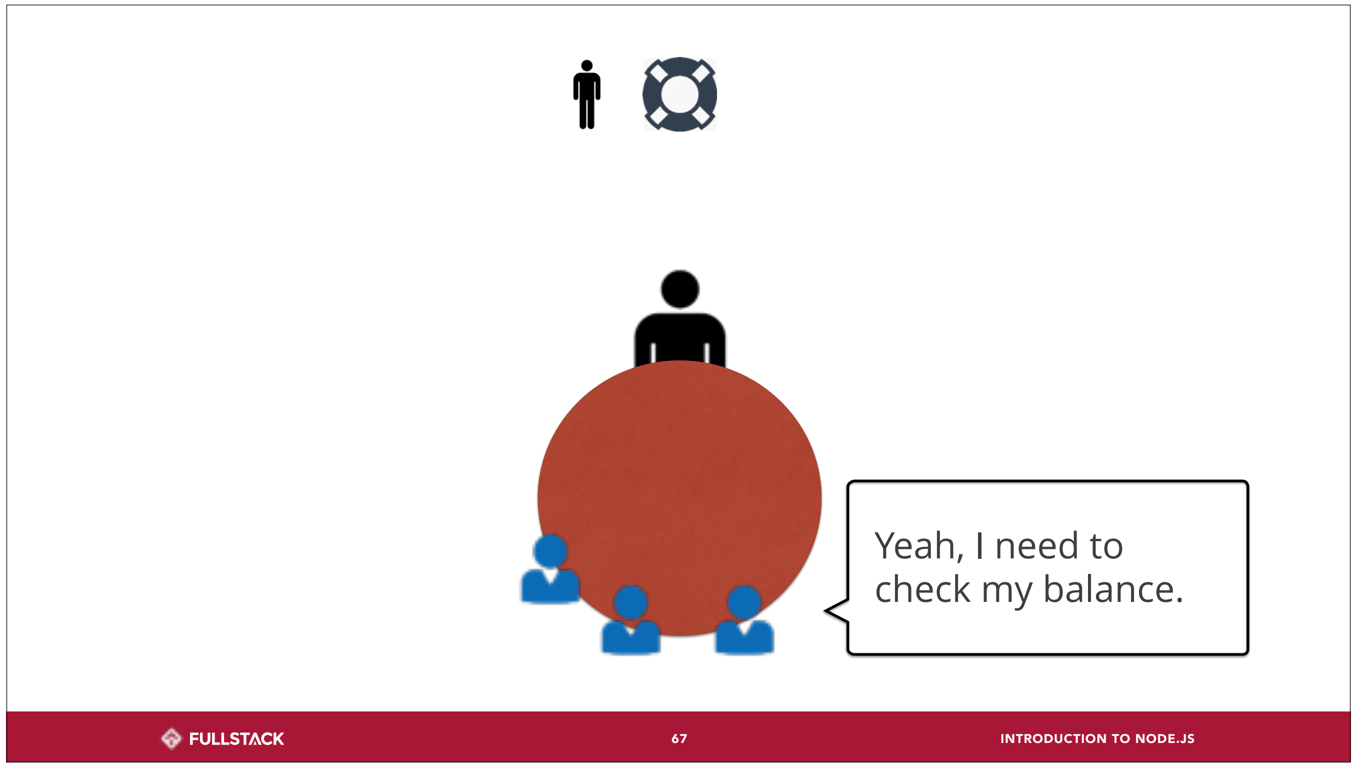
BACKGROUND THREAD

```
fs.readfile(
  ‘themonies’,
   lemmeknow
)
```

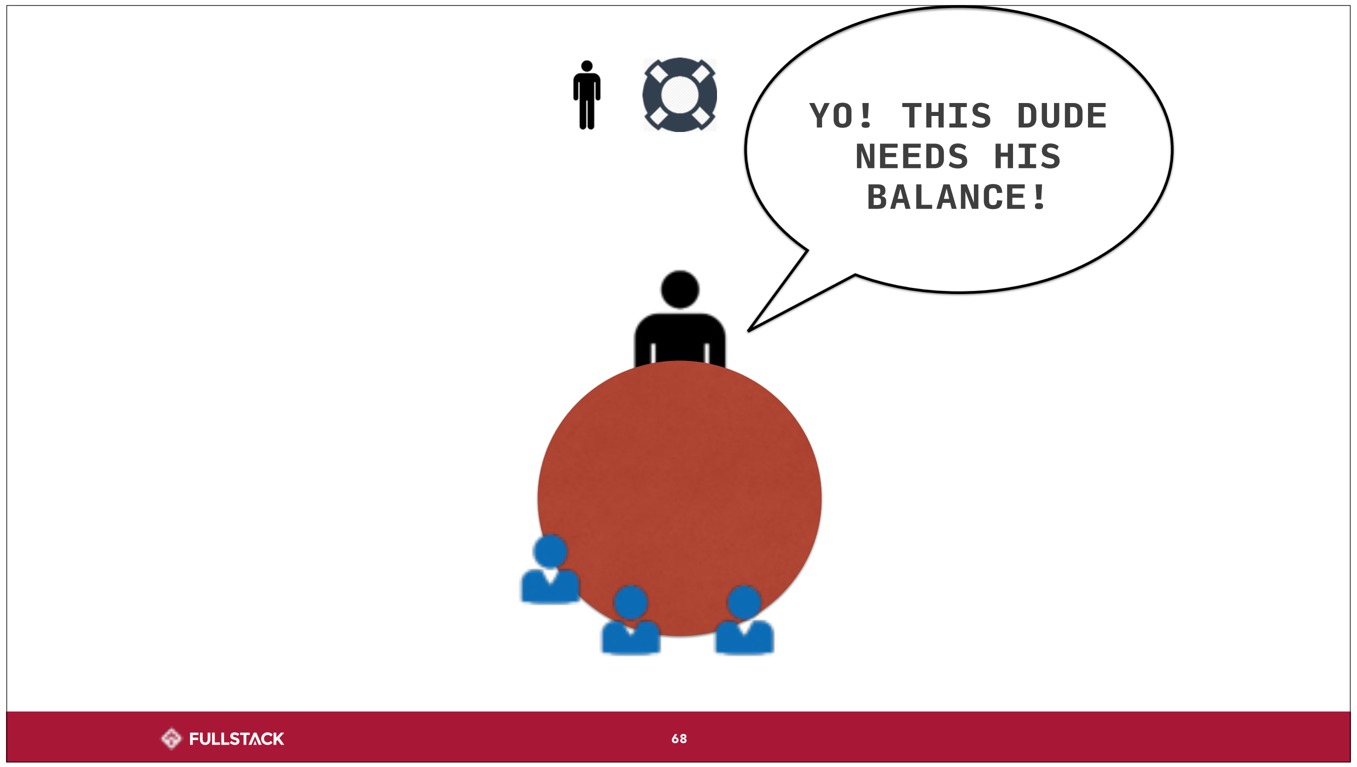
FULLSTACK 65 INTRODUCTION TO NODE.JS

This is fs.readfile. The main program doesn’t pause to go grab the resource itself; there are operations—helpers—available to do that work. Once the resource is accessed or something goes wrong, Mr. Program will be notified by that background operation.

This frees up Mr. Program to assist other customers while Brian fetches Miss Erwin’s monies.



Another request is given and acknowledged.

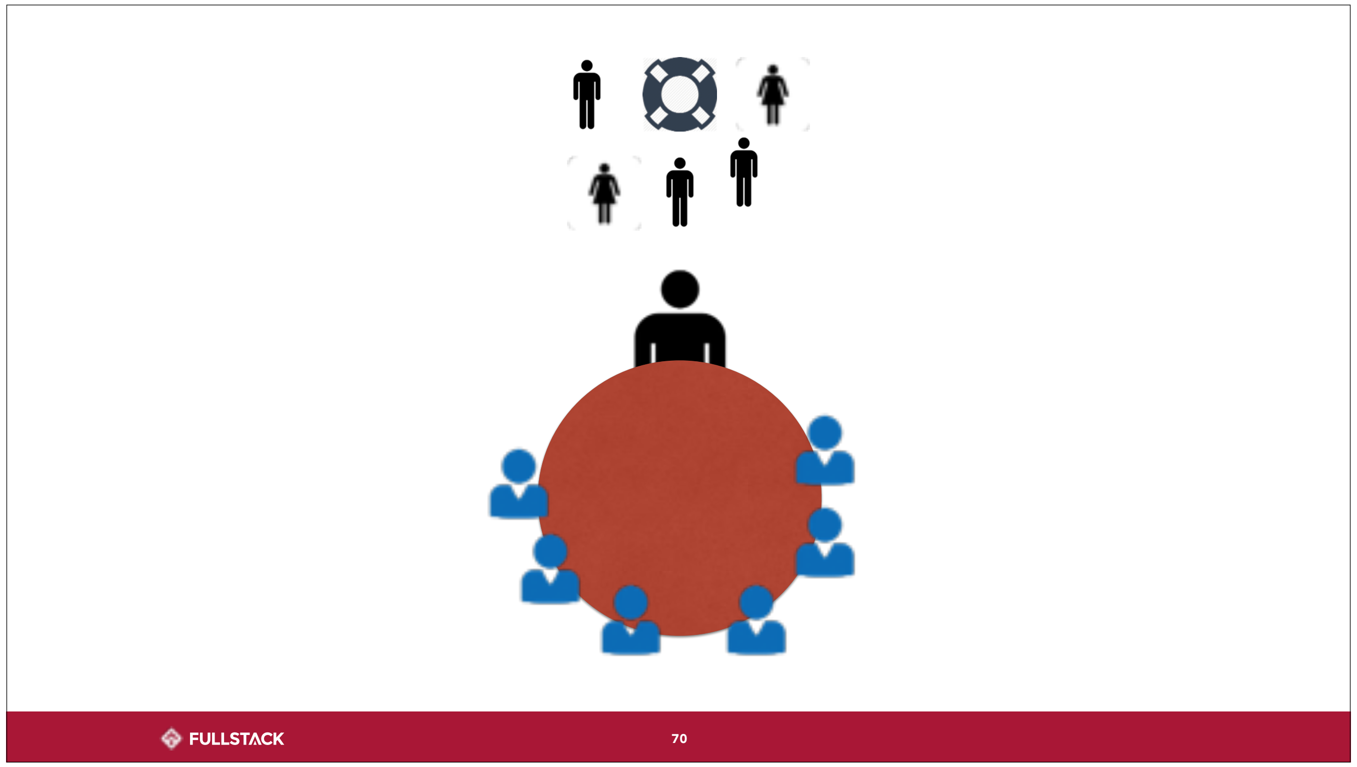


And Mr. Program, knowing he has help behind him, calls out for someone …

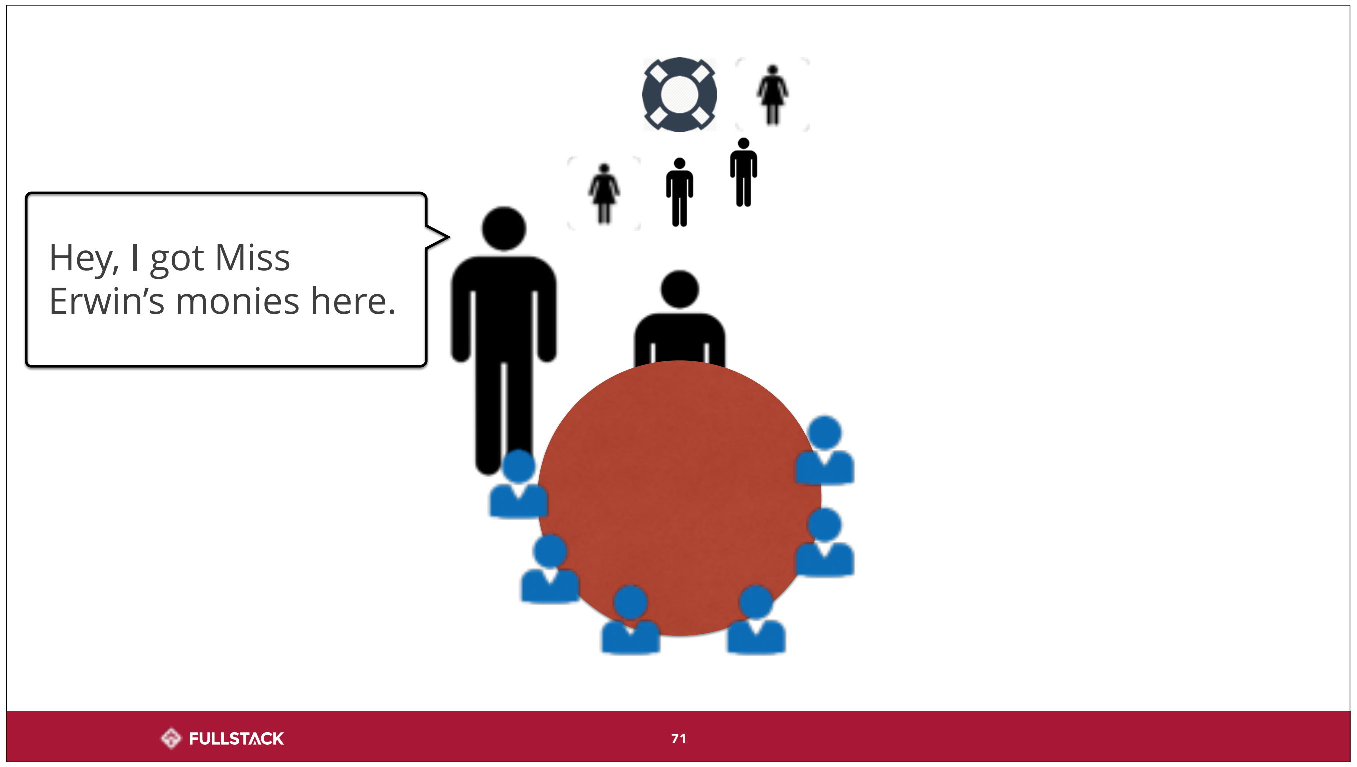
Yup, lookin' for it. I'll bring it to you.

FULLSTACK

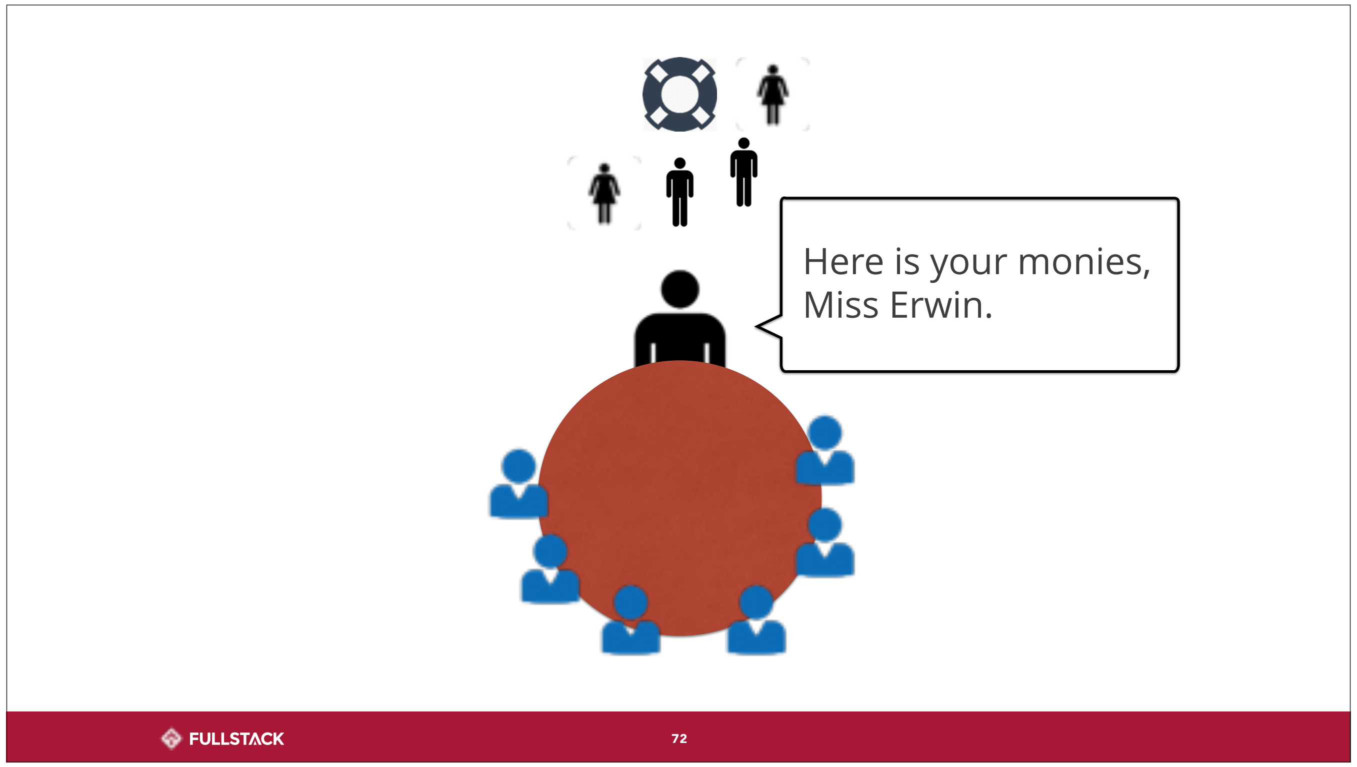
like Casey, who will count the money and bring back the balance to Mr. Program when she knows it.



Free from actually having to do any legwork, Mr. Program can acknowledge a large amount of customers and their needs while background workers grab information and resources. He is even able to hear multiple requests from the same customer and have workers fulfill those requests.

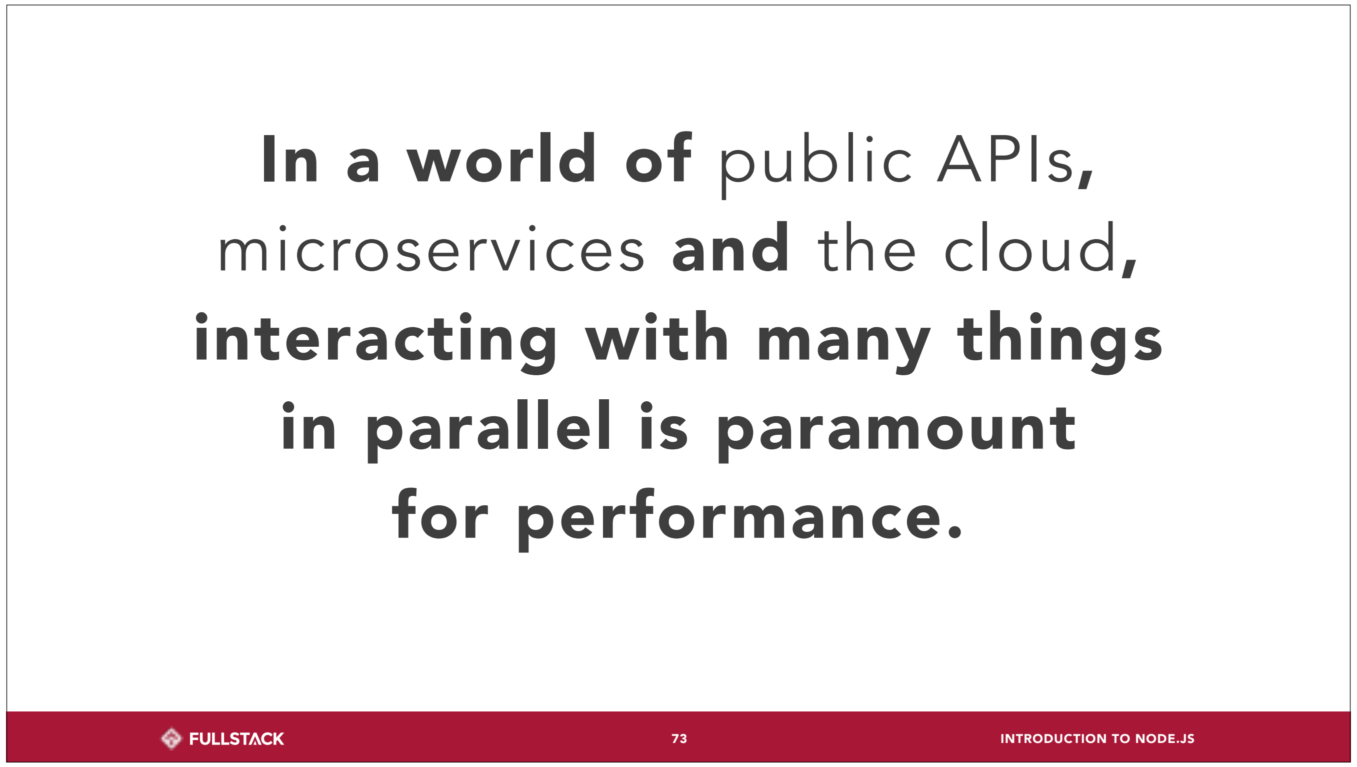


Brian has fetched all of Miss Erwin’s monies and brings it back to Mr. Program. Brian goes and lounges until Mr. Program needs his help again.



And Mr. Program hands Miss Erwin her money.

This is an analogy of what active models for parallelism can do for your technology.



In a world of public APIs**,**
microservices **and** the cloud**,**
interacting with many things
in parallel is paramount
for performance.

[…]

If your program needs to reach out to many resources before execution is completed, and some of those resources can be accessed independent of each other, it is extremely powerful to be able to have the ability to do those operations at the same time, rather than sequentially. Especially since network actions are extremely expensive.

Google NETFLIX

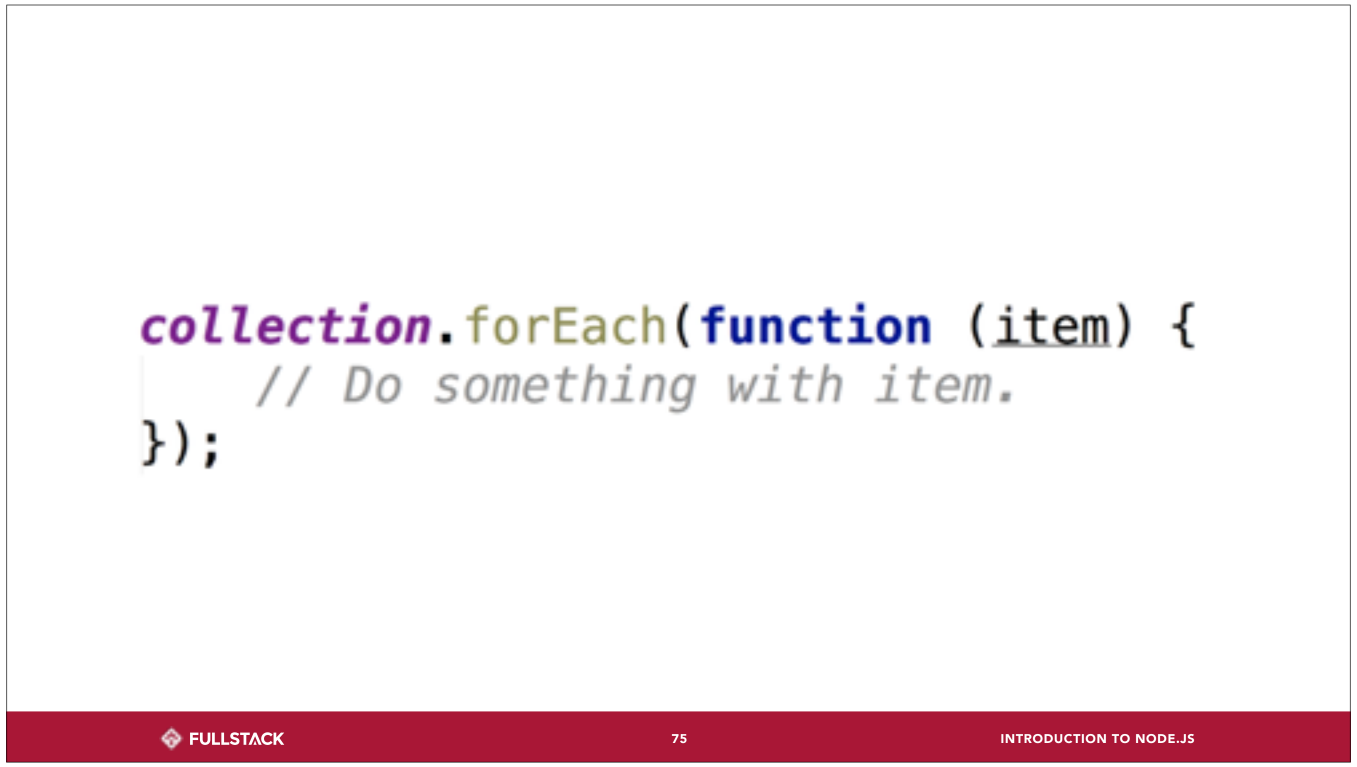
PayPal™ Microsoft®

The New York Times Walmart

This is why high-profile companies are incorporating node in their technology stacks.

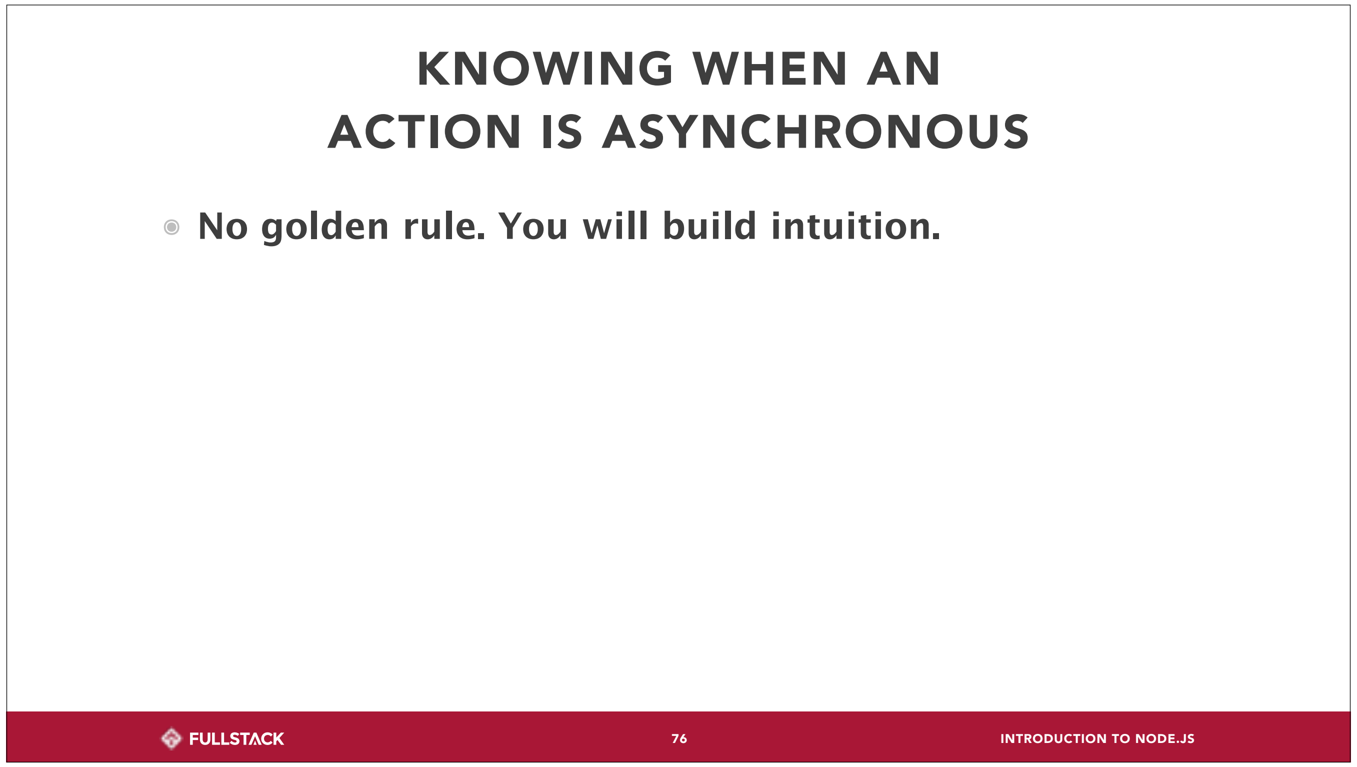
———————

Asynchronous execution is powerful, but unfortunately, this power comes with large complexity.



```
collection.forEach(function (item) {
    // Do something with item.
});
```

Is this operation asynchronous? No, this operation completes fully before the next line of the program will be evaluated, even though it seems to take a callback.

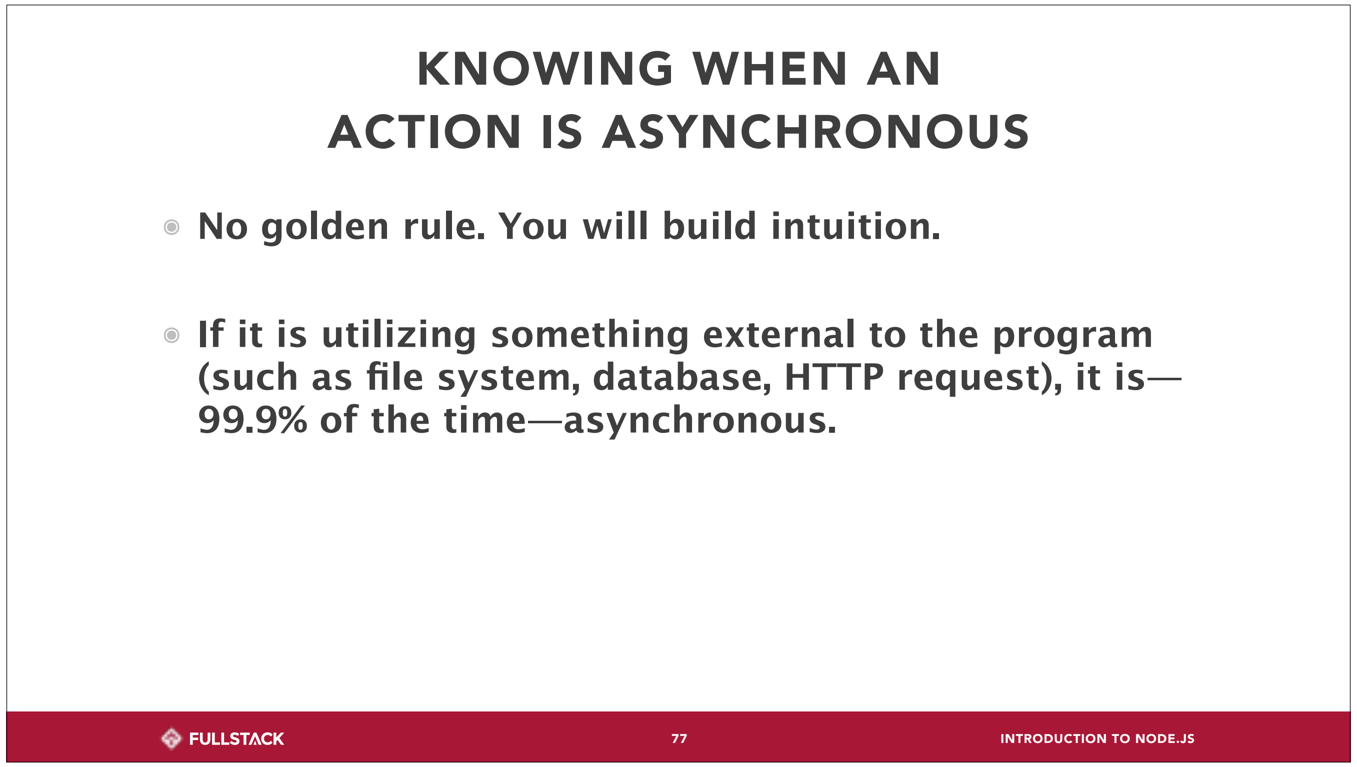


KNOWING WHEN AN ACTION IS ASYNCHRONOUS

- **No golden rule. You will build intuition.**

So how do we know when something is an asynchronous action? How do we know when an action will delegate to the background and return at some later, unknown time?

Well, there is no golden rule. As you learn and practice and code a lot, you will build intuition about what is asynchronous and what is not.

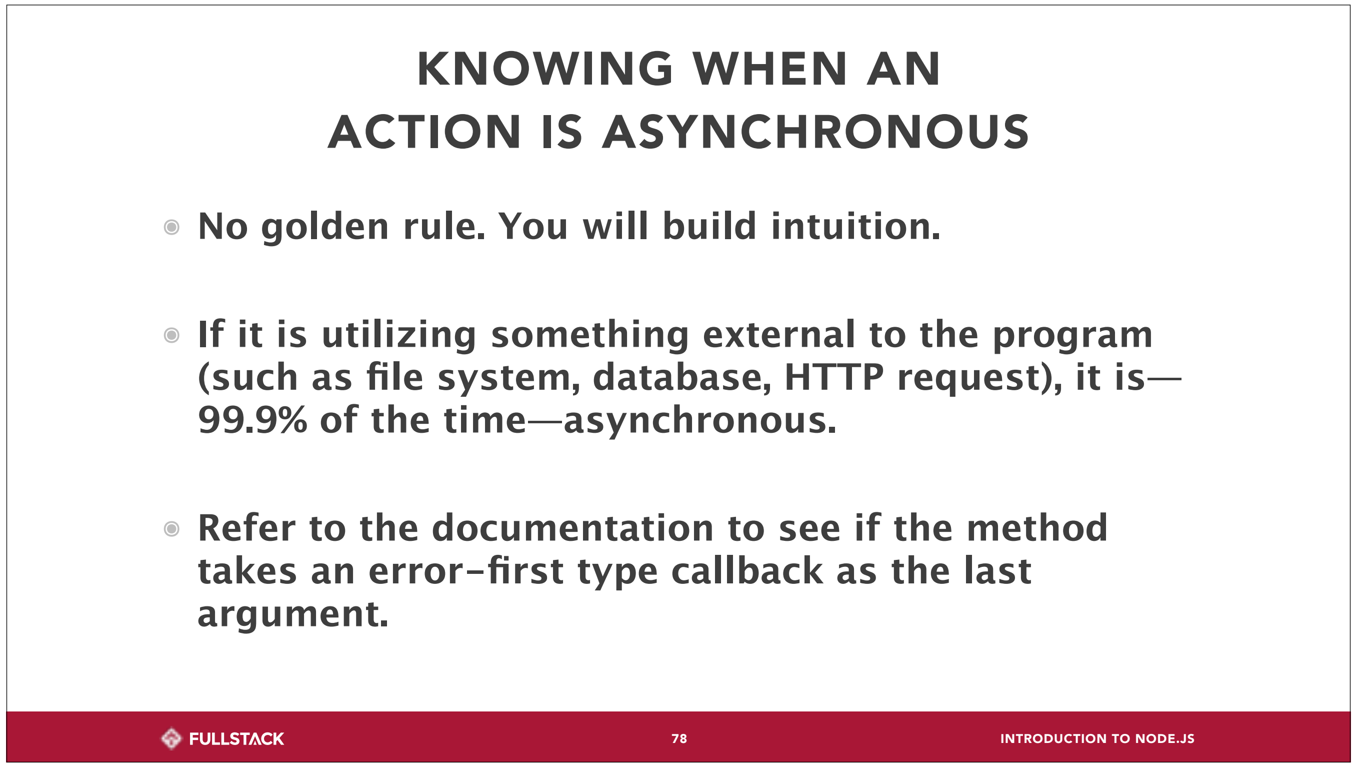


KNOWING WHEN AN ACTION IS ASYNCHRONOUS

- No golden rule. You will build intuition.
- If it is utilizing something external to the program (such as file system, database, HTTP request), it is—99.9% of the time—asynchronous.

FULLSTACK 77 INTRODUCTION TO NODE.JS

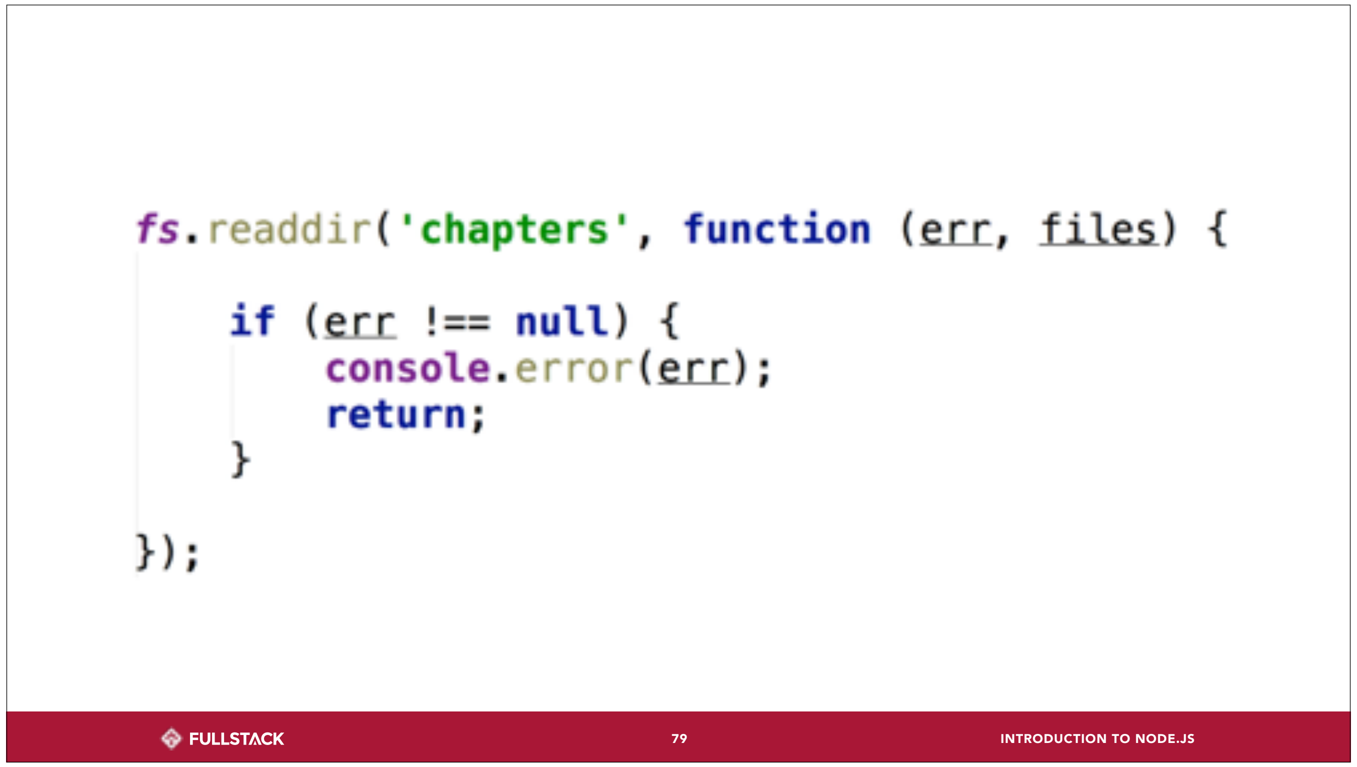
There are some pretty solid gotos though. If the operation is accessing or utilizing in any way a resource external to the program—such as the file system or database or HTTP request—it is going to be, nearly every time, asynchronous.



KNOWING WHEN AN ACTION IS ASYNCHRONOUS

- No golden rule. You will build intuition.
- If it is utilizing something external to the program (such as file system, database, HTTP request), it is—99.9% of the time—asynchronous.
- Refer to the documentation to see if the method takes an error-first type callback as the last argument.

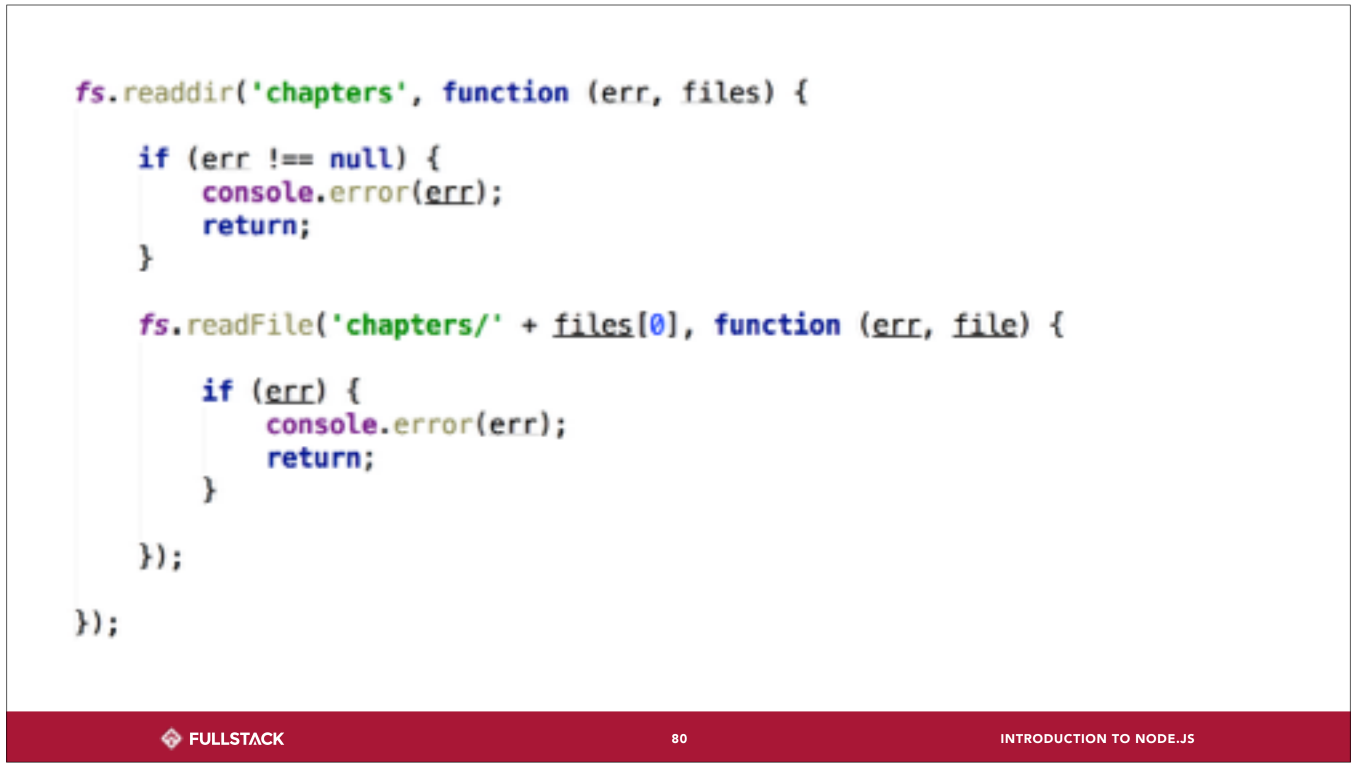
You can also tell by documentation if the operation if the operation will be asynchronous if you see that the function call takes a callback function as the last parameter. And if this callback receives the possibility of an error as the first parameter.



```
fs.readdir('chapters', function (err, files) {

    if (err !== null) {
        console.error(err);
        return;
    }

});
```

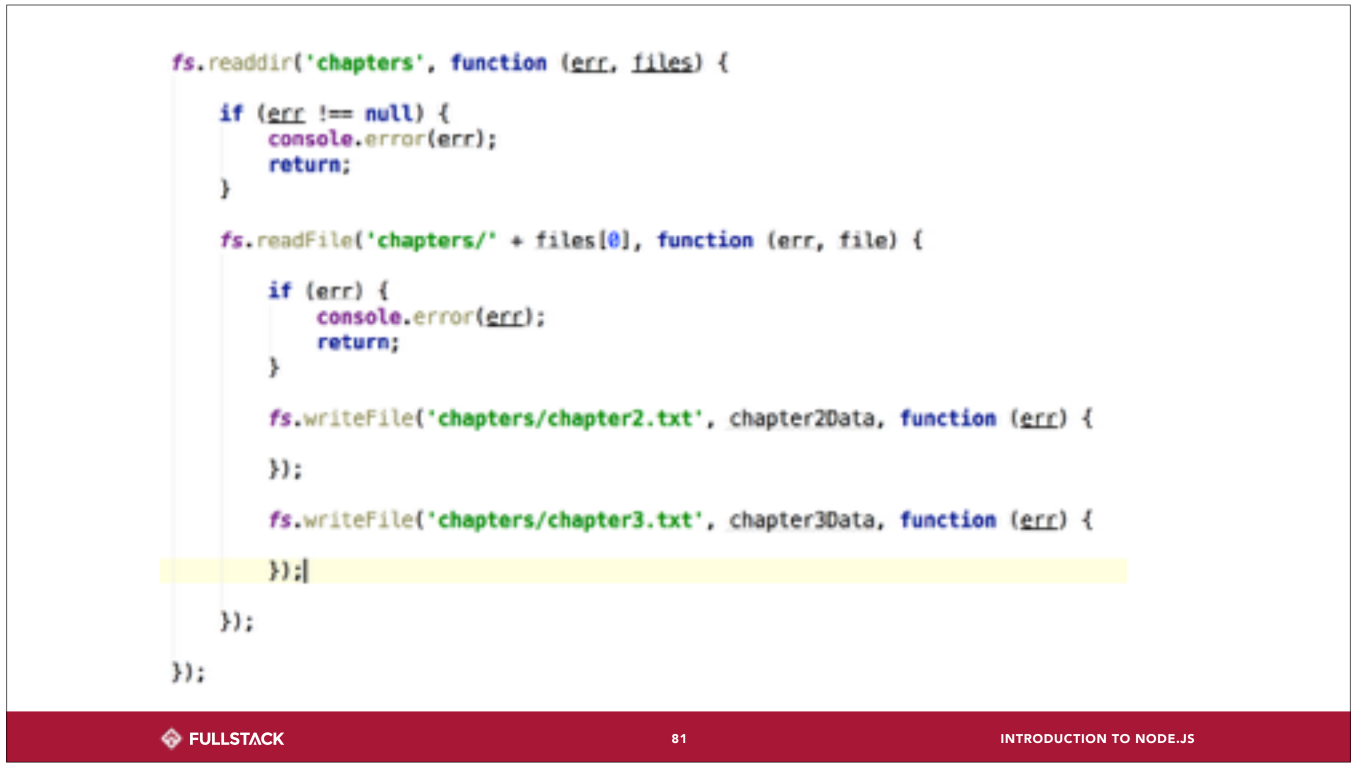
FULLSTACK

Like this method on the fs module called readdir. This method reads a directory and returns all of the file names in that directory as an array of strings.



```
fs.readdir('chapters', function (err, files) {

    if (err !== null) {
        console.error(err);
        return;
    }

    fs.readFile('chapters/' + files[0], function (err, file) {

        if (err) {
            console.error(err);
            return;
        }

    });

});
```

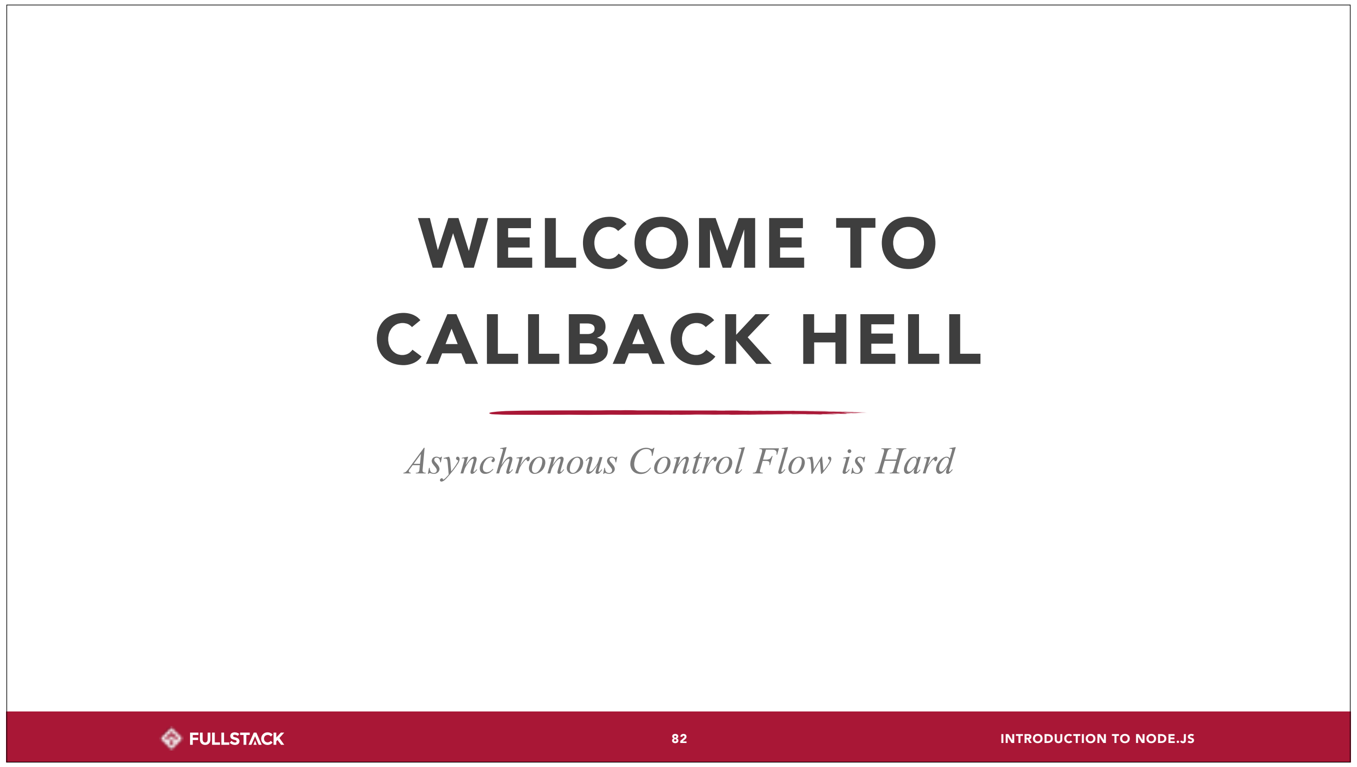
By the way, let’s say we read that directory for all the file names, and then when those files names are returned, we read the contents of the first file.



```
fs.readdir('chapters', function (err, files) {

    if (err !== null) {
        console.error(err);
        return;
    }

    fs.readFile('chapters/' + files[0], function (err, file) {

        if (err) {
            console.error(err);
            return;
        }

        fs.writeFile('chapters/chapter2.txt', chapter2Data, function (err) {

        });

        fs.writeFile('chapters/chapter3.txt', chapter3Data, function (err) {

        });

    });

});
```

And then when we get that first file, we make two parallel requests to write some data to two new files.

This code is really messy, and we have to catch errors from the file writes. And we need to somehow know when both of the file writes have been completed — rather than just one.



WELCOME TO CALLBACK HELL

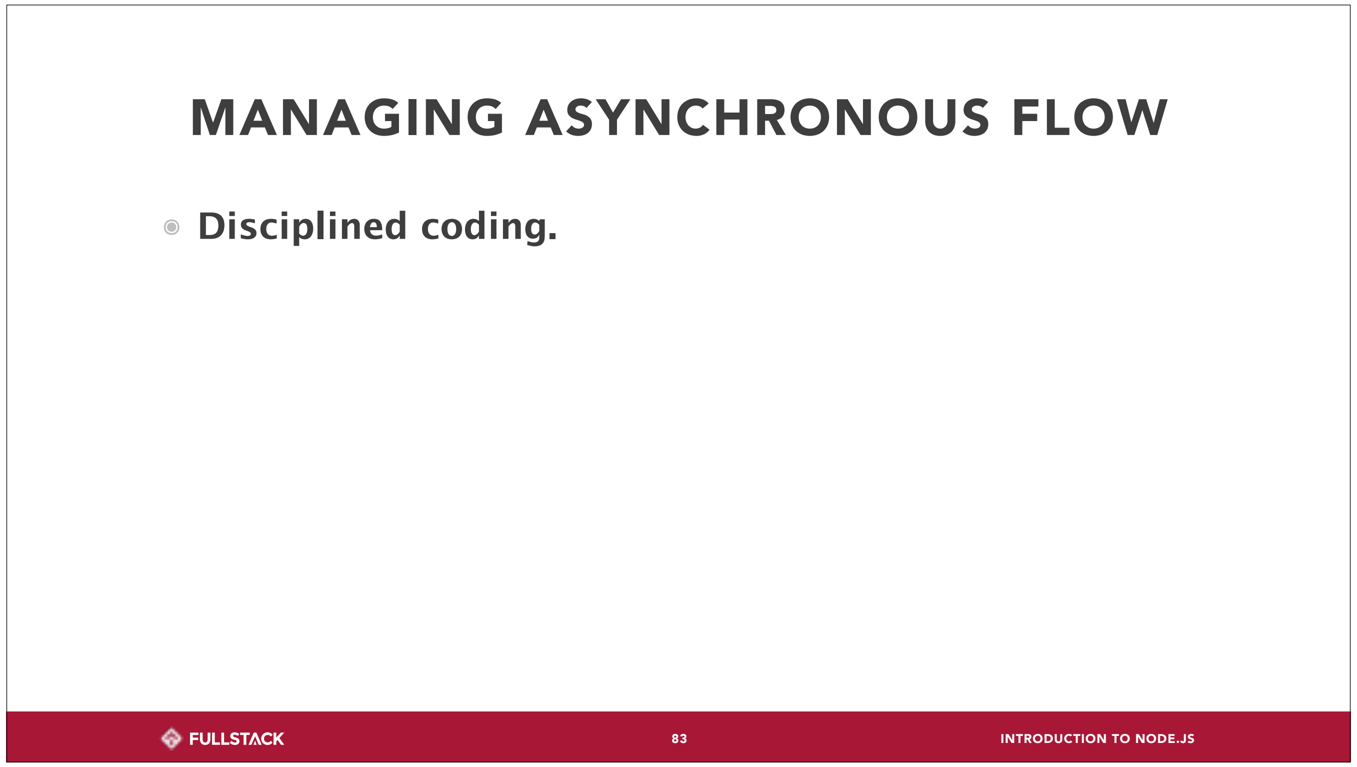
Asynchronous Control Flow is Hard

FULLSTACK 82 INTRODUCTION TO NODE.JS

[…]

Like I've mentioned, dealing with asynchronicity and parallel operations can get really hard and it will take some time, patience and frustration to wrap your head around managing it, mentally and in your written code.

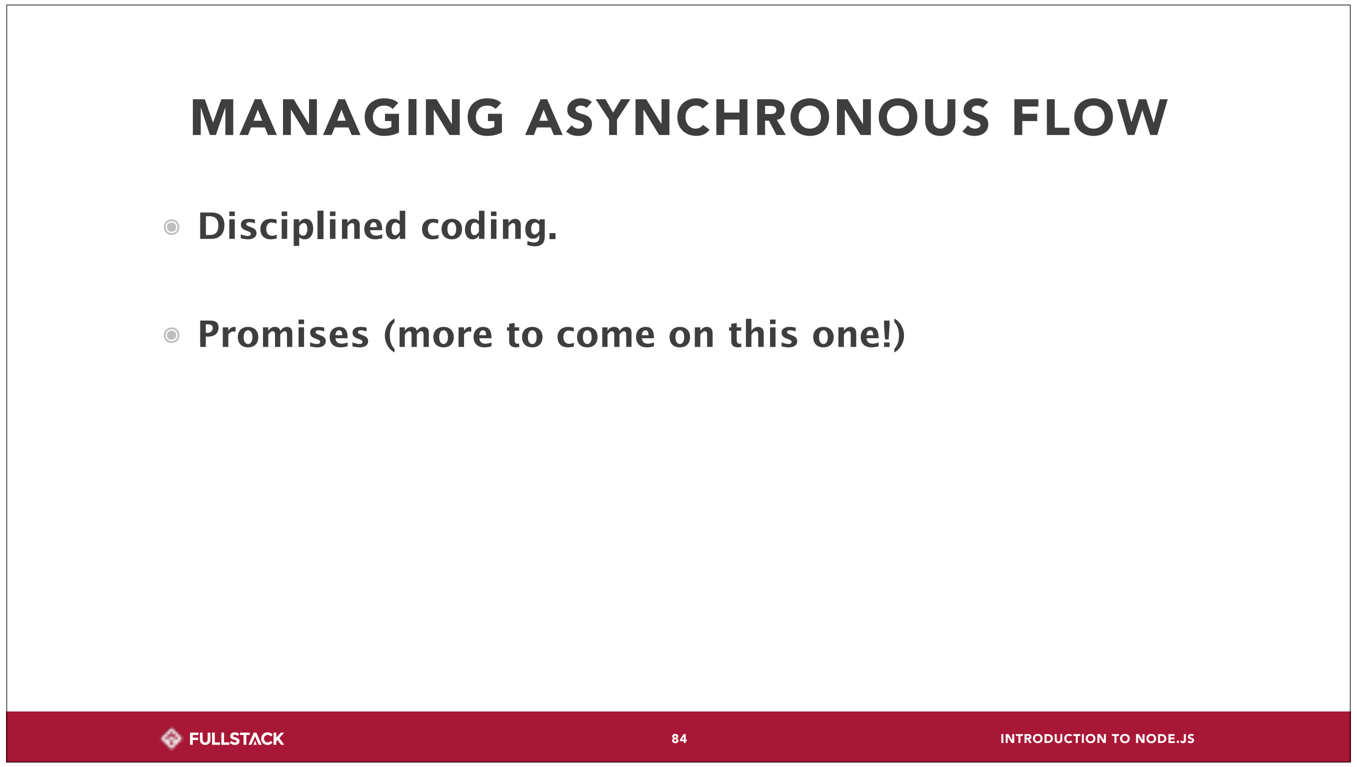
Here are some suggestions to make it possibly easier:



MANAGING ASYNCHRONOUS FLOW

- Disciplined coding.

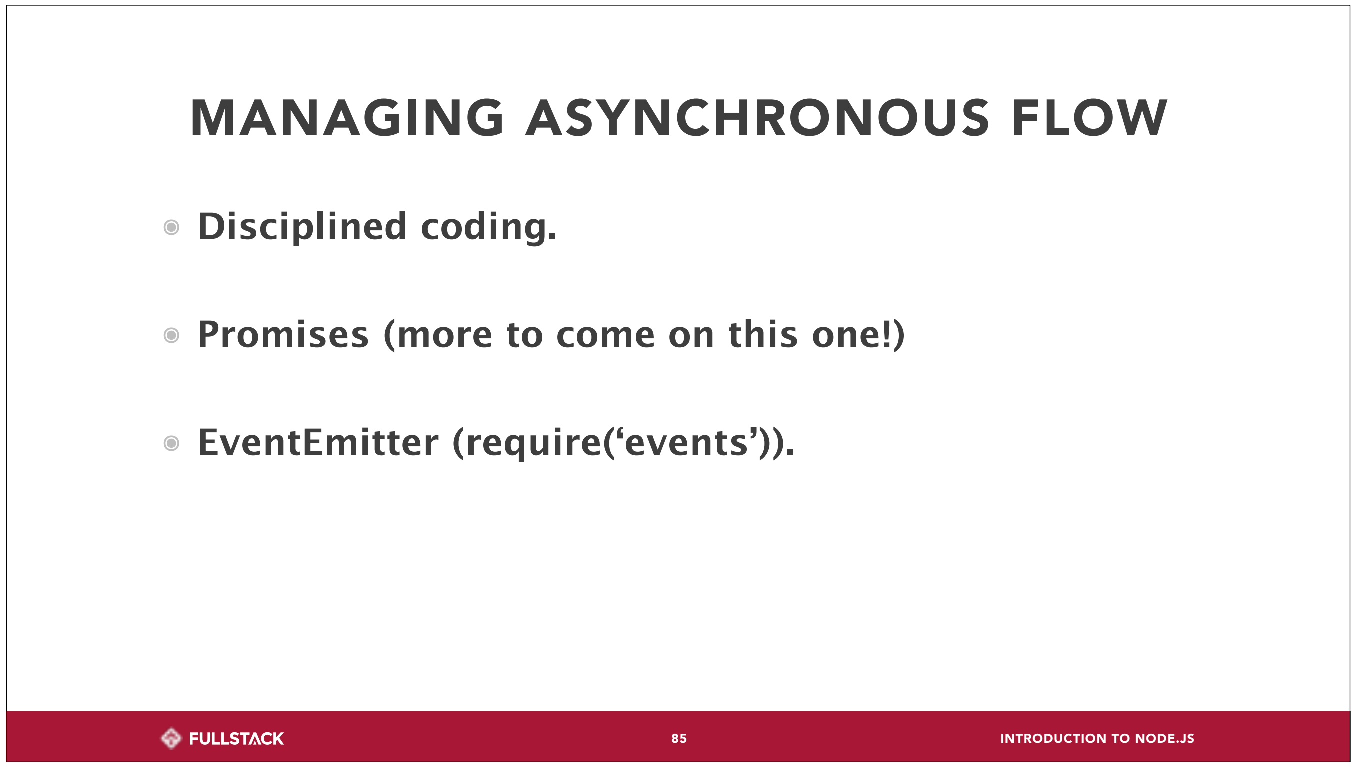
Practice disciplined coding. Try not to move too fast when trying to wrangle async operations. Go slow, write carefully, and separate your code into small logical pieces in order to have more visibility about what’s going on. Creating a new function has no performance detriment.



MANAGING ASYNCHRONOUS FLOW

- Disciplined coding.
- Promises (more to come on this one!)

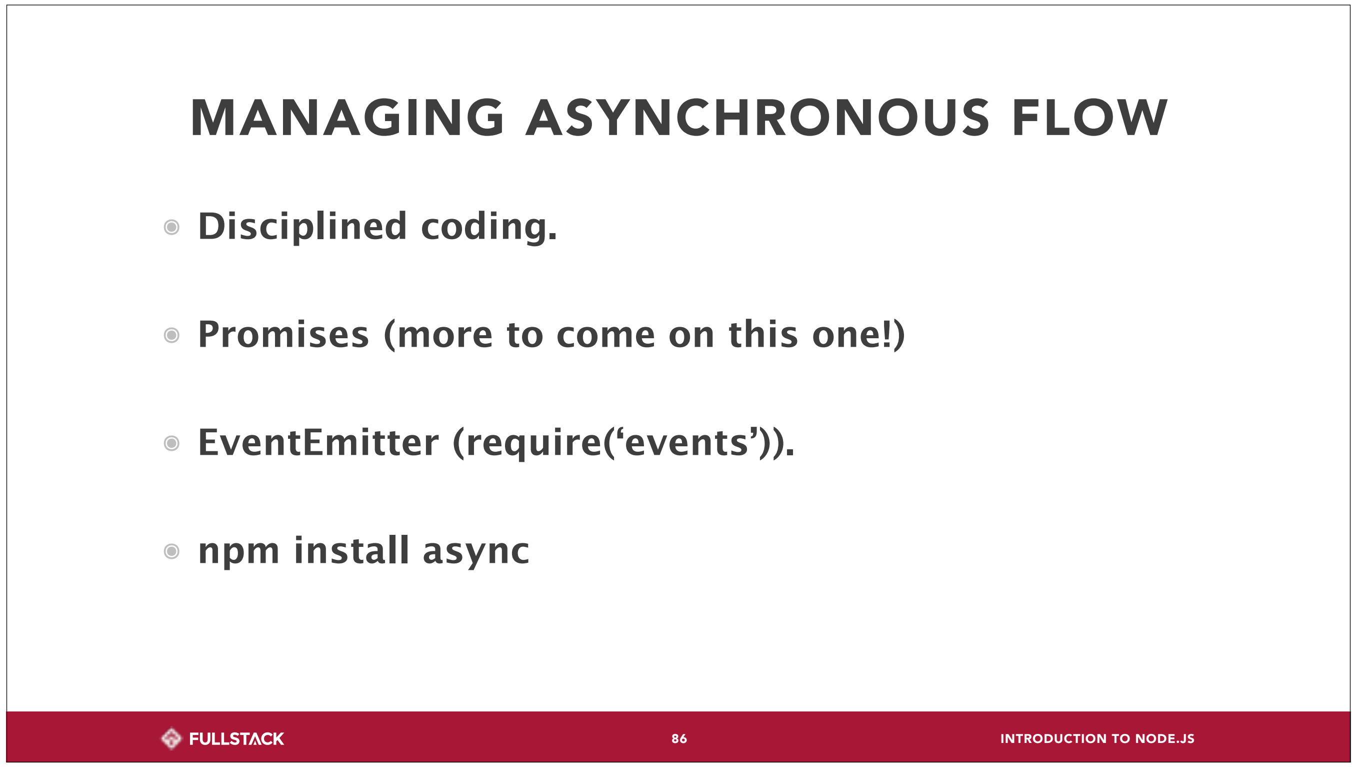
You can use an asynchronous construct called Promises. For now, you can look these up on your own if you're curious. We will be visiting promises later in the curriculum. Promises are just about my favorite thing ever and you will gain a ton of coding power and skill by learning them.



MANAGING ASYNCHRONOUS FLOW

- Disciplined coding.
- Promises (more to come on this one!)
- EventEmitter (require(‘events’)).

Create objects that inherit node’s EventEmitter. This allows your async actions to call named events when things are ready. This can be super powerful and we will be exploring this a bunch in the next lecture.



MANAGING ASYNCHRONOUS FLOW

- Disciplined coding.
- Promises (more to come on this one!)
- EventEmitter (require(‘events’)).
- npm install async

Another possibility is to use the async library. You will be working with this a bit in the workshop, so let’s take a look.