Web Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology

http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

node JS™

PART 1

SERVER SIDE JAVASCRIPT

Outline

1. **Introduction** – What Node is all about
2. **Node Execution Model** – Nodes Event-Driven, Non-Blocking I/O model
3. **Asynchrony in Node** – Events, Callbacks, Promises & Async/Awaits
4. **Node Modules** – The Building Blocks of Node
5. **Express** – A Framework for Node
6. **REST** – The Architectural Style of the Web
7. **API Design** – Exposing Application Functionality
8. **REST in Express** – Leveraging URLs, URI’s and HTTP

Outline

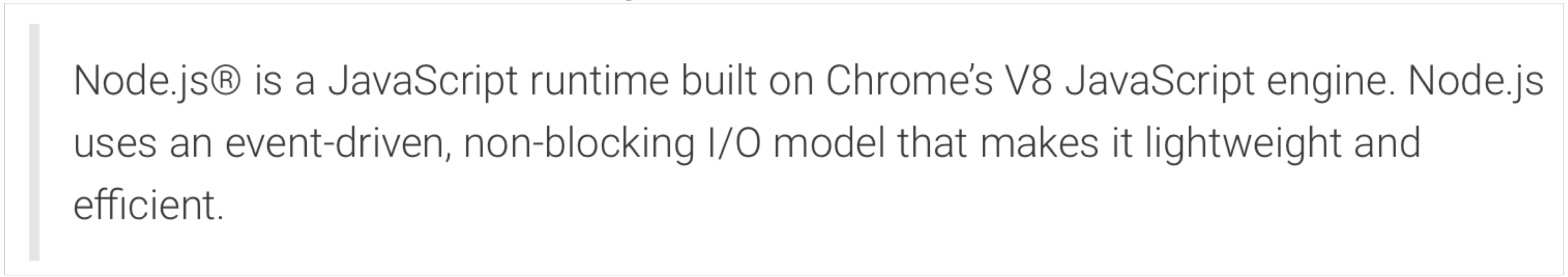
Introduction

WHAT NODE IS ALL ABOUT

So What is Node.js?

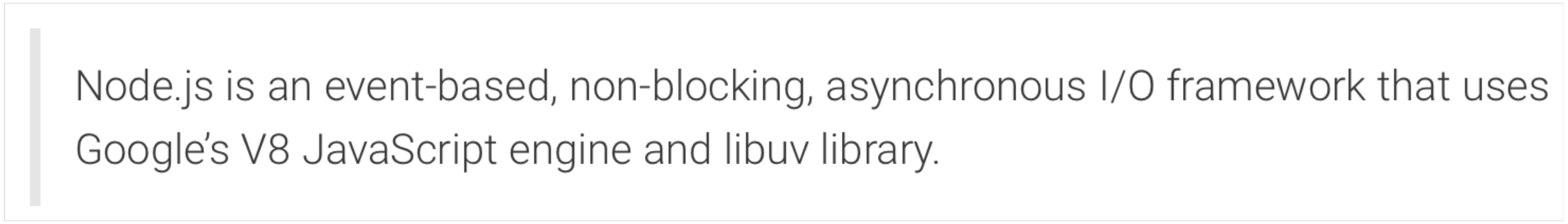
There are plenty of definitions to be found online. Let’s take a look at a couple of the more popular ones:

This is what the project’s home page has to say:



> Node.js® is a JavaScript runtime built on Chrome’s V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

And this is what StackOverflow has to offer: https://stackoverflow.com/tags/node.js/info



> Node.js is an event-based, non-blocking, asynchronous I/O framework that uses Google’s V8 JavaScript engine and libuv library.

So What is Node.js?

“non-blocking I/O”, “event-driven”, “asynchronous” — that’s quite a lot to digest in one go.

So let’s approach this from a different angle and first, begin by focusing on the other detail that both descriptions mention:

the V8 JavaScript engine.

Node Is Built on Google Chrome’s V8 JavaScript Engine

The V8 engine is the open-source JavaScript engine that runs in the Chrome, Opera and Vivaldi browsers. Designed with performance in mind, it’s responsible for compiling JavaScript directly to native machine code that your computer can execute.

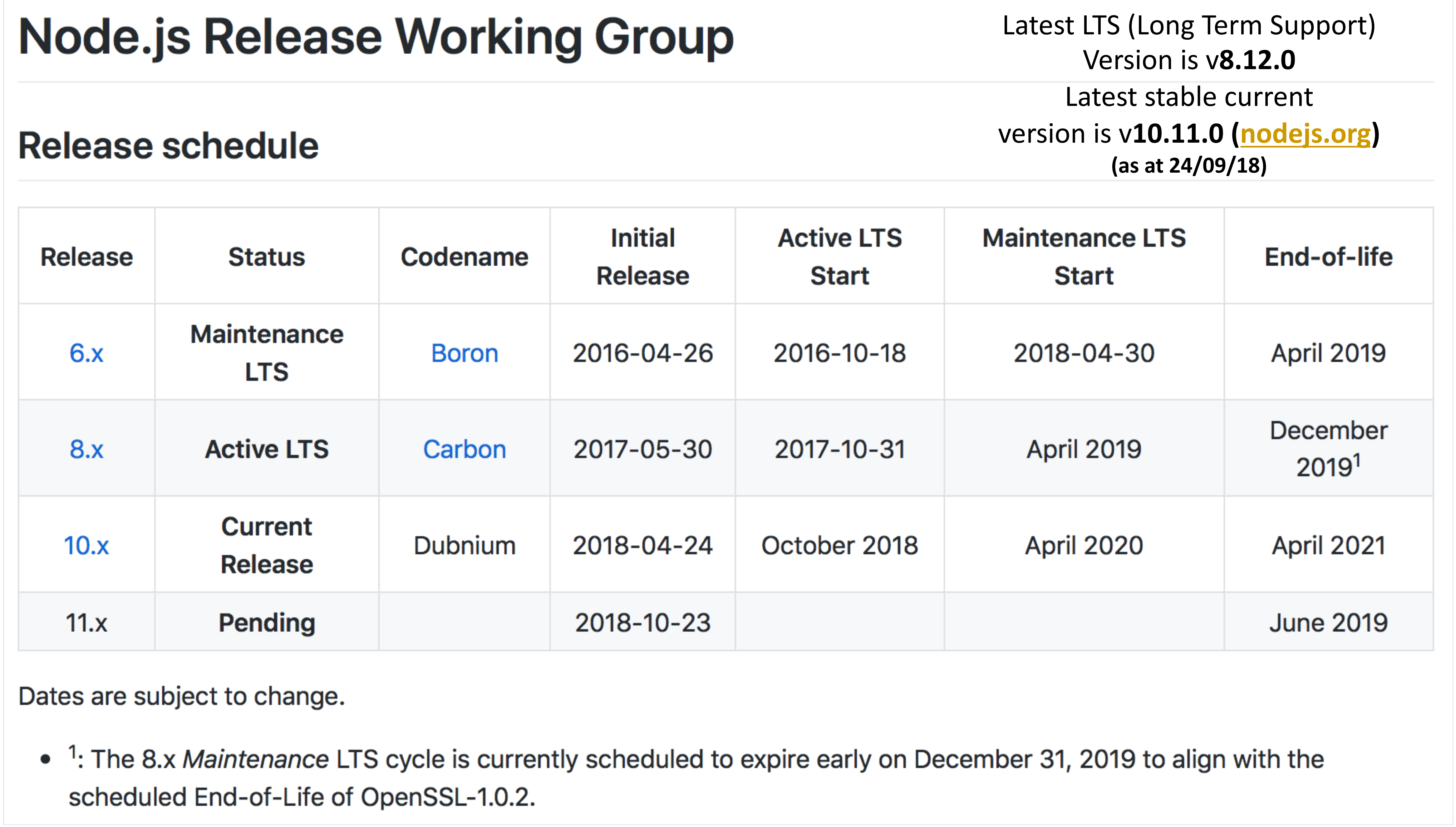
However, when we say that Node is built on the V8 engine, we don’t mean that Node programs are executed in a browser. They aren’t. Rather, the creator of Node (Ryan Dahl 2009) took the V8 engine and enhanced it with various features, such as a **file system API**, an **HTTP library**, and a number of **operating system–related utility methods**.

This means that **Node.js is a program** we can use to execute JavaScript on our computers - In other words, it’s **a JavaScript runtime**.

Latest LTS (Long Term Support) Version is v**8.12.0**

Latest stable current version is v**10.11.0** (nodejs.org)

(as at 24/09/18)



Node.js Release Working Group

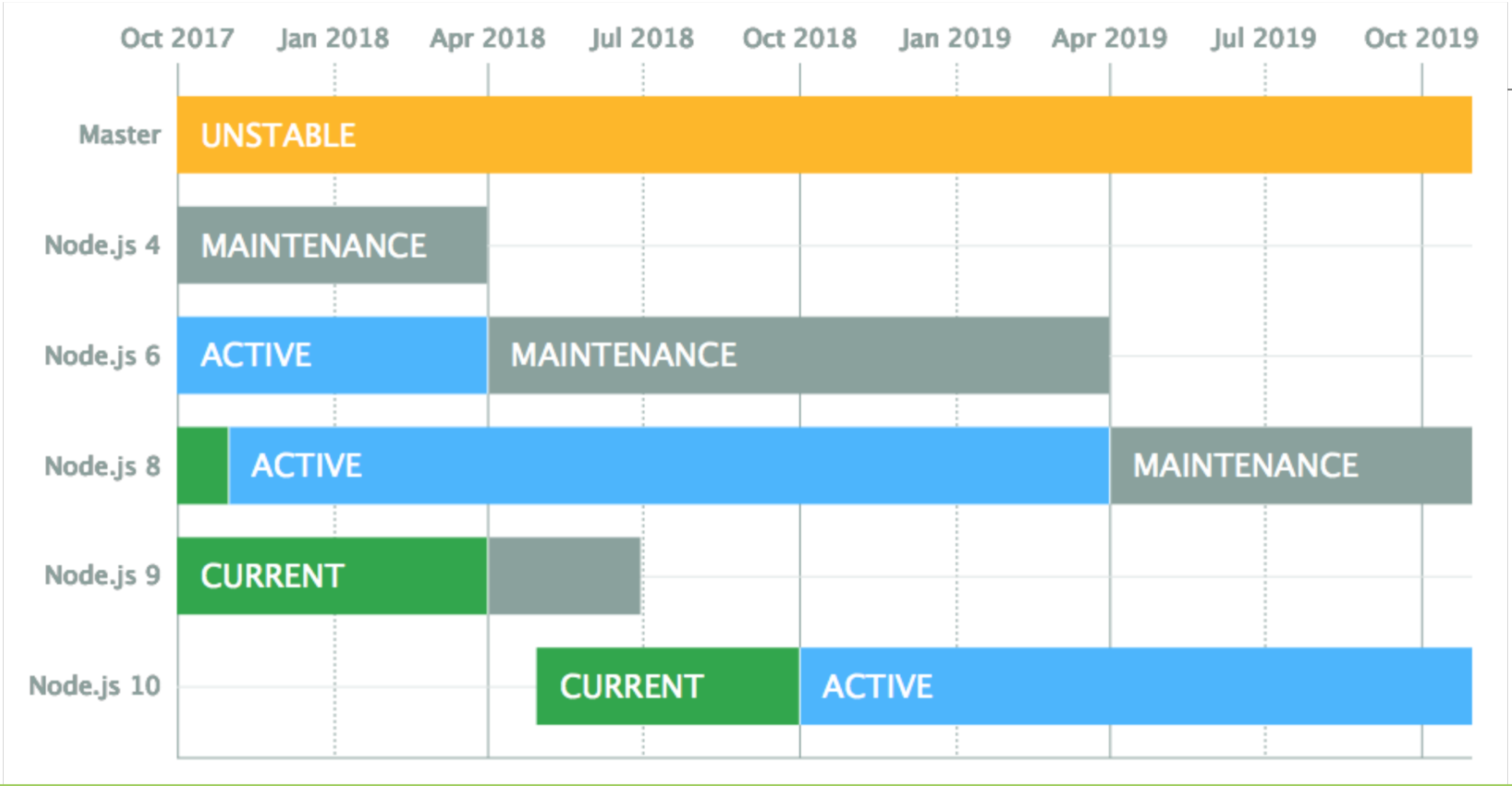
Release schedule

Release	Status	Codename	Initial Release	Active LTS Start	Maintenance LTS Start	End-of-life
6.x	**Maintenance LTS**	Boron	2016-04-26	2016-10-18	2018-04-30	April 2019
8.x	**Active LTS**	Carbon	2017-05-30	2017-10-31	April 2019	December 2019[1]
10.x	**Current Release**	Dubnium	2018-04-24	October 2018	April 2020	April 2021
11.x	**Pending**		2018-10-23			June 2019

Dates are subject to change.

- [1]: The 8.x *Maintenance* LTS cycle is currently scheduled to expire early on December 31, 2019 to align with the scheduled End-of-Life of OpenSSL-1.0.2.

Node.js LTS Release Schedule



Introduction: Basic

In simple words Node.js is ‘**server-side JavaScript**’.

In not-so-simple words Node.js is a **high-performance** network applications framework, well optimized for high concurrent environments.

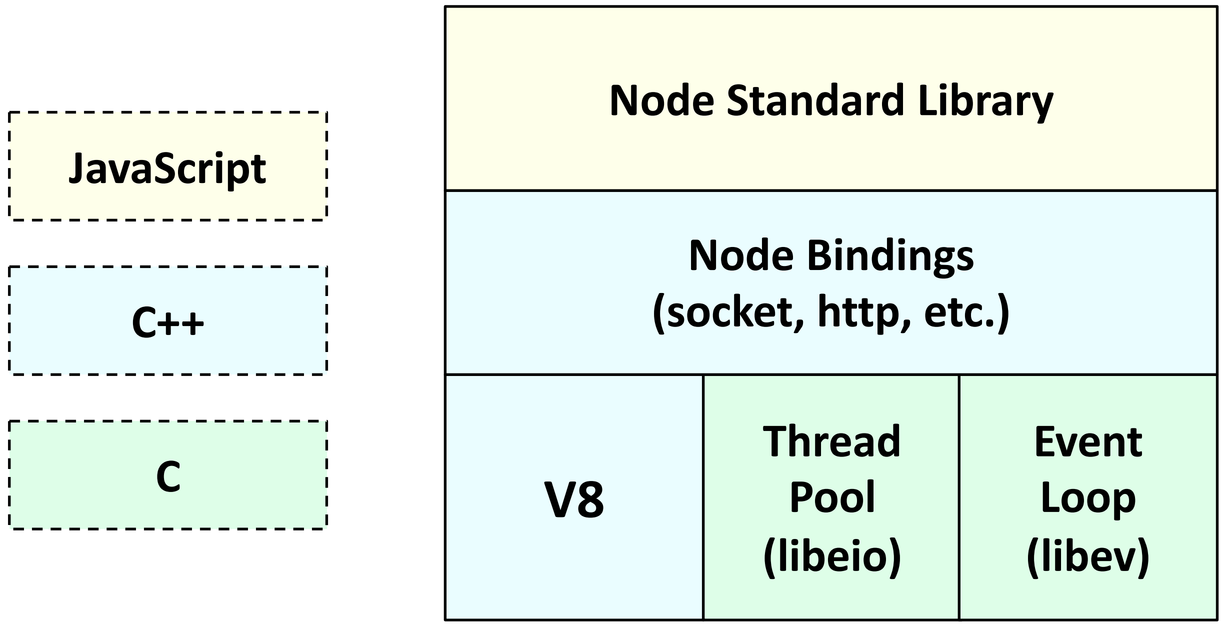
It’s a **command line** tool.

In ‘Node.js’ , ‘.js’ doesn’t mean that its solely written in JavaScript. It is **40%** JS and **60%** C++. (next slide)

From the official site:

‘Node's goal is to provide an easy way to build scalable network programs’ (nodejs.org)

Introduction: Node Architecture



Introduction: Advanced (& Confusing)

As already mentioned, Node.js uses an **event-driven**, **non-blocking I/O** model.

It makes use of **event-loops** via JavaScript's **callback** functionality to implement the **non-blocking I/O**.

Programs for Node.js are written in JavaScript but not in the same JavaScript we are use to. There is **no DOM implementation** provided by Node.js, i.e. you **can not** do this:

```
var element = document.getElementById("elementId");
```

Everything inside Node.js runs in a **single-thread** (which must **never block**!).

If your program needs to wait for something (e.g., a response from some server you contacted), it **must provide a callback function.**

NODE IS DEPLOYED BY BIG BRANDS

Big brands are using Node to power their business

Manufacturing	Financial	eCommerce	Media	Technology
BMW	ADP	amazon.com	beatsMUSIC	salesforce.com
GE	citigroup	BEST BUY	CONDÉ NAST	Apple
GM General Motors	Goldman Sachs	ebay	DOWJONES	box
Johnson Controls	PayPal	TARGET	The New York Times	intel
SIEMENS	WELLS FARGO	Zappos.com	SONY	YAHOO!

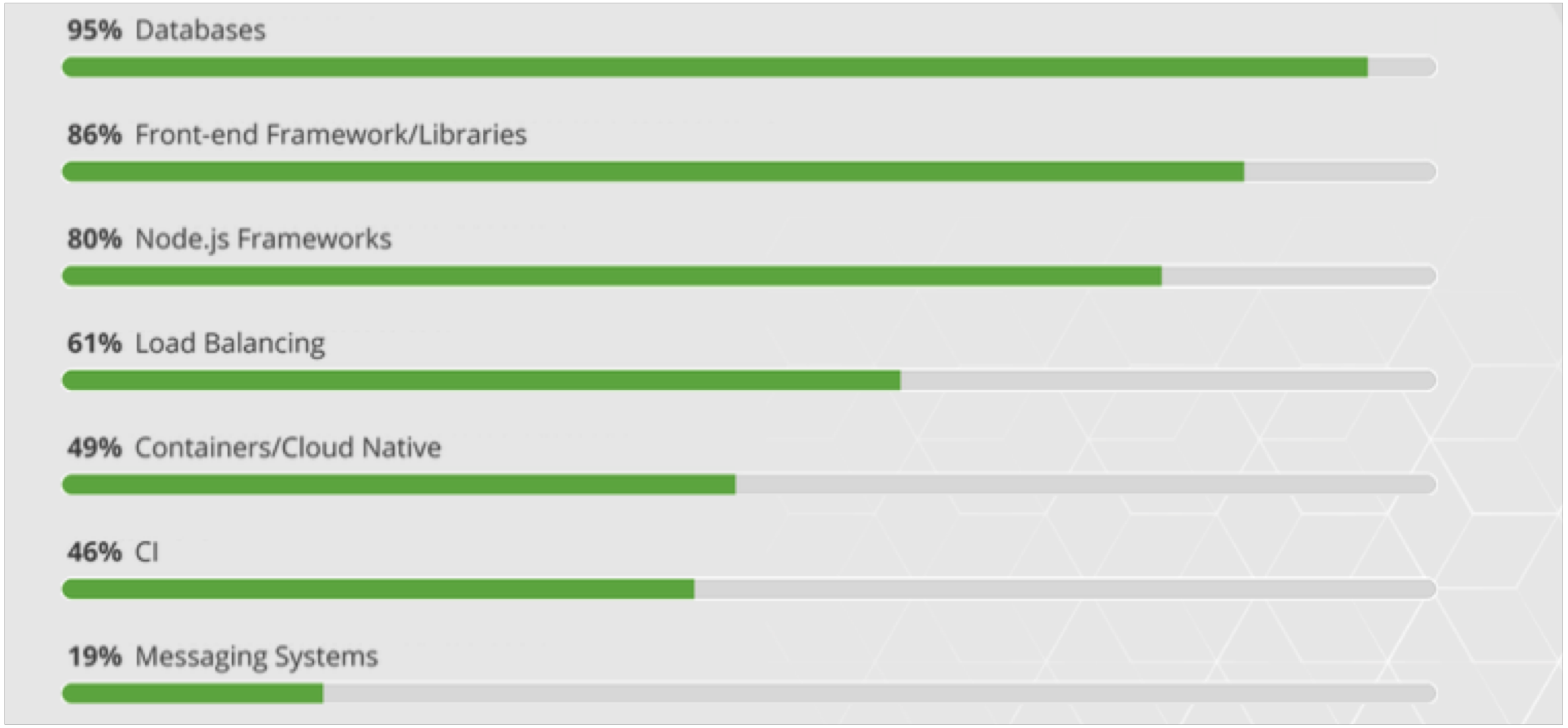
And more recently….

MetLife, MasterCard, airbnb, Department of the Navy – United States of America, nearForm, travelocity, STAPLES, BARNES & NOBLE, PEARSON, npr, Morgan Stanley, HBO, Go Daddy.com, NETFLIX

Local Company

https://hackernoon.com/node-js-emerging-as-the-universal-development-framework-for-a-diversity-of-applications-c2e788290f5f

Tools & Tech used with NodeJS (last 12 months)



When to use Node.js?

Node.js is good for creating streaming based real-time services, web chat applications, static file servers etc.

If you need high level concurrency and not worried about CPU-cycles.

If you are great at writing JavaScript code because then you can use the same language at both the places: *server-side and client-side*.

ASIDE

The async/await feature has completely changed the way we write asynchronous code, actually making it look and behave a little more like synchronous code. Supported by Node.js since v7.6, this feature came as part of the solution to the infamous ”Callback hell”.

More can be found at: http://stackoverflow.com/questions/5062614/how-to-decide-when-to-use-nodejs

When to use Node.js?

USE CASE 1: Real-time Applications

Collaborative apps (Trello, Google Docs), live-chat, instant-messaging, and online gaming are all examples of RTAs that benefit from a Node.js architecture.

These applications function within a time frame that the users sense as immediate and current. Node.js specifications are the solution for the low-latency needed for these programs to work efficiently.

It facilitates handling multiple client requests, enables reusing packages of library code and the data sync between the client and the server happens very fast.

When to use Node.js?

USE CASE 2: Single Page Applications

SPAs are web apps that load a single HTML page and dynamically update that page as the user interacts with the app. Much of the work happens on the client side, in JavaScript.

Even though these are an awesome evolution in web development, they come with some problems when it comes to rendering. This can negatively affect your SEO(Search engine Optimisation) performance for instance.

Server-side rendering in a Node.js environment is a popular option to solve this.

When to use Node.js?

USE CASE 3: Scalability

Node.js won't ever get bigger than you need it to be. The beauty of it is that it's minimalist enough to customize depending on the use case. Performance-wise, that's key.

Even its name emphasizes that it's made to assemble multiple small distributed *nodes* communicating with each other.

Node's modularity allows you to create small apps without having to deal with a bloated, overkill ecosystem. You choose the tools you need for the job and then scale as needed.

This scalability is not free from complications though, and if you're not careful, Node.js can become... dangerous.

https://iwf1.com/apache-vs-nginx-vs-node-js-and-what-it-means-about-the-performance-of-wordpress-vs-ghost

Some Node.js benchmarks

About The Tests

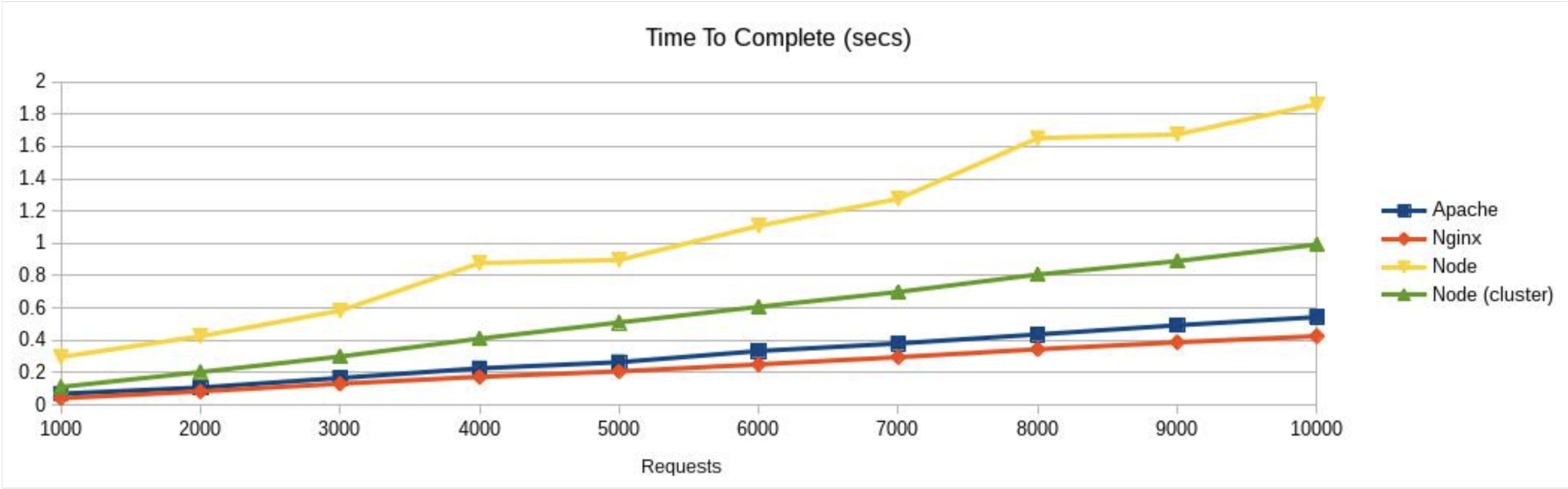
All test were ran locally on an:

- Intel core i7-2600k machine of 4 cores and 8 threads.
- Gentoo Linux is the operating system used to run the tests.

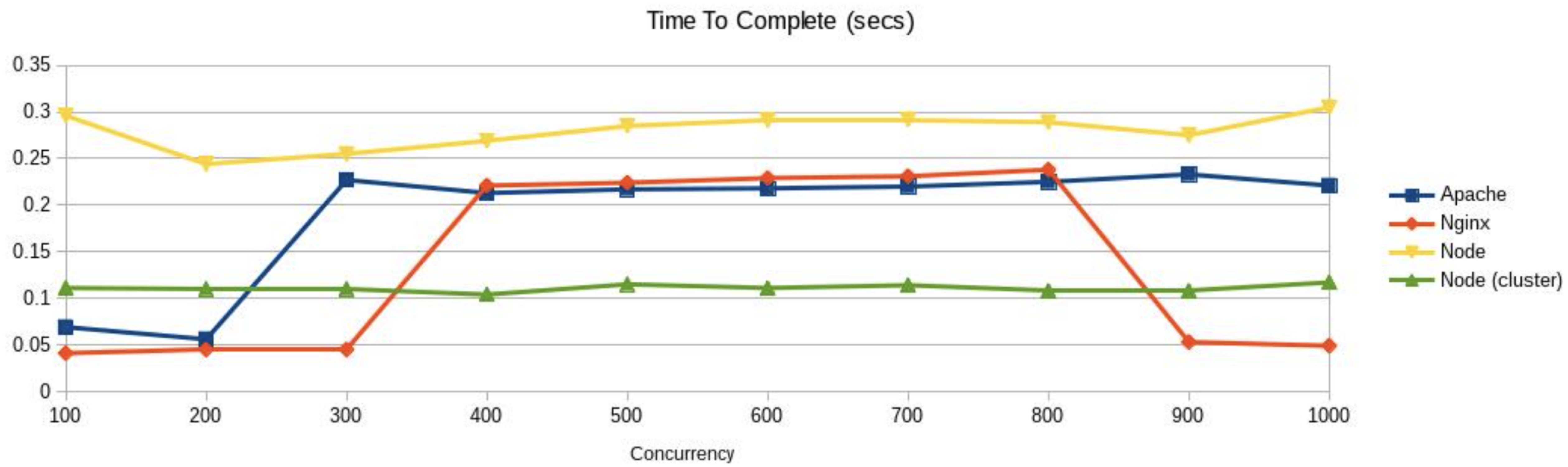
The tool used for benchmarking: ApacheBench, Version 2.3 <$Revision: 1748469 $>.

The tests included a series of benchmarks, starting from 1,000 to 10,000 requests and a concurrency of 100 to 1,000 – the results were quite surprising.

In addition, stress test to measure server function under high load was also issued.



Apache vs Nginx vs Node: performance under requests load (per 100 concurrent users)



Apache vs Nginx vs Node: performance under concurrent users load (per 1,000 requests)

When to not use Node.js

When you are doing heavy and CPU intensive calculations on the server side, because event-loops are CPU hungry.

Node.js is out of beta, but it will keep on changing from one revision to another and there is little backward compatibility at the moment (but it is improving). A lot of the packages are also unstable and constantly changing.

Node.js is a no match for enterprise level application frameworks like Spring (java), Django (python), Symfony (php) etc. Applications written on such platforms are meant to be highly user interactive and involve complex business logic.

Read further on disadvantages of Node.js on Quora: http://www.quora.com/What-are-the-disadvantages-of-using-Node-js

When to not use Node.js

Put bluntly, Node.js allows you to easily shoot yourself in the foot. Configuration & customization come at a price, and if you're inexperienced or undisciplined, you might lose yourself—or your client.

Contrary to a more conventional approach, *you* create the structure that supports your backend. That involves a lot of decision-making, meaning that you have to know what you're doing and where you are going if your project scales.

With other languages like Ruby and its well-known framework Ruby on Rails, for instance, we were used to the paradigm "convention over configuration." These traditional frameworks took developers by the hand and shone some light on the safe path.

With Node.js this goes head over heels. More freedom is given to developers, but the road might get dark and scary if you make the wrong decisions.

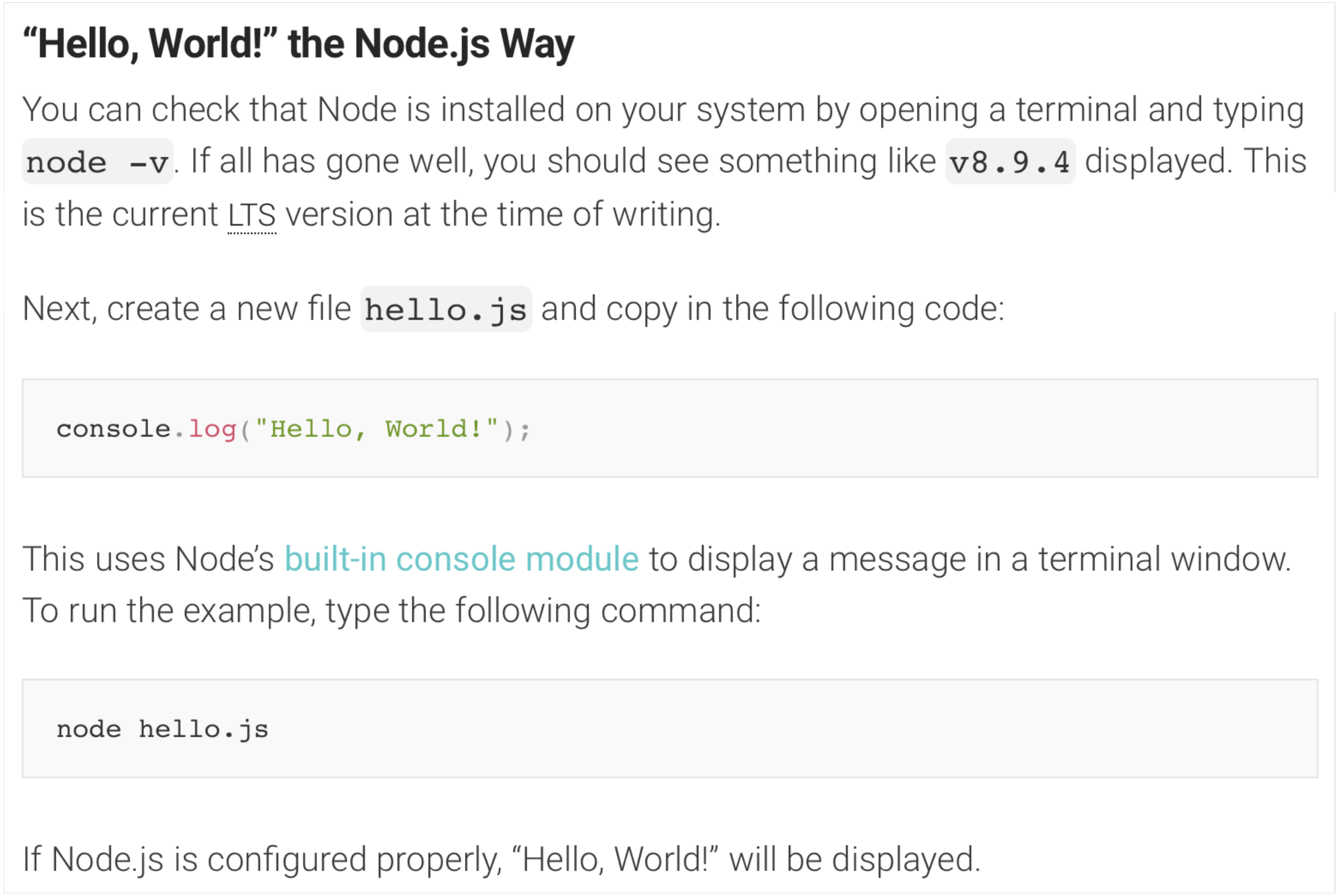
And then you'll find out what "callback hell" really is….

When to not use Node.js

Now this doesn't mean that you can't build bigger server applications with it, but you should always keep these factors in mind.

Even the creator of Node.js, Ryan Dahl, eventually realized the limitations of the system before leaving to work on other projects. He was very transparent about it:

"*[...] I think Node is not the best system to build a massive server web. I would use Go for that. And honestly, that's the reason why I left Node. It was the realization that: oh, actually, this is not the best server-side system ever.*"



“Hello, World!” the Node.js Way

You can check that Node is installed on your system by opening a terminal and typing `node -v`. If all has gone well, you should see something like `v8.9.4` displayed. This is the current LTS version at the time of writing.

Next, create a new file `hello.js` and copy in the following code:

```
console.log("Hello, World!");
```

This uses Node’s built-in console module to display a message in a terminal window. To run the example, type the following command:

```
node hello.js
```

If Node.js is configured properly, “Hello, World!” will be displayed.

Nodes Execution Model

NODES EVENT-DRIVEN, NON-BLOCKING I/O MODEL

Overview of Blocking vs Non-Blocking

Blocking is when the execution of additional JavaScript in the Node.js process must wait until a non-JavaScript operation (such as I/O) completes. This happens because the event loop is unable to continue running JavaScript while a blocking operation is occurring.

In Node.js, JavaScript that exhibits poor performance due to being CPU intensive, rather than waiting on a non-JavaScript operation, isn't typically referred to as blocking. Synchronous methods in the Node.js standard library that use *libuv* are the most commonly used blocking operations. Native modules may also have blocking methods.

All of the I/O methods in the Node.js standard library provide asynchronous versions, which are non-blocking, and accept callback functions. Some methods also have blocking counterparts, which have names that end with *Sync*.

Overview of Blocking vs Non-Blocking

Blocking methods execute **synchronously** and **non-blocking** methods execute **asynchronously.**

Using the File System module as an example, this is a **synchronous** file read:

```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
```

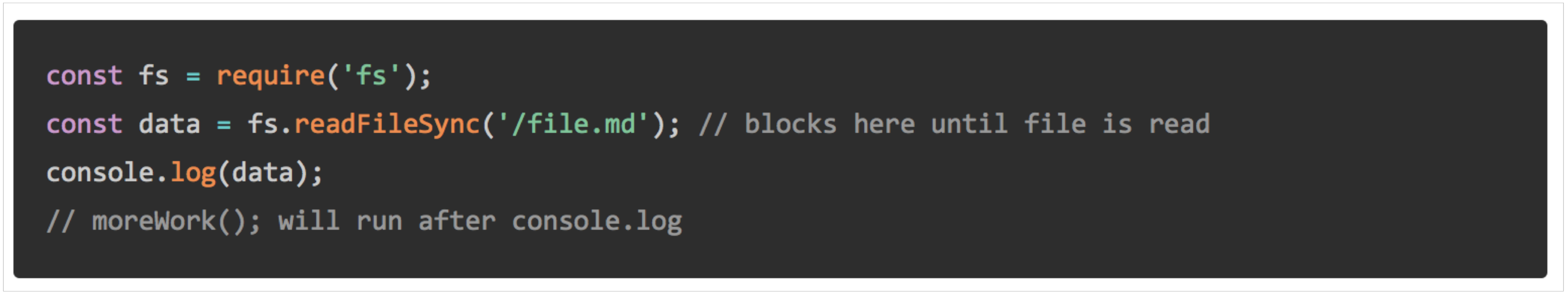
And here is an equivalent **asynchronous** example:

```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
});
```

Overview of Blocking vs Non-Blocking

The first example appears simpler than the second but has the disadvantage of the second line **blocking** the execution of any additional JavaScript until the entire file is read. Note that in the **synchronous** version if an error is thrown it will need to be caught or the process will crash. In the **asynchronous** version, it is up to the author to decide whether an error should throw as shown.

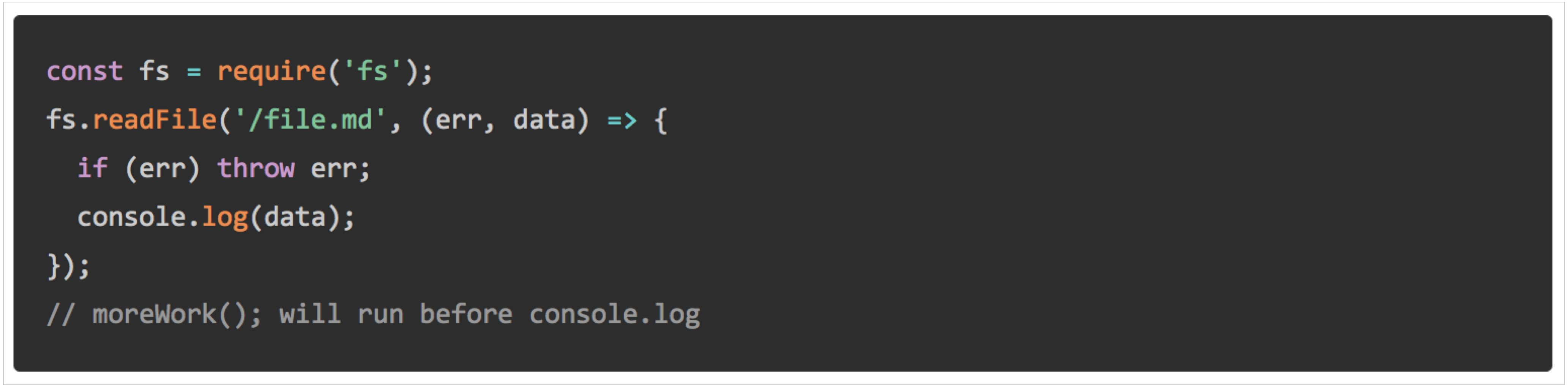
Let's expand our example a little bit:



```
const fs = require('fs');
const data = fs.readFileSync('/file.md'); // blocks here until file is read
console.log(data);
// moreWork(); will run after console.log
```

Overview of Blocking vs Non-Blocking

And here is a similar, but not equivalent asynchronous example:



```
const fs = require('fs');
fs.readFile('/file.md', (err, data) => {
  if (err) throw err;
  console.log(data);
});
// moreWork(); will run before console.log
```

In the first example, console.log will be called before moreWork(). In the second example fs.readFile() is non-blocking so JavaScript execution can continue and moreWork() will be called first.

The ability to run moreWork() without waiting for the file read to complete is a key design choice that allows for higher throughput.

Blocking | I/O Model

Example: ways in which a server can process orders from customers

Hi, my name is Apache. How may I take your order?

- The server serves one customer at a time.
- As each customer is deciding on their order, the server **sits and waits**.
- When the customer decides on an order, the server processes their order and moves on to the next customer.

Blocking | I/O Model

Blocking | I/O Model

Pseudocode:

```
order1 = db.query(“SELECT * FROM
menu WHERE preference = most”)

order1.process

order2.process
```

Blocking | I/O Model

The more customers you want to serve at once, the more cashier lines you'll need.

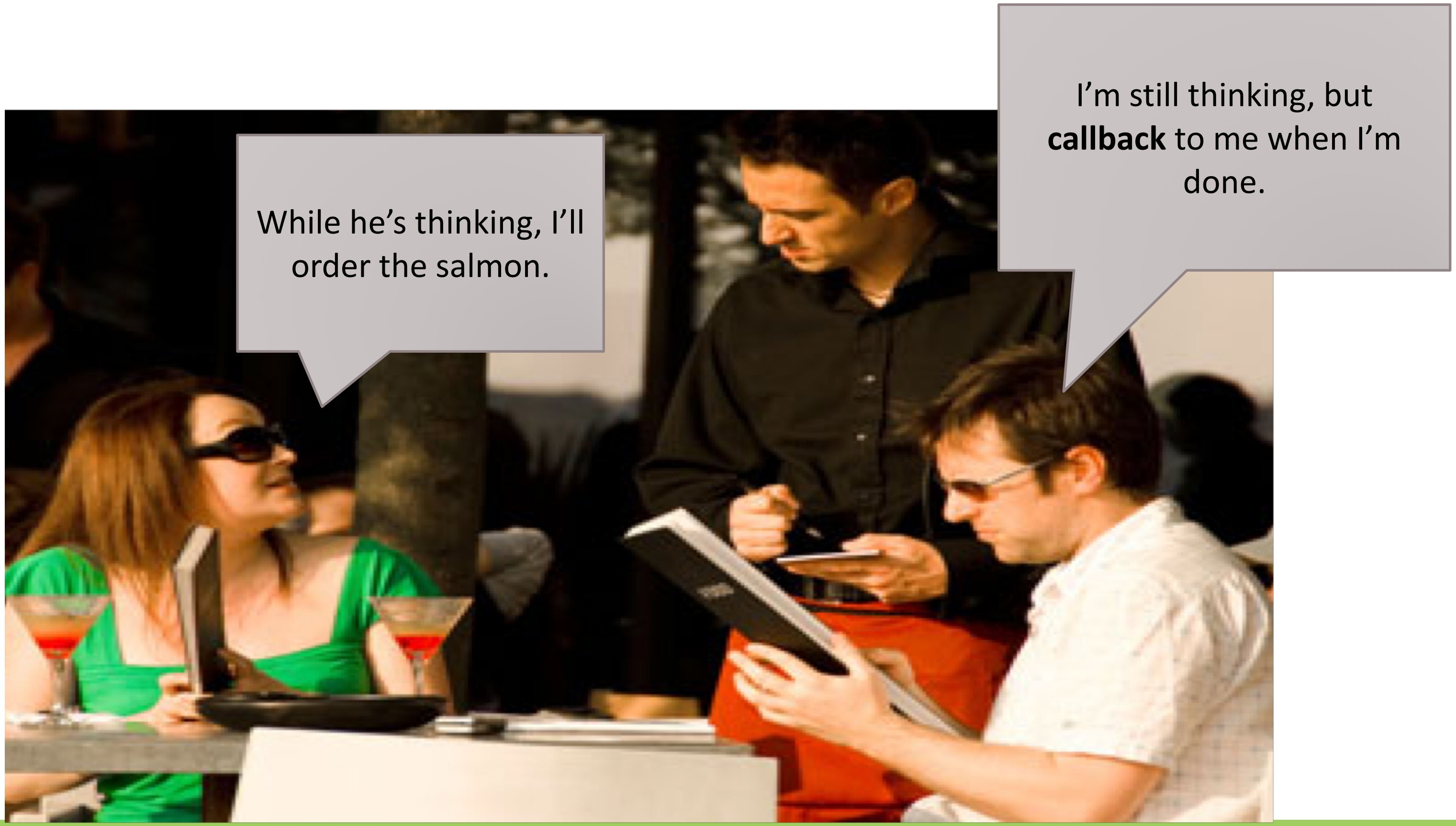
Cashier lines ~ threads in computing

Multi-threaded processing

Parallel code execution

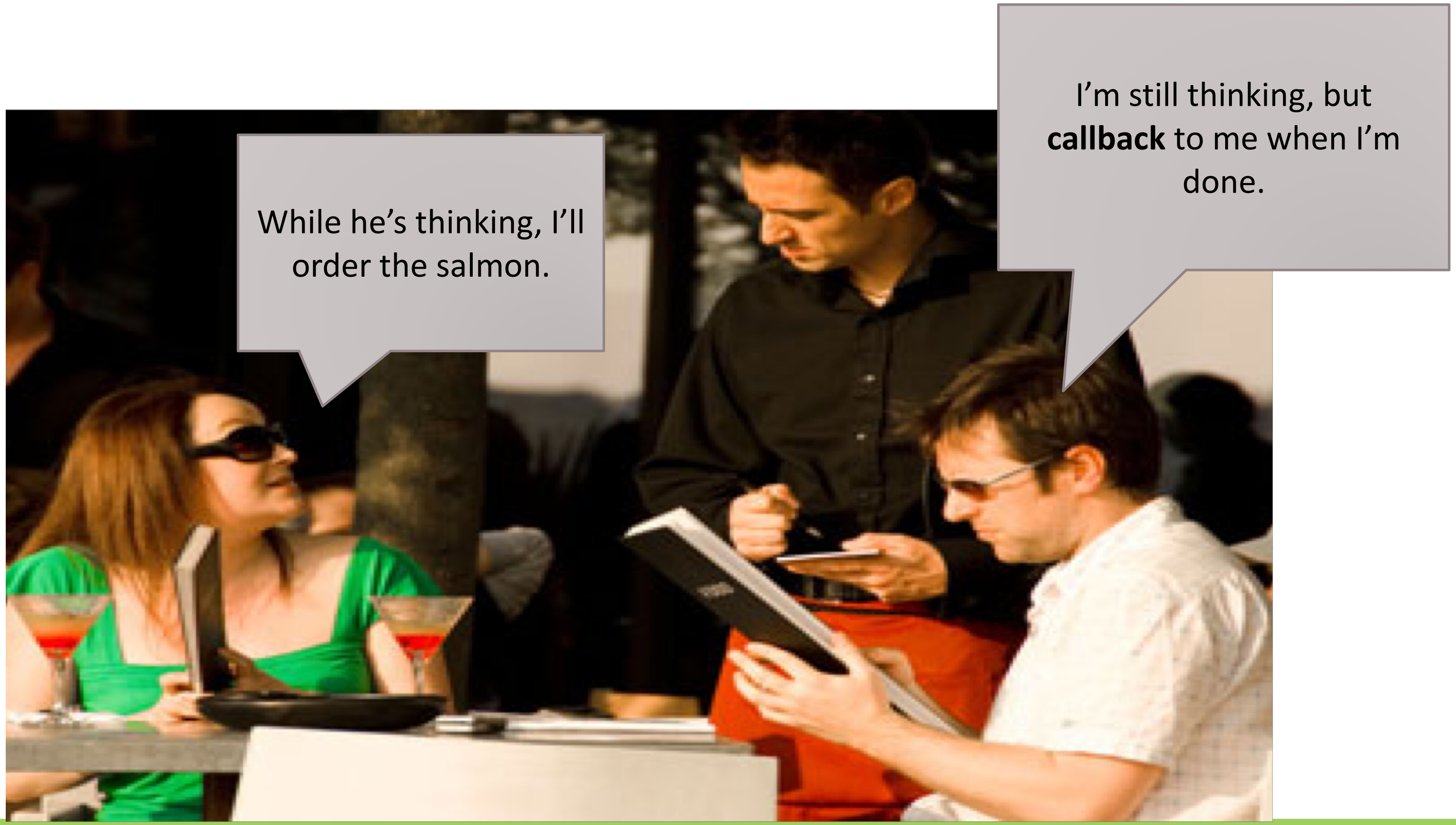
Multiple CPUs run at a time, utilizing shared resources (memory)

Non-Blocking | I/O Model


While he's thinking, I'll order the salmon.
I'm still thinking, but **callback** to me when I'm done.

- Node loops through the customers and polls them to determine which ones are ready to order.
- During a function's queue, Node can listen to another event.
- When the other customer is finally ready to order, he'll issue a *callback*.
- Asynchronous **callbacks**: *"come back to me when I'm finished"*
  - function called at the completion of a given task.

Non-Blocking | I/O Model

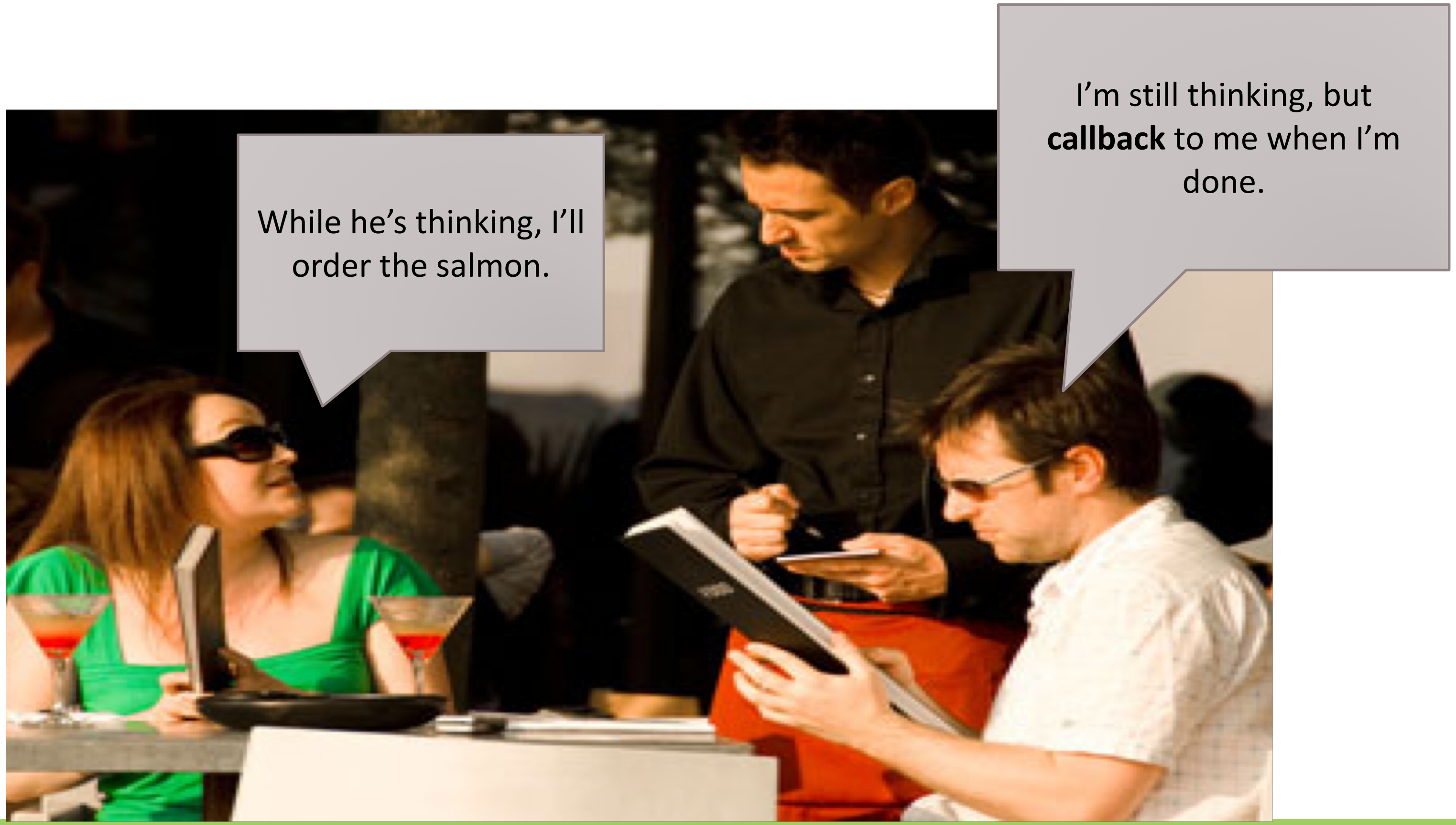

While he's thinking, I'll order the salmon.
I'm still thinking, but **callback** to me when I'm done.

Node code

```
console.log('Hello');

setTimeout(function() {
  console.log('World');
         },
  5000);

console.log('Bye');

// Outputs:
// Hello
// Bye
// World
```

Allows for high concurrency

Non-Blocking | I/O Model


While he's thinking, I'll order the salmon.
I'm still thinking, but **callback** to me when I'm done.

Single-threaded

No parallel code execution

Single CPU

The Node.js Execution Model

In very simplistic terms, when you connect to a traditional server, such as Apache, it will spawn a new thread to handle the request. In a language such as PHP or Ruby, any subsequent I/O operations (for example, interacting with a database) block the execution of your code until the operation has completed.

That is, the server has to wait for the database lookup to complete before it can move on to processing the result.

If new requests come in while this is happening, the server will spawn new threads to deal with them. This is potentially inefficient, as a large number of threads can cause a system to become sluggish — and, in the worse case, for the site to go down.

The most common way to support more connections is to add more servers.

The Node.js Execution Model

Node.js, however, is **single-threaded**. It is also **event-driven**, which means that everything that happens in Node is in reaction to an **event**. For example, when a new request comes in (one kind of event) the server will start processing it.

If it then encounters a blocking I/O operation, instead of **waiting** for this to complete, it will register a **callback** before continuing to process the next event.

When the I/O operation has finished (another kind of event), the server will execute the **callback** and continue working on the original request.

Under the hood, Node uses the libuv library to implement this asynchronous (i.e. non-blocking) behaviour.

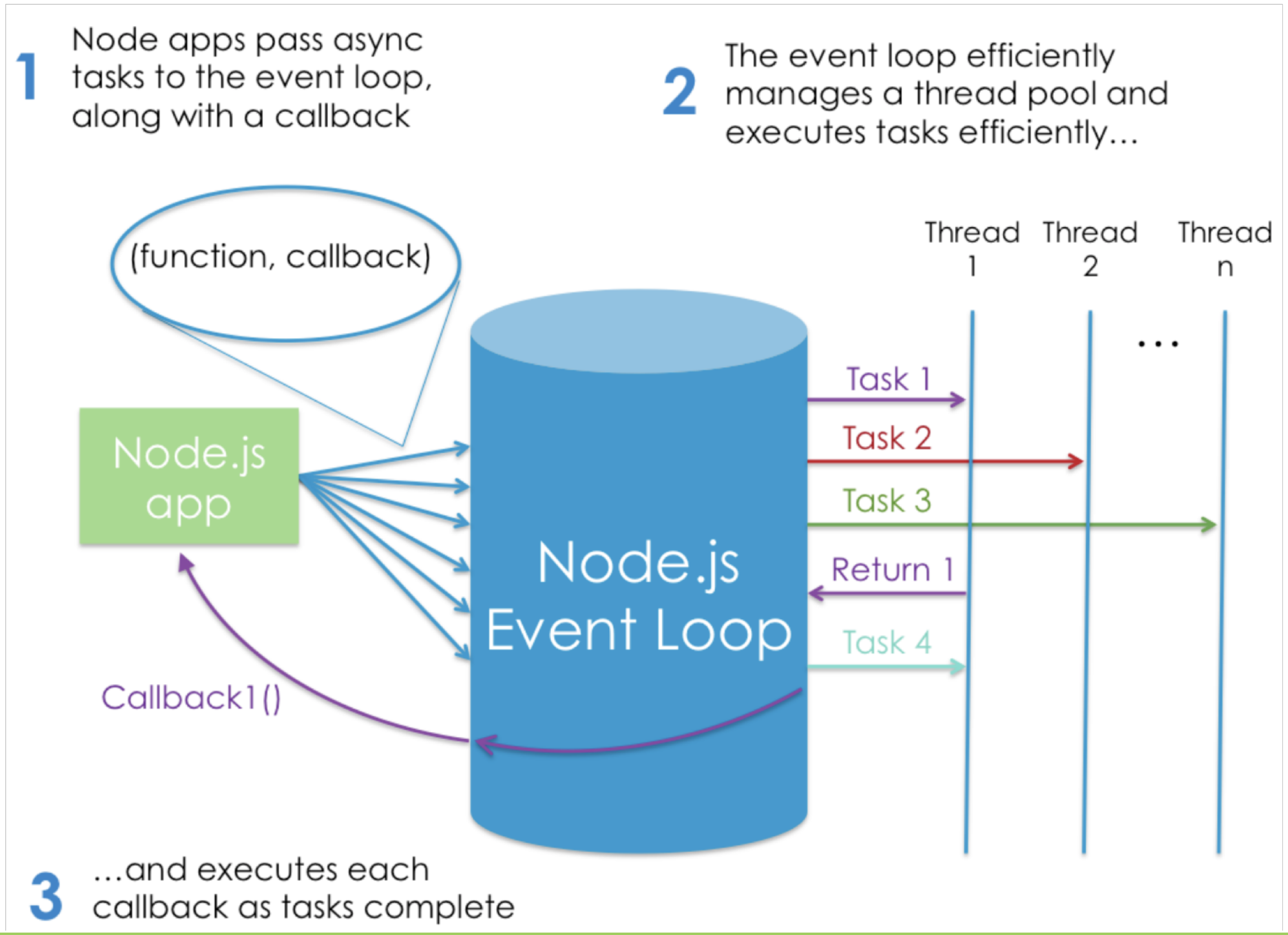
The Node.js Execution Model

Node's execution model causes the server very little overhead, and consequently it's capable of handling a large number of simultaneous connections.

The traditional approach to scaling a Node app is to clone it and have the cloned instances share the workload. Node.js even has a built-in module to help you implement a cloning strategy on a single server.

The following image depicts Node's execution model:

The Node.js Execution Model


1
Node apps pass async tasks to the event loop, along with a callback
2
The event loop efficiently manages a thread pool and executes tasks efficiently…
(function, callback)
Node.js app
Node.js Event Loop
Thread 1
Thread 2
Thread n
…
Task 1
Task 2
Task 3
Return 1
Task 4
Callback1()
3
…and executes each callback as tasks complete

Node.js Event-loop

Warning! Be careful to keep CPU intensive operations off the event loop.

Are there any Downsides?

The fact that Node runs in a single thread does impose some limitations.

For example, blocking I/O calls should be avoided, and errors should always be handled correctly for fear of crashing the entire process.

Some developers also dislike the **callback-based** style of coding that JavaScript imposes (so much so that there's even a site dedicated to the horrors of writing asynchronous JavaScript).

But with the arrival of native Promises, followed closely by async await (which is enabled by default as of Node version 7.6), this is rapidly becoming a thing of the past. (we'll cover these topics soon)

Asynchrony in Node

EVENTS, CALLBACKS, PROMISES & ASYNC/AWAITS

https://medium.com/codebuddies/getting-to-know-asynchronous-javascript-callbacks-promises-and-async-await-17e0673281ee

Background

According to Wikipedia:

"Asynchrony in computer programming refers to the occurrence of events independently of the main program flow and ways to deal with such events."

In programming languages like Java or C# for instance the "main program flow" happens on the main thread or process and "the occurrence of events independently of the main program flow" is the spawning of new threads or processes that runs code in parallel to the "main program flow".

(Hence, multi-threaded programming)

Background

This is not the case with JavaScript. That is because a JavaScript program is single threaded and all code is executed in a sequence, not in parallel.

In JavaScript this is handled by using what is called an *“asynchronous non-blocking I/O model”*. What that means is that while the execution of JavaScript is blocking, I/O operations are not.

I/O operations can be fetching data over the internet with Ajax or over WebSocket connections, querying data from a database such as MongoDB or accessing the filesystem with the Node.js “fs” module.

All these kind of operations are done in parallel to the execution of your code and it is not JavaScript that does these operations; to put it simply, the underlying (V8) engine does it (and remember, Node.js is built on top of V8).

Callbacks

In a *synchronous* program, you would write something along the lines of:

```
function processData () {
     var data = fetchData ();
     data += 1;
  return data;
}
```

This works just fine and is very typical in other development environments.

However, if *fetchData()* takes a long time to load the data, then this causes the whole program to 'block' – or just sitting still and waiting - until it loads the data.

Node.js, being an ***asynchronous*** platform, doesn't wait around for things like file I/O to finish - Node.js uses callbacks.

https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks

Callback Functions

So, for JavaScript to know when an asynchronous operation has a result (a result being either returned data or an error that occurred during the operation), it points to a **function** that will be executed **once that result is ready**.

This function is what we call a "**callback function**". Meanwhile, JavaScript continues its normal execution of code.

Here is an example of fetching data from an URL using a module called "**request**":

Callbacks

```
const request = require('request');
 request('https://www.somepage.com', function(error, response, body) {

      if(error){
         // Handle error.
      }
      else {
         // Successful, do something with the result.
      }

 });
```

Callbacks : arrow notation

```
const request = require(‘request’);
 request('https://www.somepage.com', (error, response, body) => {

      if(error){
         // Handle error.
      }
      else {
         // Successful, do something with the result.
      }

 });
```

Callbacks

As you can see, “request” takes an *anonymous* function (the **callback**) as its last argument.

This function is not executed together with the code above. It is saved to be executed later once the underlying I/O operation of fetching data over HTTP(s) is done.

The underlying HTTP(s) request is an **asynchronous** operation and does not block the execution of the rest of the JavaScript code.

The **callback function** is put on a sort of queue called the “**event loop**” until it will be executed with a result from the request.

Callback Hell

Callbacks are a good way to declare what will happen once an I/O operation has a result, but what if you want to use that data in order to make another request?

You can only handle the result of the request (if we use the previous example) *within* the callback function provided.

In this example the variable “result” will not have a value when printed to the console at the last line:

Callback Hell

```
const request = require(‘request’);
let result;
 request('https://www.somepage.com', (error, response, body) => {

     if(error){
        // Handle error.
     }
     else {
        // Successful, do something with the result.
        result = body;
     }

 });
console.log(result);
```

Callback Hell

The last line will output "undefined" to the console because at the time that line is being executed, the callback has not been called and completed.

So if we want to do a second request based on the result of the first one, we have to do it inside the callback function of the first request because that is where the result will be available:

Callback Hell

```
request('http://www.somepage.com’, (error, response, body) => {

    if(error){
         // Handle error.
         }
     else {
         request(`http://www.somepage.com/${body.someValue}`, (nxtError, nxtResponse, nxtBody) => {
             if(nxtError){
                 // Handle error.
             }
             else {
                 // Use nxtBody for something
             }
         });
     }

});
```

Callback Hell

When you have a callback in a callback like this, the code tends to be a bit less readable and a bit messy. *In some cases you may have a callback in a callback in a callback or even a callback in a callback in a callback in a callback*. You get the point: it gets messy – very messy.

One thing to note here is the first argument in every callback function will contain an **error** if something went wrong, or will be empty if all went well.

This pattern is called "**error first callbacks**" and is very common. It is the standard pattern for callback-based APIs in Node.js. This means that for every callback declared we need to check if there is an error and that just adds to the mess when dealing with nested callbacks.

This is the anti-pattern that has been named "callback hell".

Defining an Error-First Callback

There's really only two rules for defining an error-first callback:

- The first argument of the callback is reserved for an error object. If an error occurred, it will be returned by the first err argument.
- The second argument of the callback is reserved for any successful response data. If no error occurred, err will be set to null and any successful data will be returned in the second argument.

```
fs.readFile('/foo.txt', (error, data) => {

        // Need to handle possible error.
        console.log(data);
      }
});
```

Defining an Error-First Callback

fs.readFile() takes in a file path to read from, and calls your callback once it has finished.

If all goes well, the file contents are returned in the data argument.

But if something goes wrong (the file doesn't exist, permission is denied, etc) the first error argument will be populated with an error object containing information about the problem.

Its up to you, the callback creator, to properly handle this error. You can throw an error if you want your entire application to shutdown. Or if you're in the middle of some asynchronous flow you can propagate that error out to the next callback. The choice depends on both the situation and the desired behavior.

Defining an Error-First Callback

```
fs.readFile('/foo.txt’, function(error, data) {

     if(error) {
      // handle possible error here.
     return;
             }
     // Do whatever with data.

     console.log(data);
     }

});
```

Or…

```
fs.readFile('/foo.txt', (error, data) => {

     if(error) {
      // handle possible error here.
     return;
             }
     // Do whatever with data.

     console.log(data);
     }

});
```

Callbacks Vs Promises

A promise is an object that wraps an asynchronous operation and notifies when it’s done. This sounds exactly like callbacks, but the important differences are in the usage of Promises.

Instead of providing a callback, a promise has its own methods which you call to tell the promise what will happen when it is successful or when it fails.

The methods a promise provides are “then(…)” for when a successful result is available and “catch(…)” for when something went wrong.

There are lots of frameworks for creating and dealing with promises in JavaScript, but native JavaScript promises were introduced in ECMAScript 2015 (ES6).

Understanding Promises

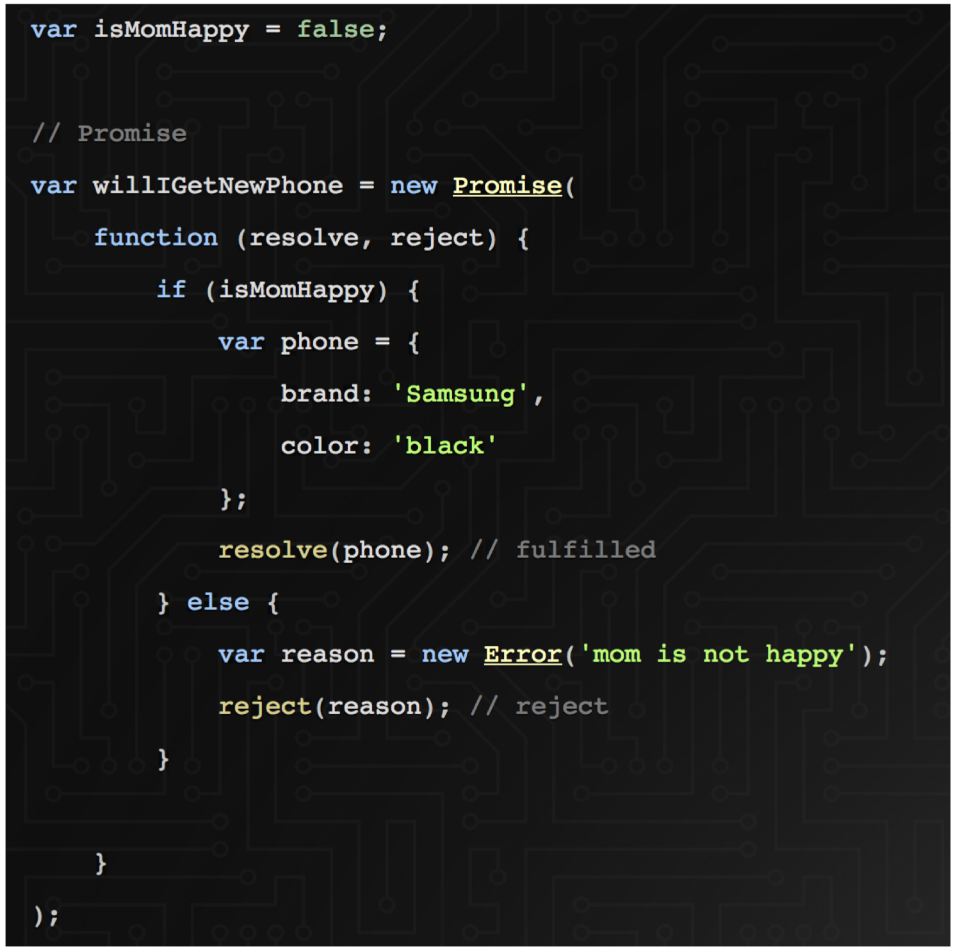
"Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."

You *don't know* if you will get that phone until next week. Your mom can either *really buy* you a brand new phone, or *stand you up* and withhold the phone if she is not happy (Bummer!) – That's a **promise**.

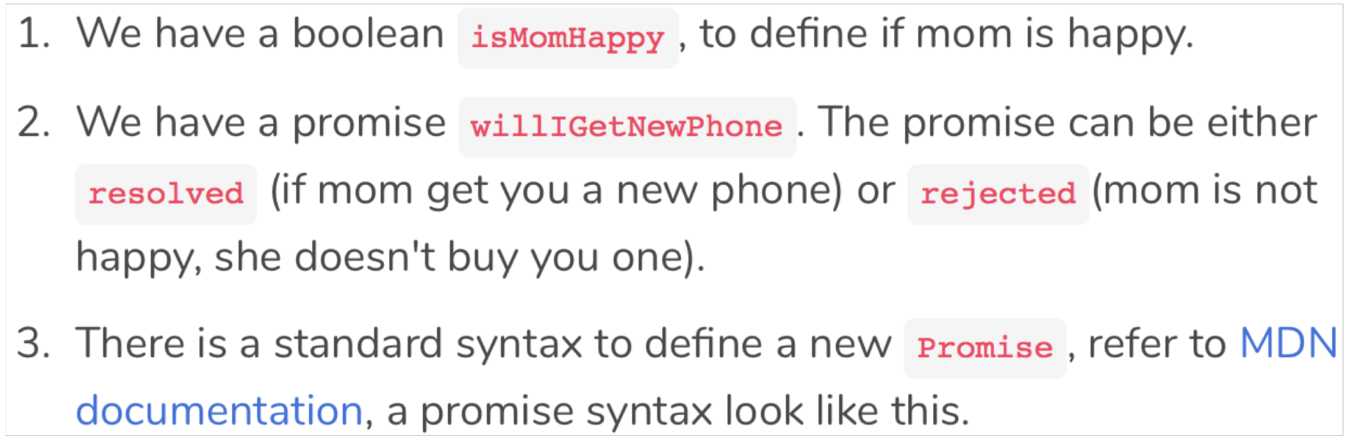
A promise has 3 states. They are:

1. Promise is **pending**: You don't know if you will get that phone until next week.
2. Promise is **resolved**: Your mom really buys you a brand new phone.
3. Promise is **rejected**: You don't get a new phone because your mom is not happy.

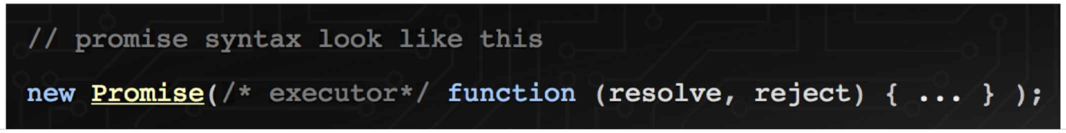
Creating a Promise



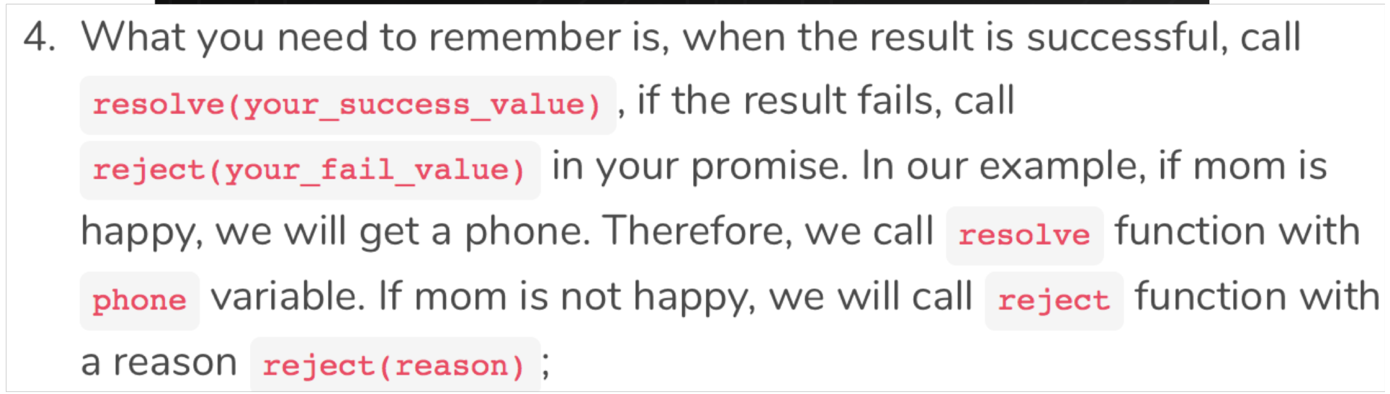
```
var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```



1. We have a boolean `isMomHappy`, to define if mom is happy.
2. We have a promise `willIGetNewPhone`. The promise can be either `resolved` (if mom get you a new phone) or `rejected`(mom is not happy, she doesn't buy you one).
3. There is a standard syntax to define a new `Promise`, refer to MDN documentation, a promise syntax look like this.

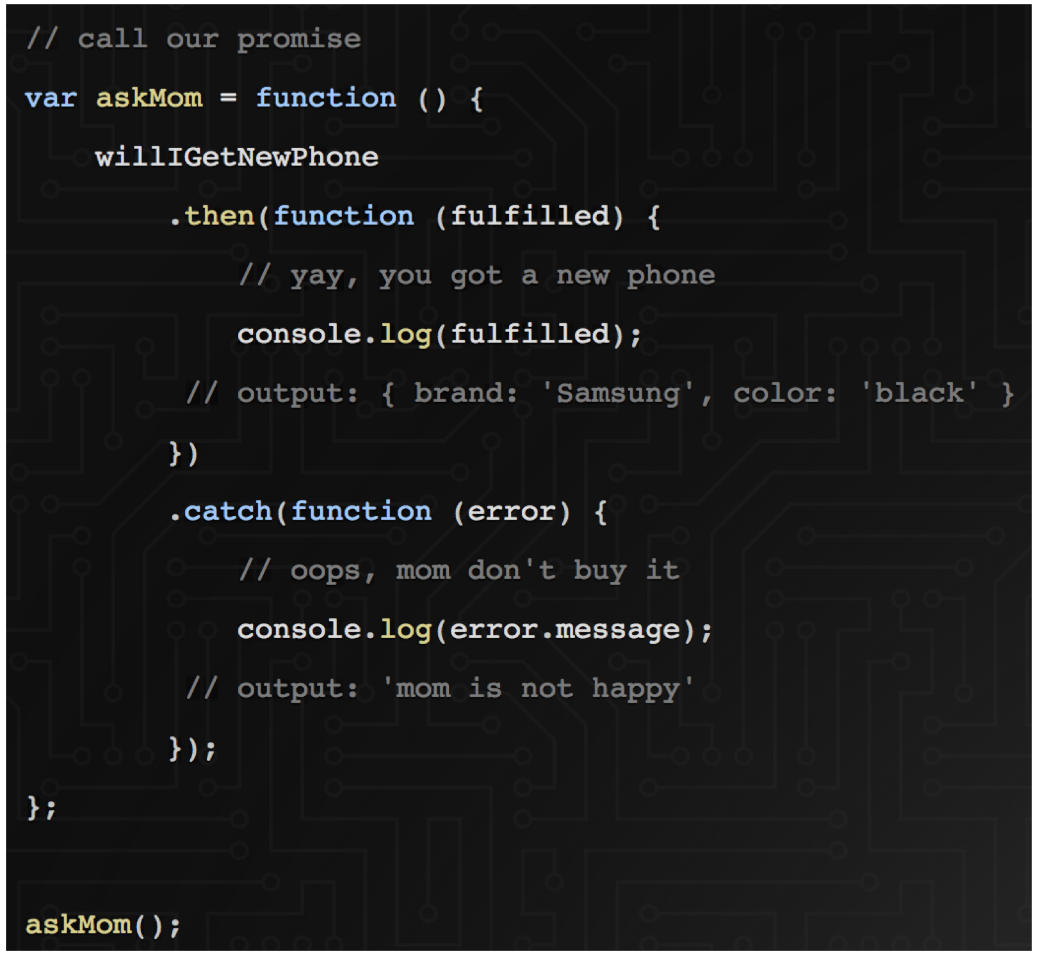


```
// promise syntax look like this
new Promise(/* executor*/ function (resolve, reject) { ... } );
```



4. What you need to remember is, when the result is successful, call `resolve(your_success_value)`, if the result fails, call `reject(your_fail_value)` in your promise. In our example, if mom is happy, we will get a phone. Therefore, we call `resolve` function with `phone` variable. If mom is not happy, we will call `reject` function with a reason `reject(reason)`;

Consuming Promises

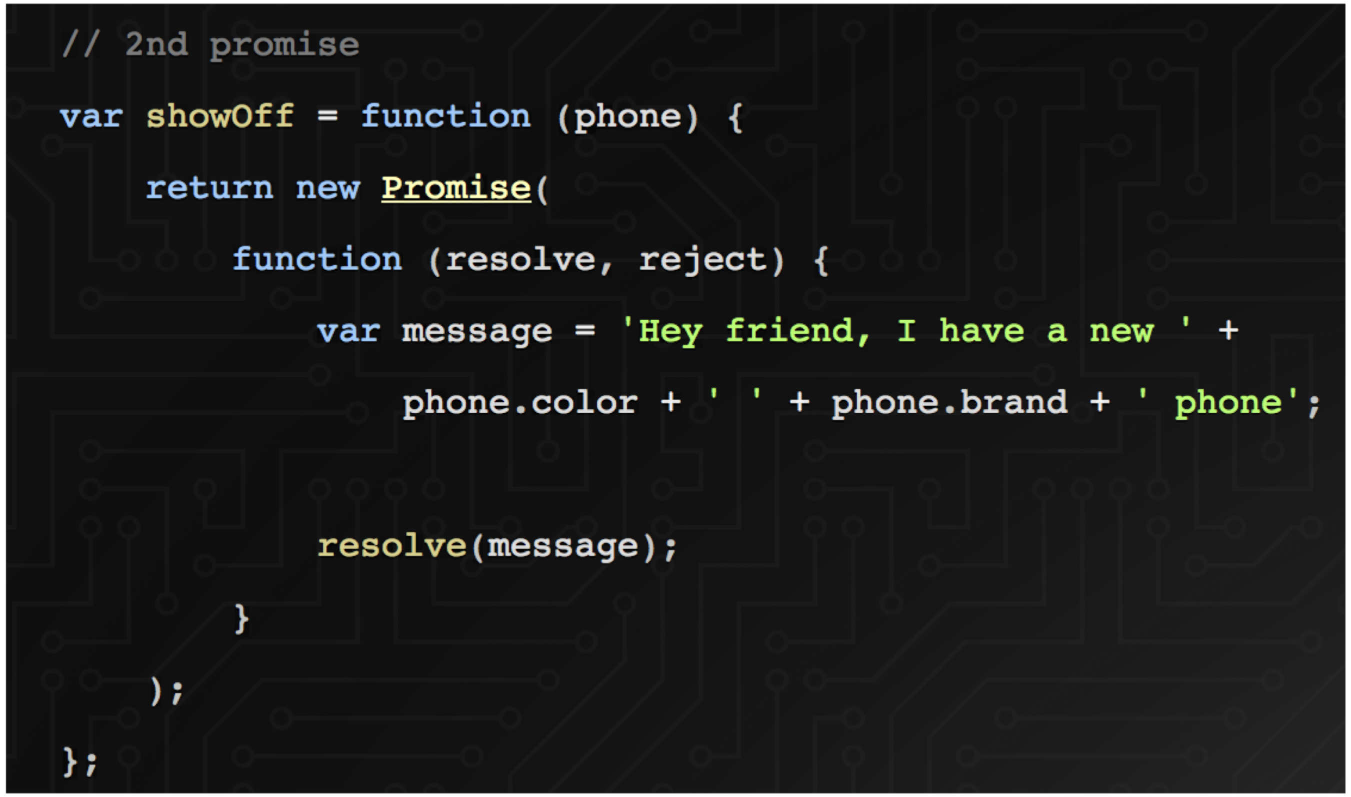


```
// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
         // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};

askMom();
```

1. We have a function call `askMom`. In this function, we will consume our promise `willIGetNewPhone`.
2. We want to take some action once the promise is resolved or rejected, we use `.then` and `.catch` to handle our action.
3. In our example, we have `function(fulfilled) { ... }` in `.then`. What is the value of `fulfilled`? The `fulfilled` value is exactly the value you pass in your promise `resolve(your_success_value)`. Therefore, it will be `phone` in our case.
4. We have `function(error){ ... }` in `.catch`. What is the value of `error`? As you can guess, the `error` value is exactly the value you pass in your promise `reject(your_fail_value)`. Therefore, it will be `reason` in our case.

Chaining Promises

Let's say, you, the kid, promise your friend that you will show them the new phone when your mom buys you one – That’s another promise.

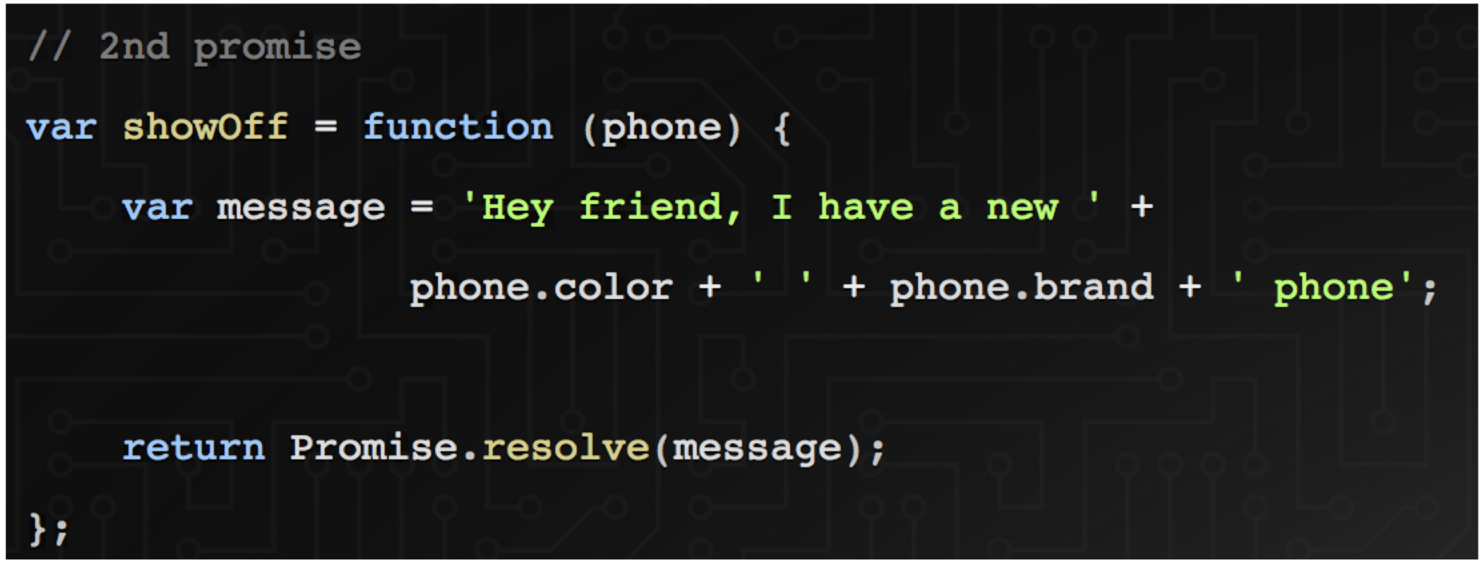


```
// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

Chaining Promises

In the previous example, you might notice we didn't call reject. It's optional.

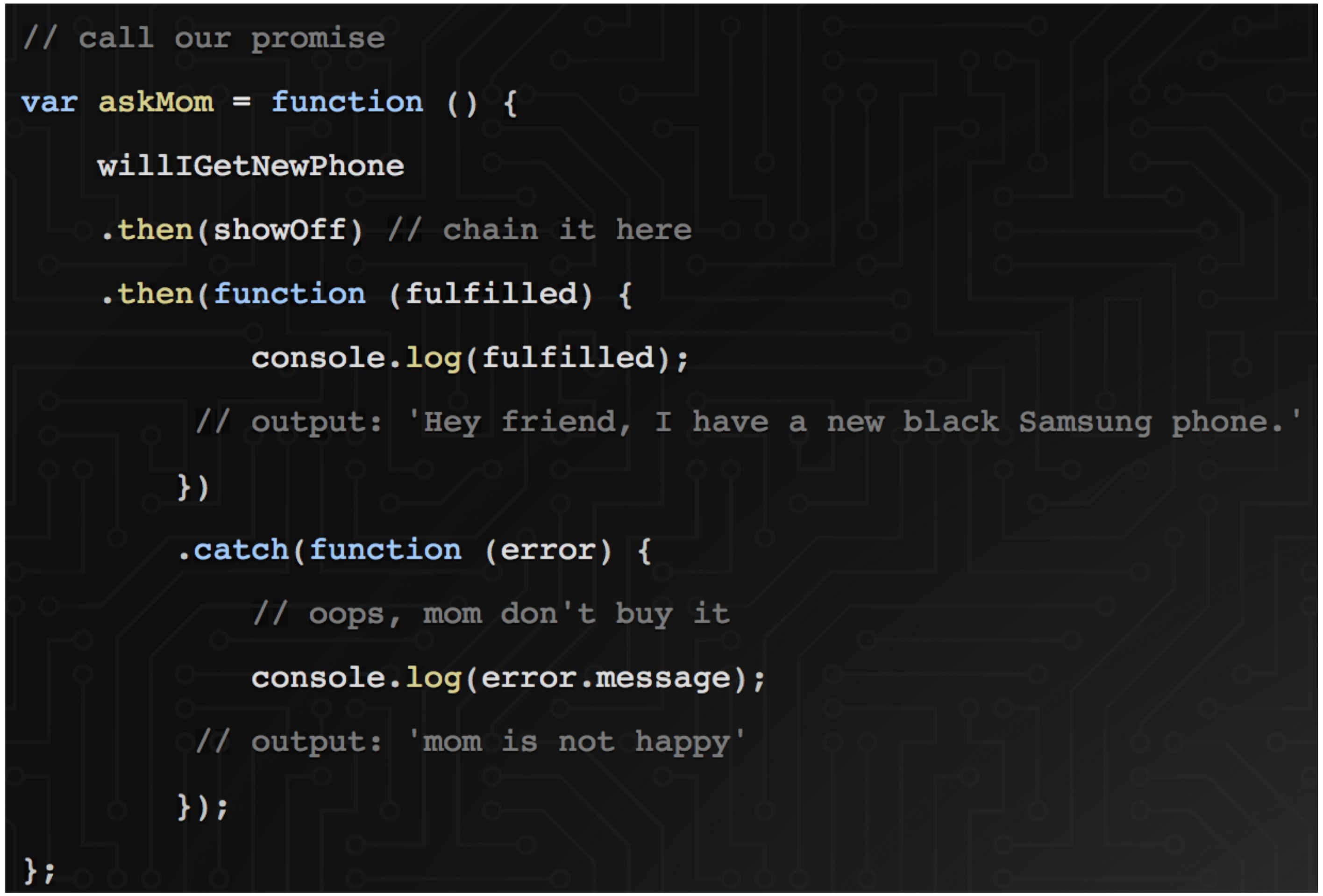
We can shorten this example by using Promise.resolve instead.



```
// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

Chaining Promises

Let's **chain** the promises. You, the kid can only start the showOff promise after the willIGetNewPhone promise.



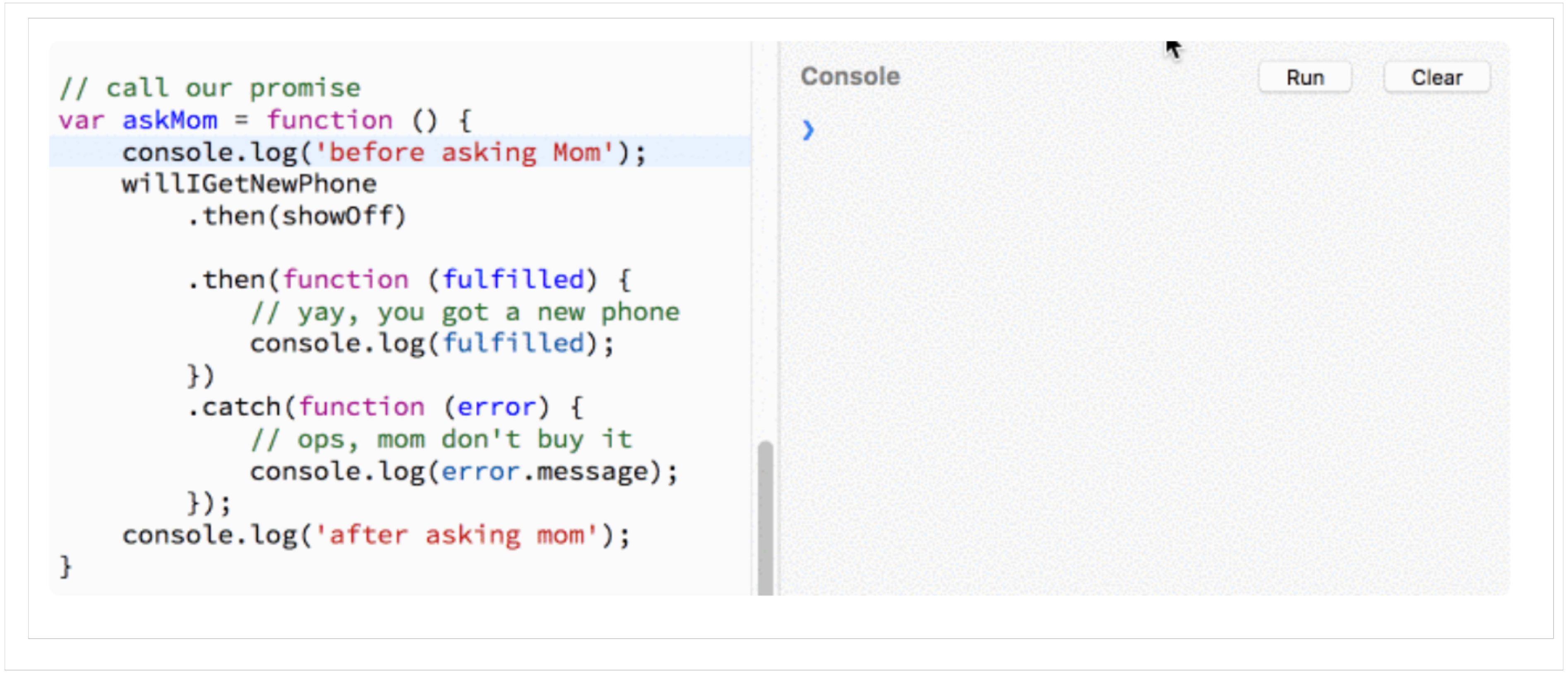
```
// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
         // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};
```

Chaining Promises

Promises are asynchronous - Why? Because life (or JS) waits for no man.

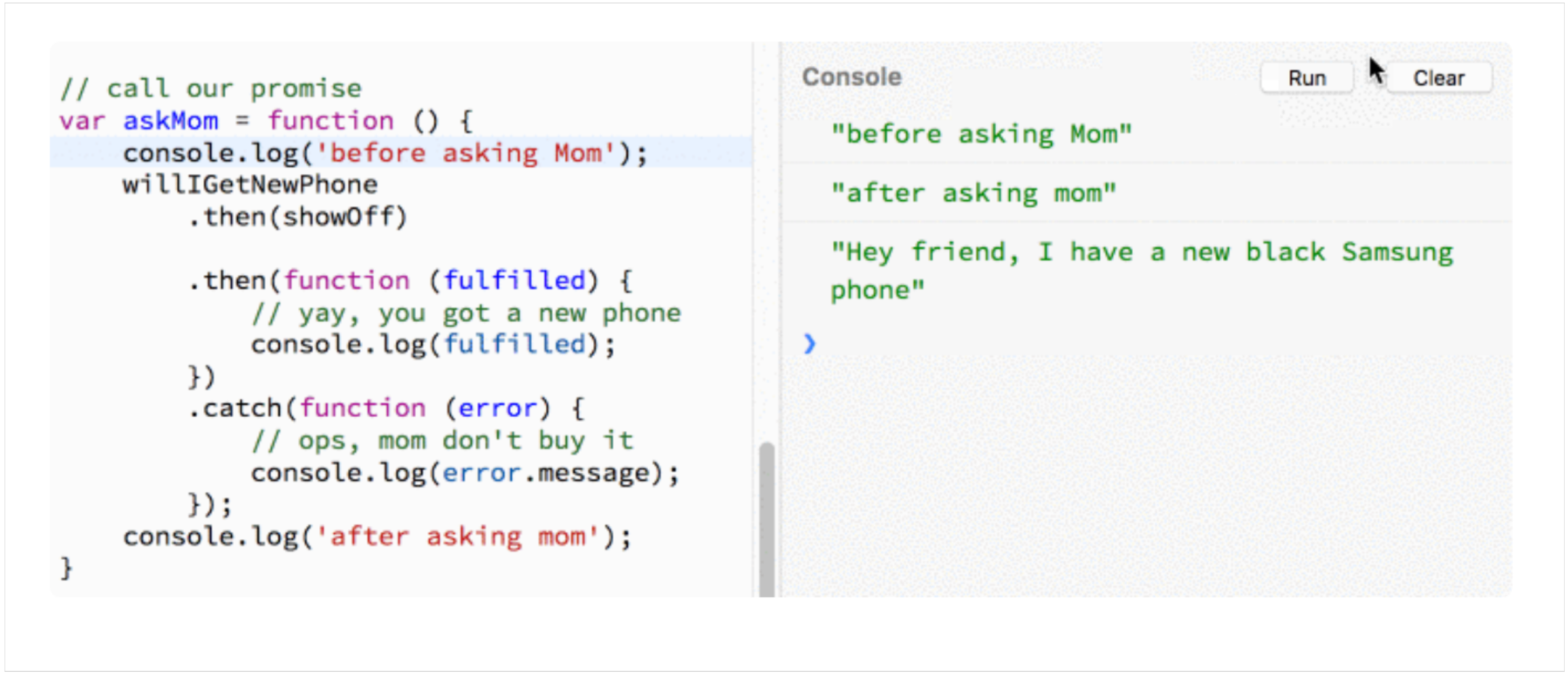
You, the kid, wouldn't stop playing while waiting for your mom's promise (the new phone). Right? That's something we call **asynchronous**, the code will run without blocking or waiting for the result. Anything that needs to wait for a promise to proceed, you put that in a .then.

Chaining Promises



```
// call our promise
var askMom = function () {
    console.log('before asking Mom');
    willIGetNewPhone
        .then(showOff)

        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
    console.log('after asking mom');
}
```

Chaining Promises



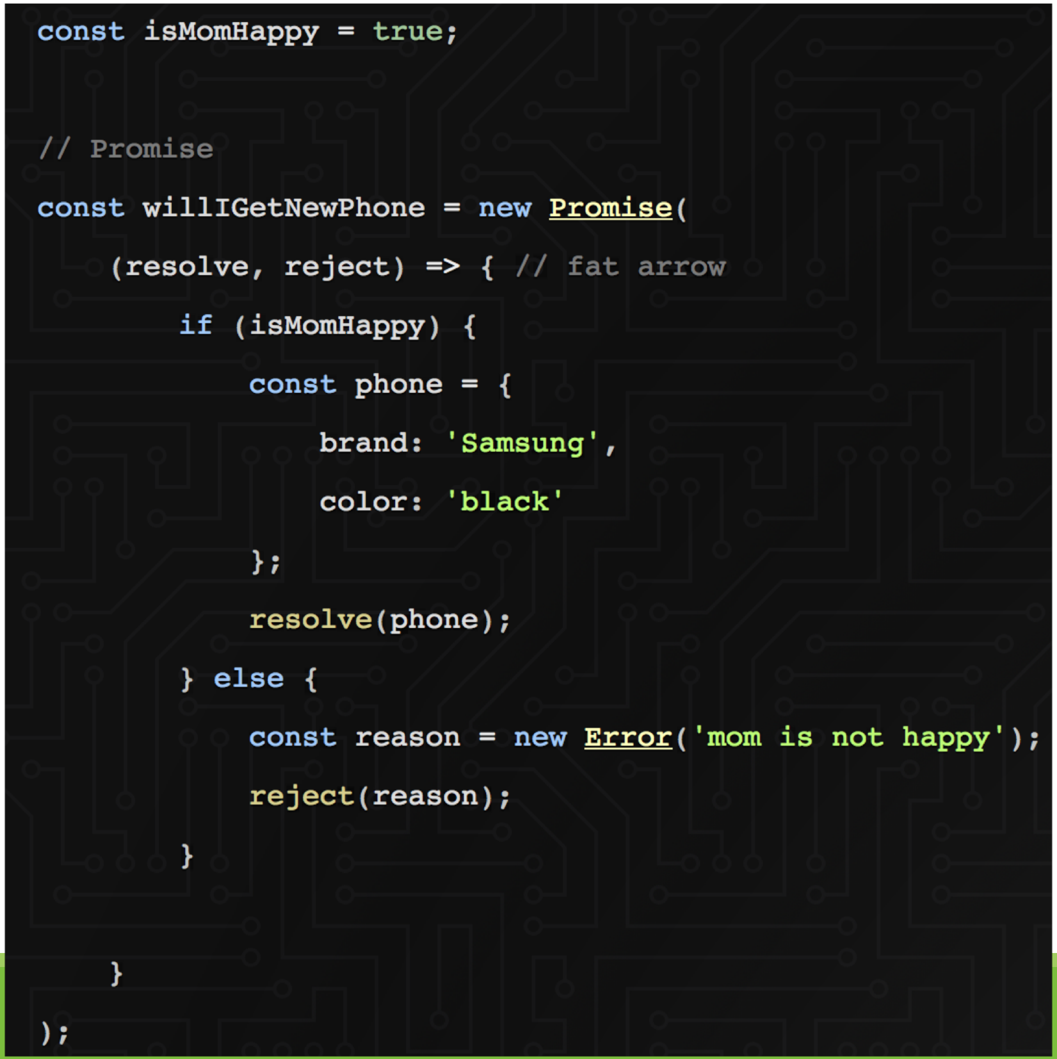
```
// call our promise
var askMom = function () {
    console.log('before asking Mom');
    willIGetNewPhone
        .then(showOff)

        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
    console.log('after asking mom');
}
```

```
"before asking Mom"
"after asking mom"
"Hey friend, I have a new black Samsung
phone"
❯
```

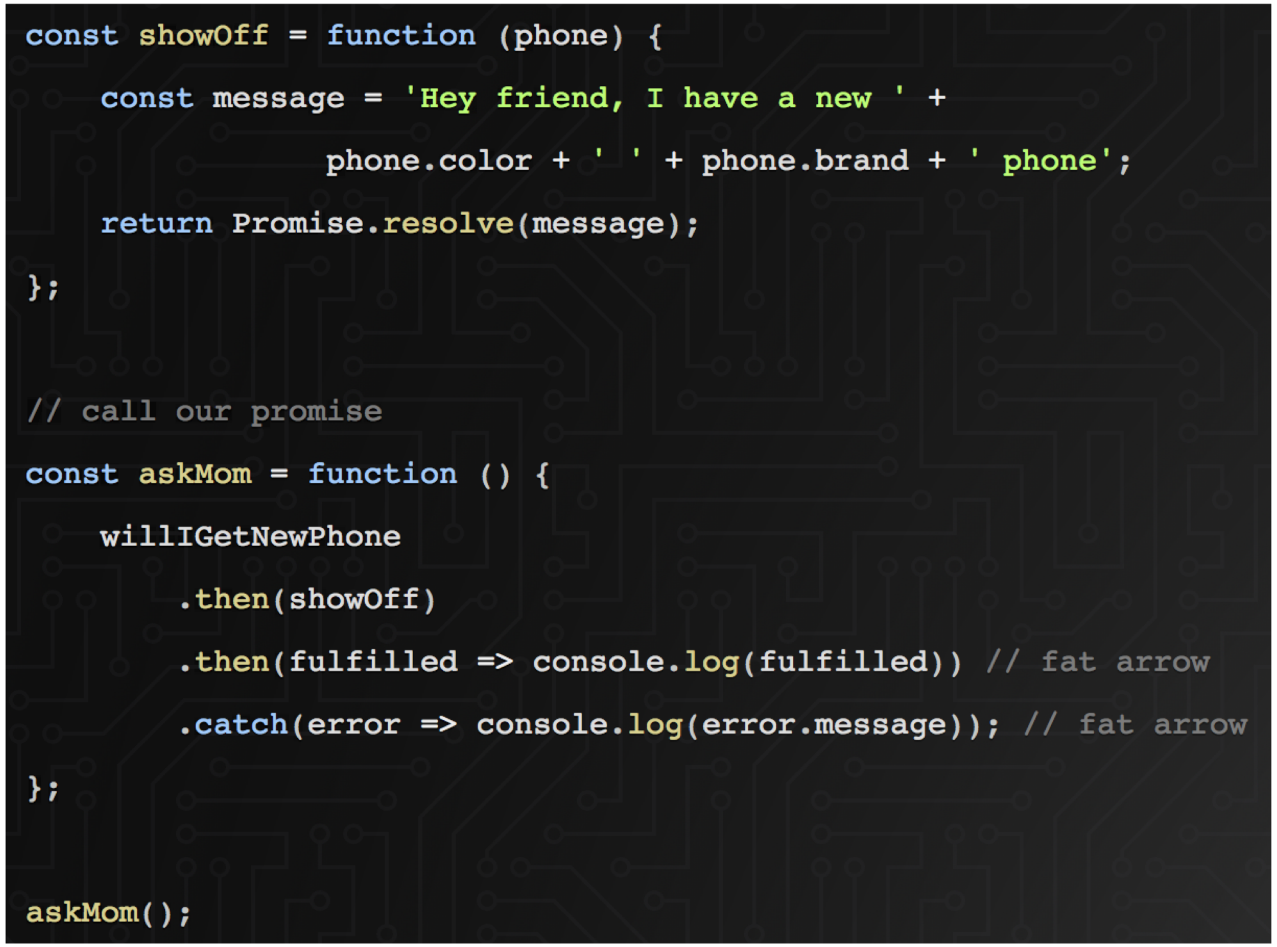
Promises & ECMAScript 6 (ES6)

The demo code works out of the box because ES6 supports promises natively. In addition, with ES6 functions, we can further simplify the code with fat arrow => and use const and let.

Promises & ECMAScript 6 (ES6)



```
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);
```



```
const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};

// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};

askMom();
```

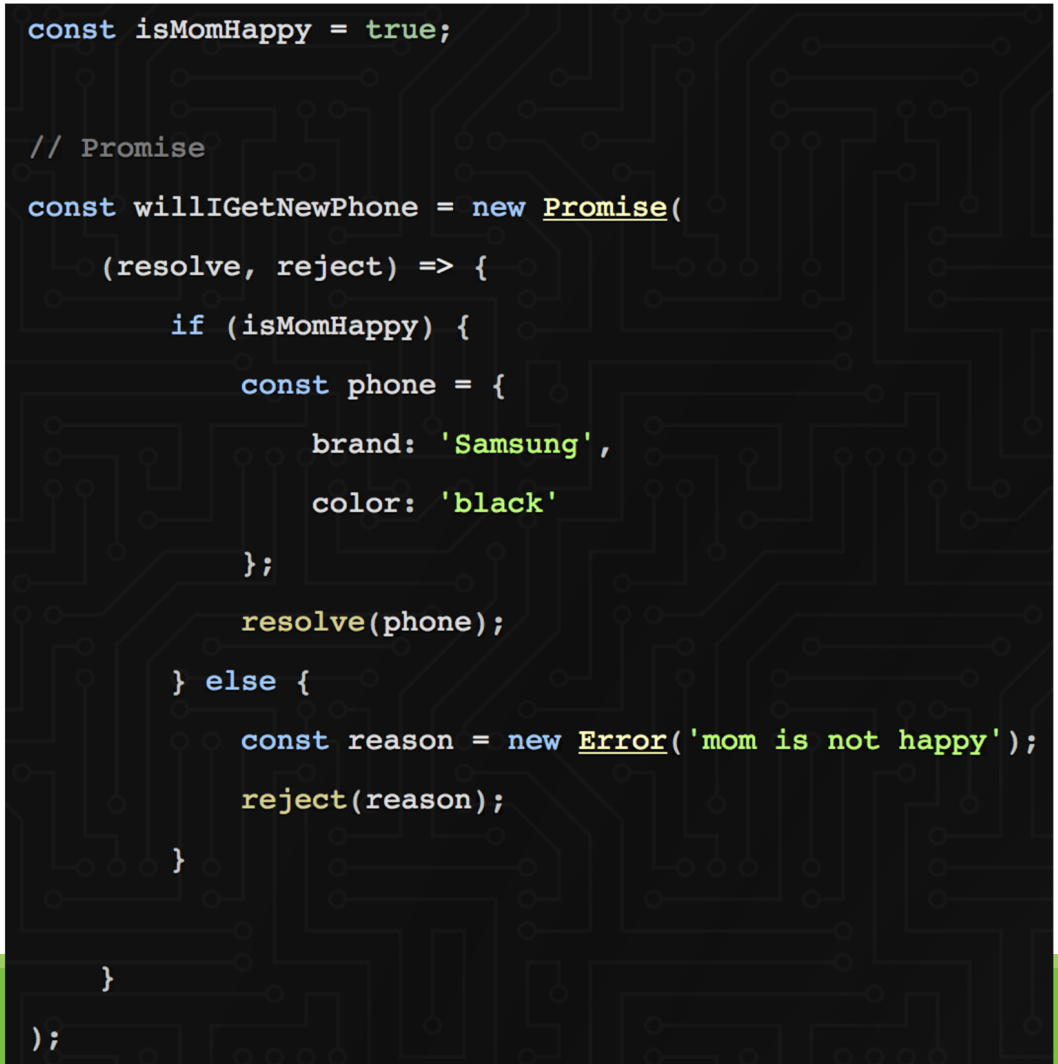
Promises & ES7 – Async/Await

Async/Await is a language feature that is a part of the ES7 standard. It was implemented in version 7.6 of NodeJs. If you are new to JavaScript this concept might be a bit hard to wrap your head around, but I would advise that you still give it a try. You don’t have to use it if you don’t want to. You will be fine with just using Promises.

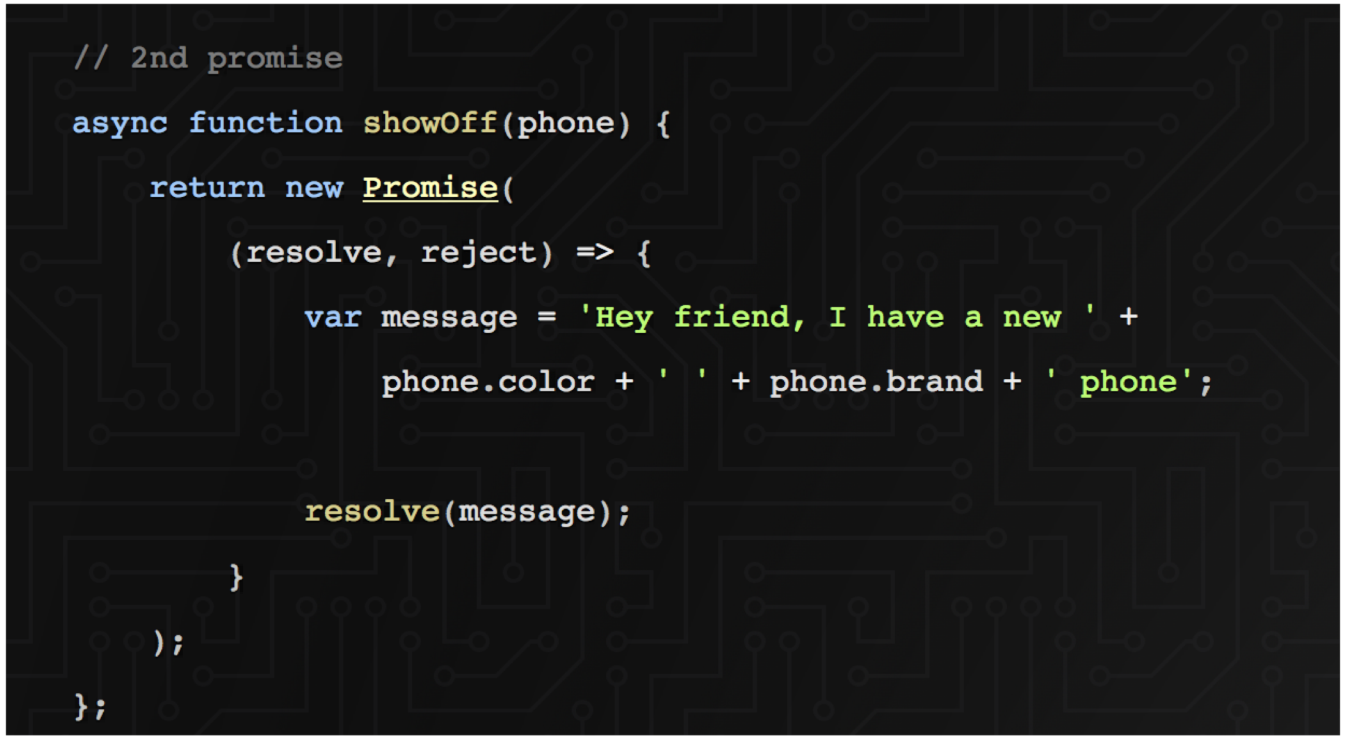
Async/Await is the next step in the evolution of handling asynchronous operations in JavaScript. It gives you two new keywords to use in your code: “async” and “await”. Async is for declaring that a function will handle asynchronous operations and await is used to declare that we want to “await” the result of an asynchronous operation inside a function that has the async keyword.

It makes the asynchronous syntax look prettier and easier to understand, without the .then and .catch.

Promises & ES7 – Async/Await

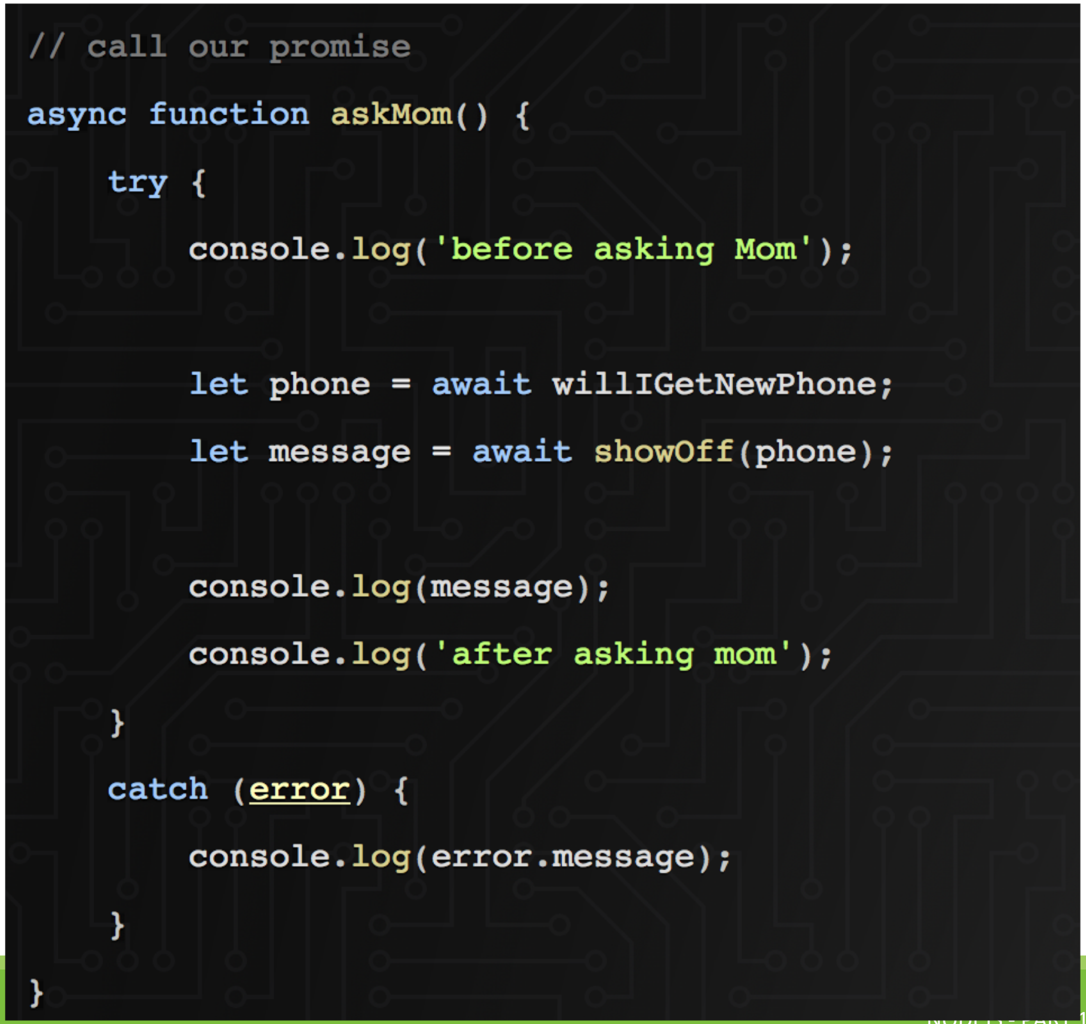


```
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => {
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);
```

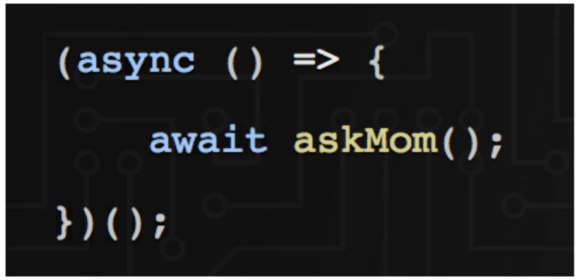


```
// 2nd promise
async function showOff(phone) {
    return new Promise(
        (resolve, reject) => {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

Promises & ES7 – Async/Await

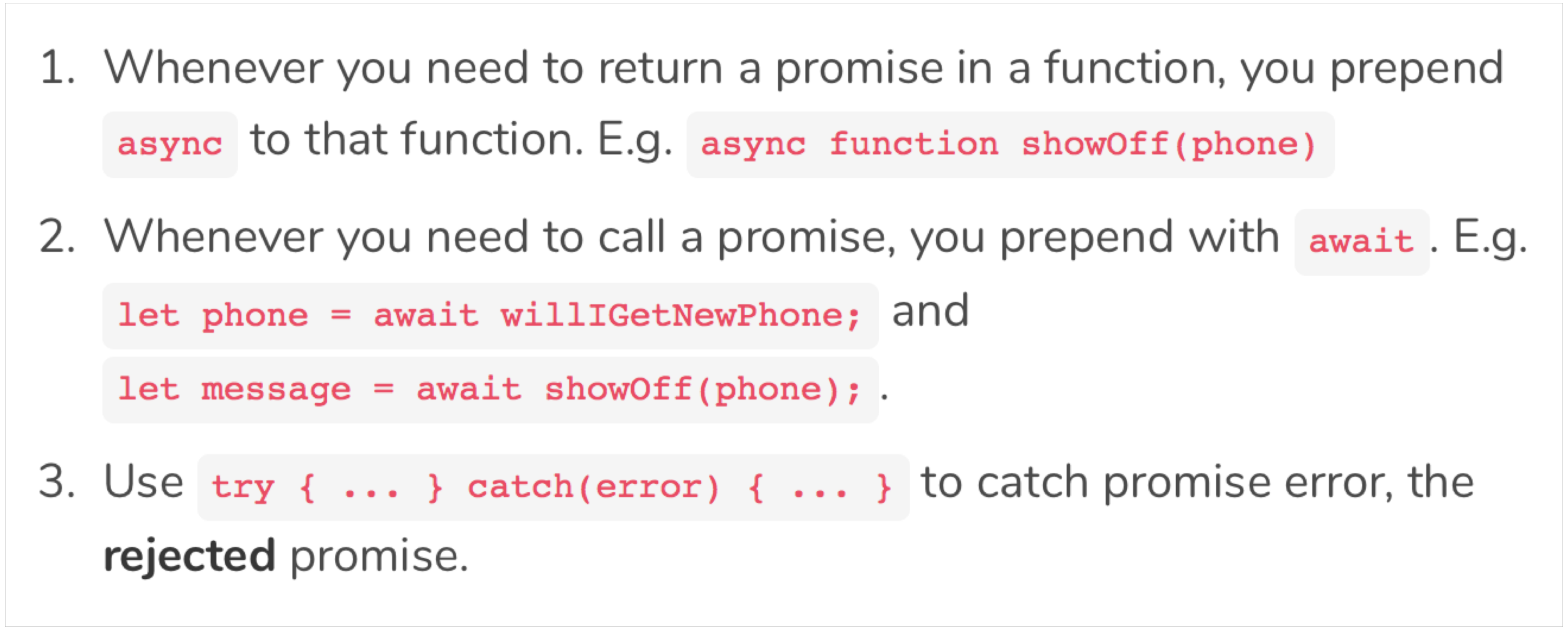


```
// call our promise
async function askMom() {
    try {
        console.log('before asking Mom');

        let phone = await willIGetNewPhone;
        let message = await showOff(phone);

        console.log(message);
        console.log('after asking mom');
    }
    catch (error) {
        console.log(error.message);
    }
}
```



```
(async () => {
    await askMom();
})();
```

Promises & ES7 – Async/Await



1. Whenever you need to return a promise in a function, you prepend `async` to that function. E.g. `async function showOff(phone)`
2. Whenever you need to call a promise, you prepend with `await`. E.g. `let phone = await willIGetNewPhone;` and `let message = await showOff(phone);`.
3. Use `try { ... } catch(error) { ... }` to catch promise error, the **rejected** promise.

https://scotch.io/tutorials/javascript-promises-for-dummies

Node Modules

THE BUILDING BLOCKS OF NODE

Node Modules

When you write **Node.js** applications, you could actually put all your code into one huge *index.js* file, no matter how large or complex your application is. The Node.js interpreter doesn't care. But in terms of code organization, you would end up with a hard to understand and hard to debug mess quite quickly. So as a developer, you should care about how to structure your code. This is where **modules** come in.

You can think of Node.js modules as JavaScript libraries - a certain part of your overall codebase (for example, a collection of functions) which you want to keep together, but which you also want to keep separated from the rest of your codebase to keep things cleanly separated.

Built-in Modules

Even if we don't create any Node.js modules ourselves (more on that next), we already have modules at our disposal because the Node.js environment provides **built-in** modules for us – for example :

```
var http = require('http');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.end('Hello World!');
}).listen(8080);
```

Here, requiring the http module gives us direct access to an http object, whose functions, like createServer, we can then use.

External Modules

The built-in modules which ship with Node.js allow to solve a lot of coding problems without reinventing the wheel for every new application, but what really boosts Node.js programming productivity is the huge ecosystem of open source modules provided by the Node.js community.

These modules can be integrated into our codebase, too, but because they are not built-in and don't ship directly with each installation of Node.js, it is not enough to *require* them from your own code. You need to install the codebase containing the external module locally first, which is made very easy thanks to *NPM*, the Node Package Manager.

NPM

- Common npm commands:
  - npm init initialize a package.json file
  - npm install <package name> -g install a package, if –g option is given package will be installed globally, --save and --save-dev will add package to your dependencies
  - npm install install packages listed in package.json
  - npm ls –g listed local packages (without –g) or global packages (with –g)
  - npm update <package name> update a package

External Modules

Node.js relies heavily on **modules** and allows you to also **create you own.**

Creating a module is easy, just put your JavaScript code in a separate js file and include it in your code by using the keyword **require**, like:

```
var modulex = require(‘./modulex’);
```

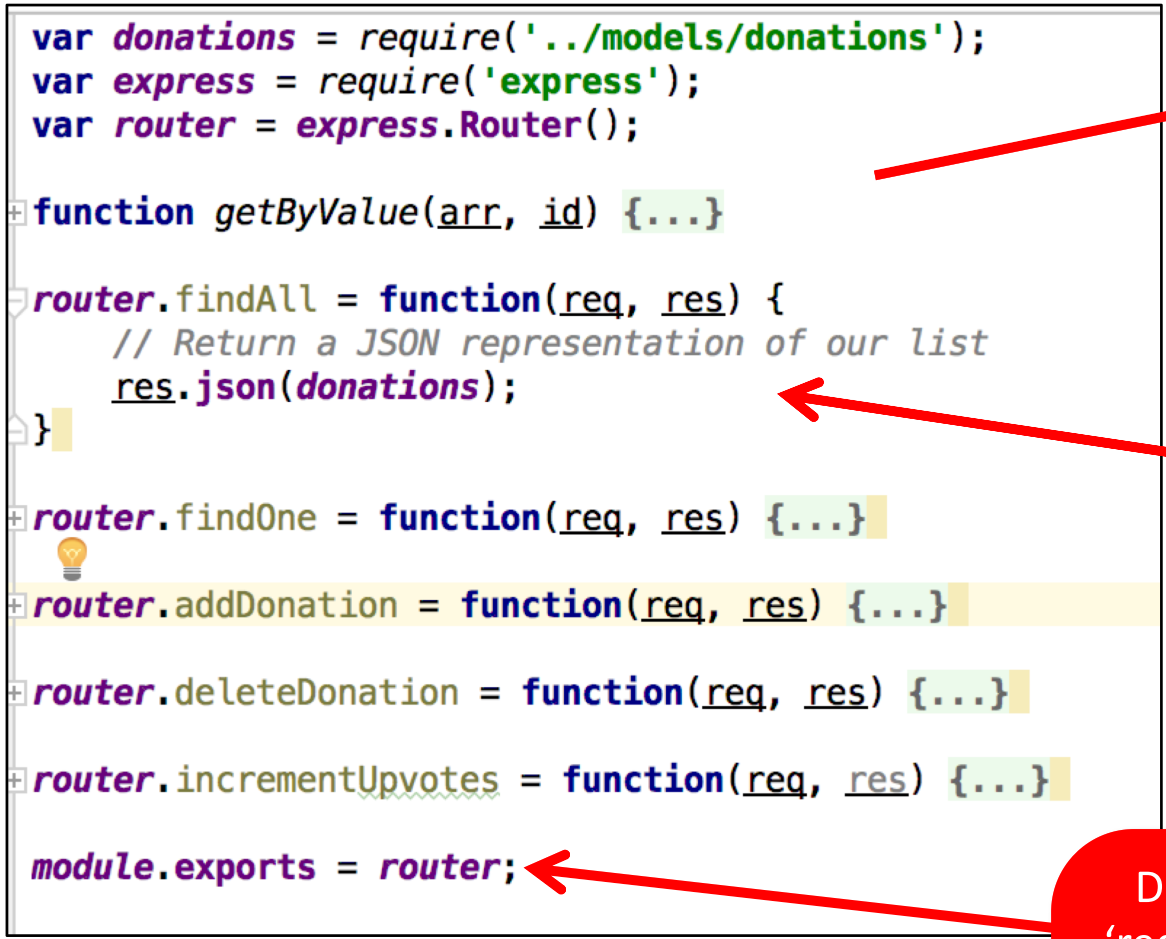
Libraries in Node.js are called *packages* and they can be installed by typing

```
npm install “package_name”; //installs in current folder
//package should be available in npm registry @ nmpjs.org
```

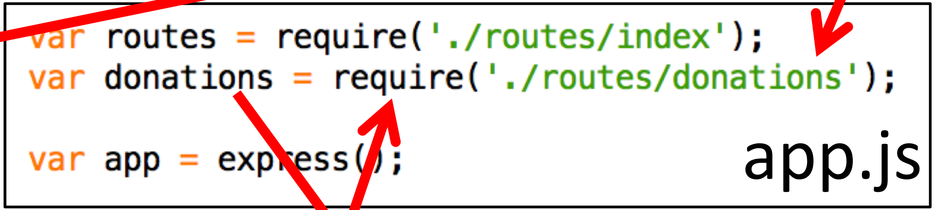
NPM downloads and installs modules, placing them into a **node_modules** folder in your current folder.

Creating your own Node Modules

donations.js

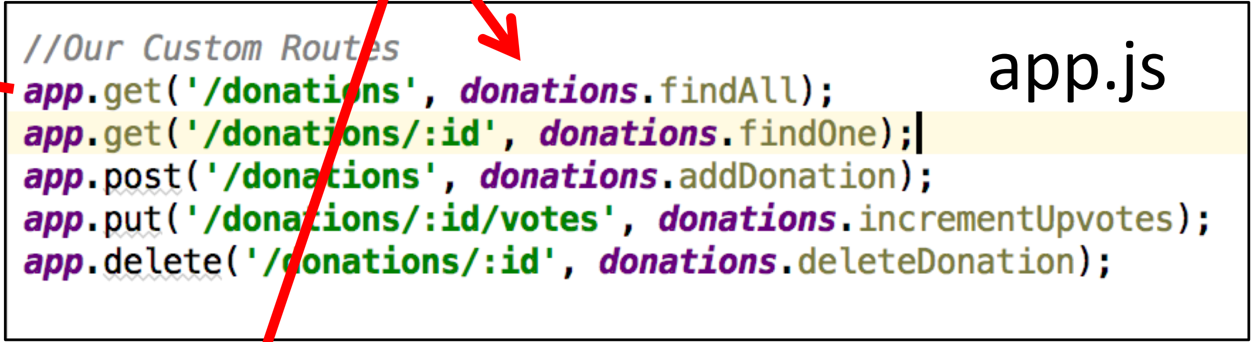


```
var donations = require('../models/donations');
var express = require('express');
var router = express.Router();

function getByValue(arr, id) {...}

router.findAll = function(req, res) {
    // Return a JSON representation of our list
    res.json(donations);
}

router.findOne = function(req, res) {...}

router.addDonation = function(req, res) {...}

router.deleteDonation = function(req, res) {...}

router.incrementUpvotes = function(req, res) {...}

module.exports = router;
```



```
var routes = require('./routes/index');
var donations = require('./routes/donations');

var app = express();
```

app.js



```
//Our Custom Routes
app.get('/donations', donations.findAll);
app.get('/donations/:id', donations.findOne);
app.post('/donations', donations.addDonation);
app.put('/donations/:id/votes', donations.incrementUpvotes);
app.delete('/donations/:id', donations.deleteDonation);
```

app.js

Defines what 'require' returns

The require search

- Require searches for modules based on path specified:

```
var myMod = require('./myModule'); //current dir
var myMod = require('../myModule'); //parent dir
var myMod = require('../modules/myModule');
```

- Just providing the module name will search in node_modules folder

```
var myMod = require('myModule');
```

Resources & References

Official Tutorial – https://nodejs.org/documentation/tutorials/

Official API – https://nodejs.org/api/

Developer Guide – https://nodejs.org/documentation

Video Tutorials – http://nodetuts.com

Video Introduction – https://www.youtube.com/watch?v=FqMIyTH9wSg

YouTube Channel – https://www.youtube.com/channel/UCvhIsEIBIfWSn_Fod8FuuGg

Articles, explanations, tutorials – https://nodejs.org/community/

https://www.sitepoint.com/an-introduction-to-node-js/

https://snipcart.com/blog/javascript-nodejs-backend-development

https://scotch.io/tutorials/javascript-promises-for-dummies

https://www.nodebeginner.org/blog/post/nodejs-tutorial-what-are-node.js-modules/

Questions?