Fundamentals of Web Applications
COMP 2406A – Winter 2020

Introduction to Node.js

Dave McKenney
david.mckenney@carleton.ca

Learning Outcomes

by the End of this Lecture, Students that have Completed the Reading Assignment and Review Questions should be Able to:

Understand the basic architecture of Node.js

Explain how an event-loop operates

Create and use modules in Node.js

Create a basic HTTP server in Node.js

What is Node.js

“As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications” - https://nodejs.org/en/about/

What is Node.js

Node.js is:

Open source

Cross platform

Asynchronous

Event-driven

Single-threaded

Server-side Javascript

Typical HTTP Request/Response

We saw that HTTP uses a request/response model

Client makes a request, it arrives at the server, the server handles the request, the server sends a response

This part often involves: reading files, databases, network resources, etc.

Benefits of Asynchronous I/O

Asynchronous I/O is what gives Node.js a huge advantage

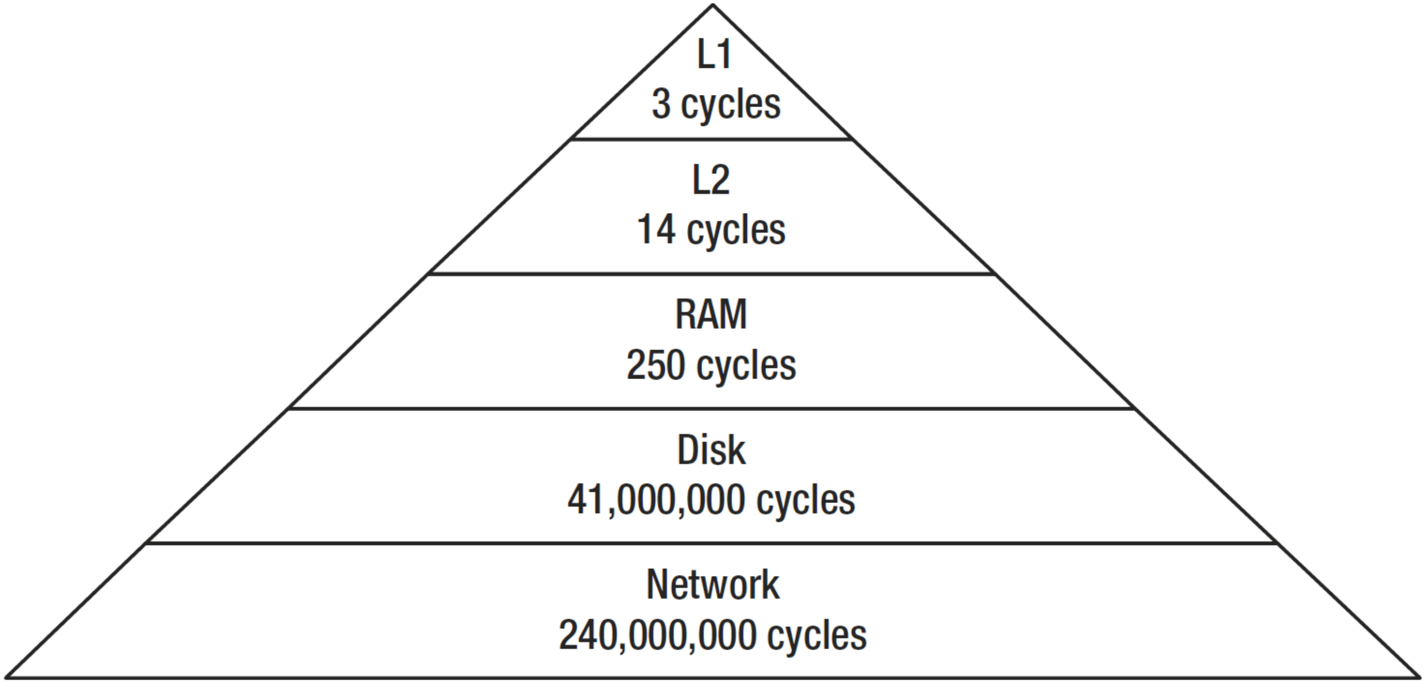
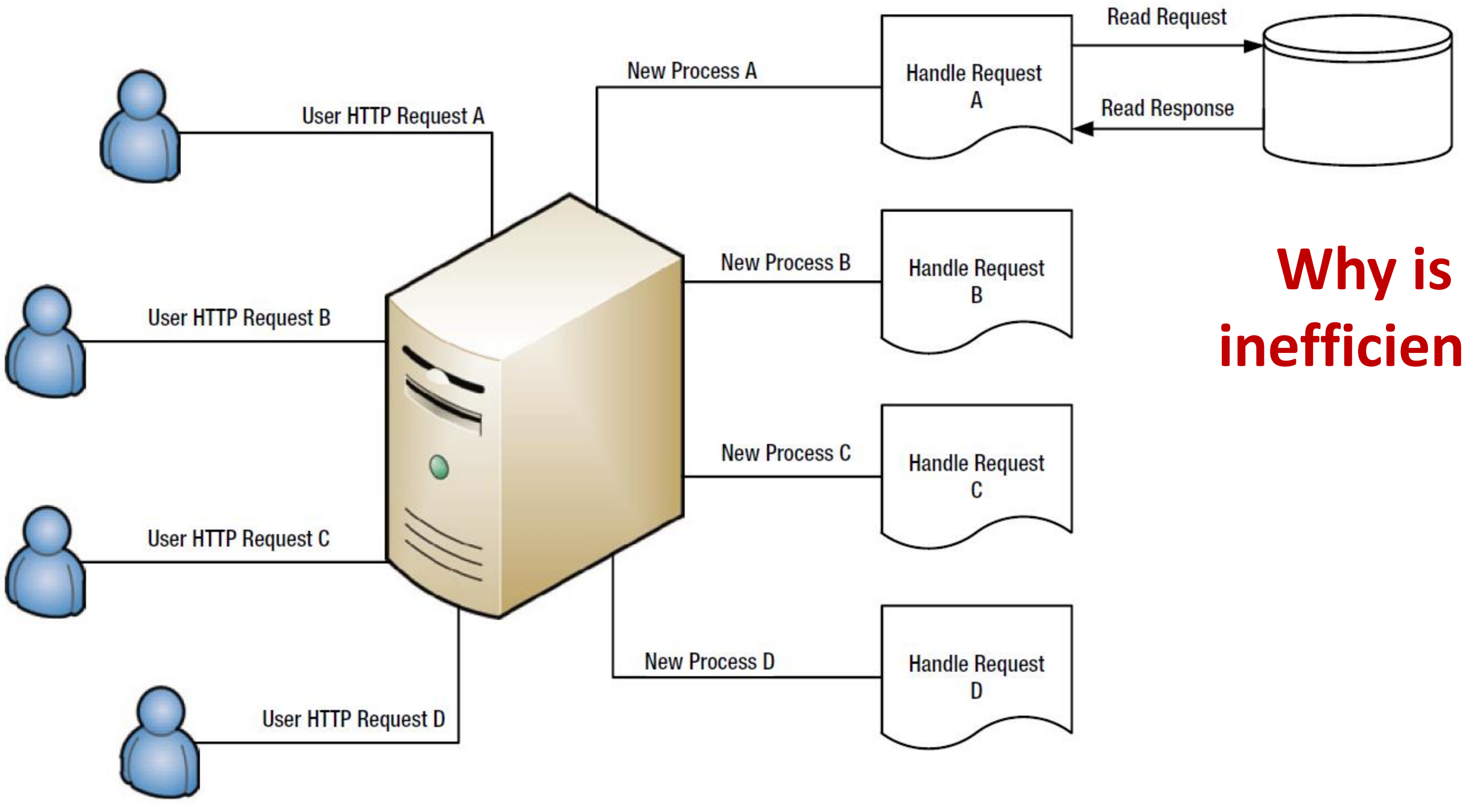


Figure 2-1. *Comparing common I/O sources*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp23.

Concurrent vs. Asynchronous Servers

A classic architecture for web servers – a process is spawned for every request



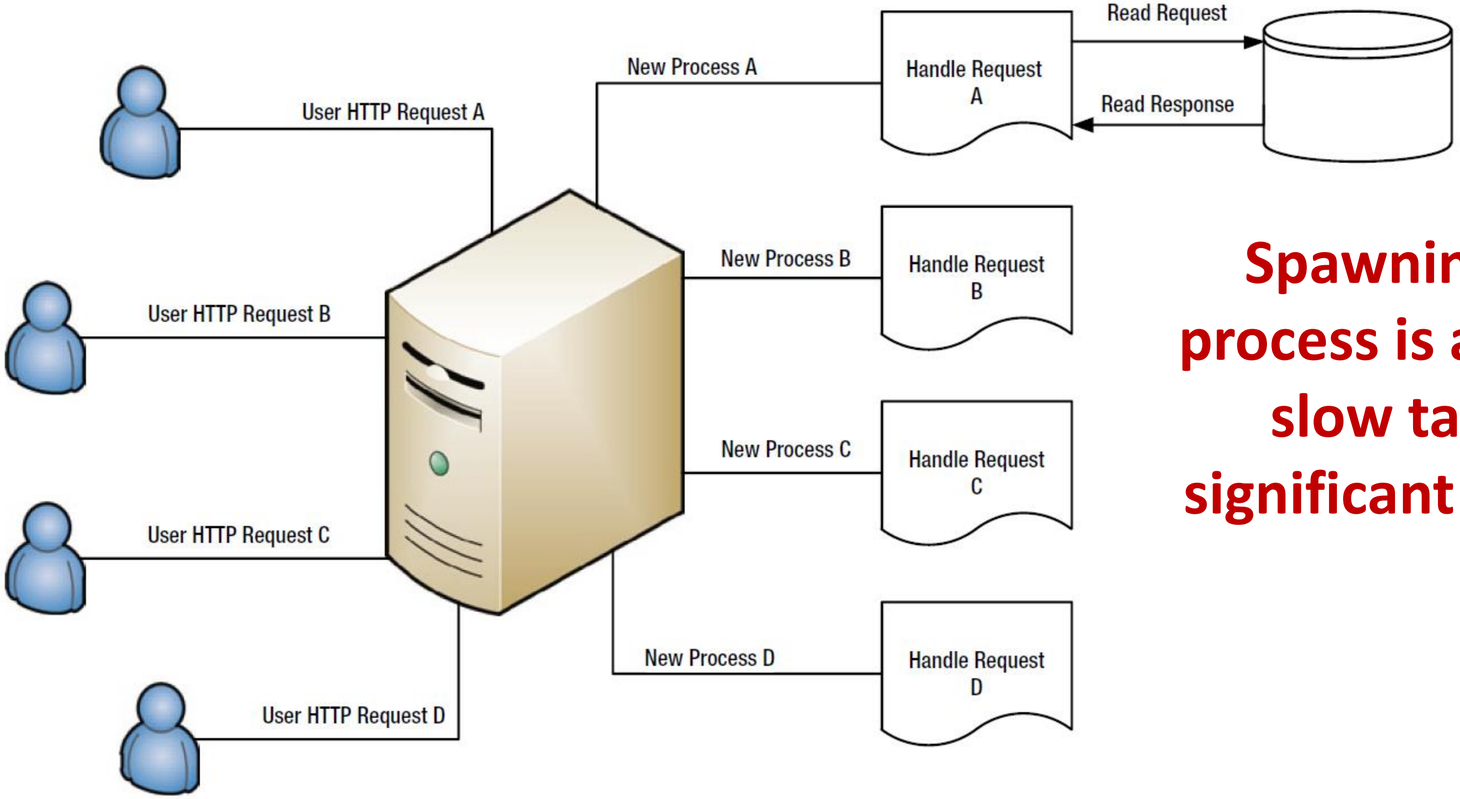
Why is this an inefficient design?

Figure 2-2. *Traditional web server using Processes*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp24.

Concurrent vs. Asynchronous Servers

A classic architecture for web servers – a process is spawned for every request



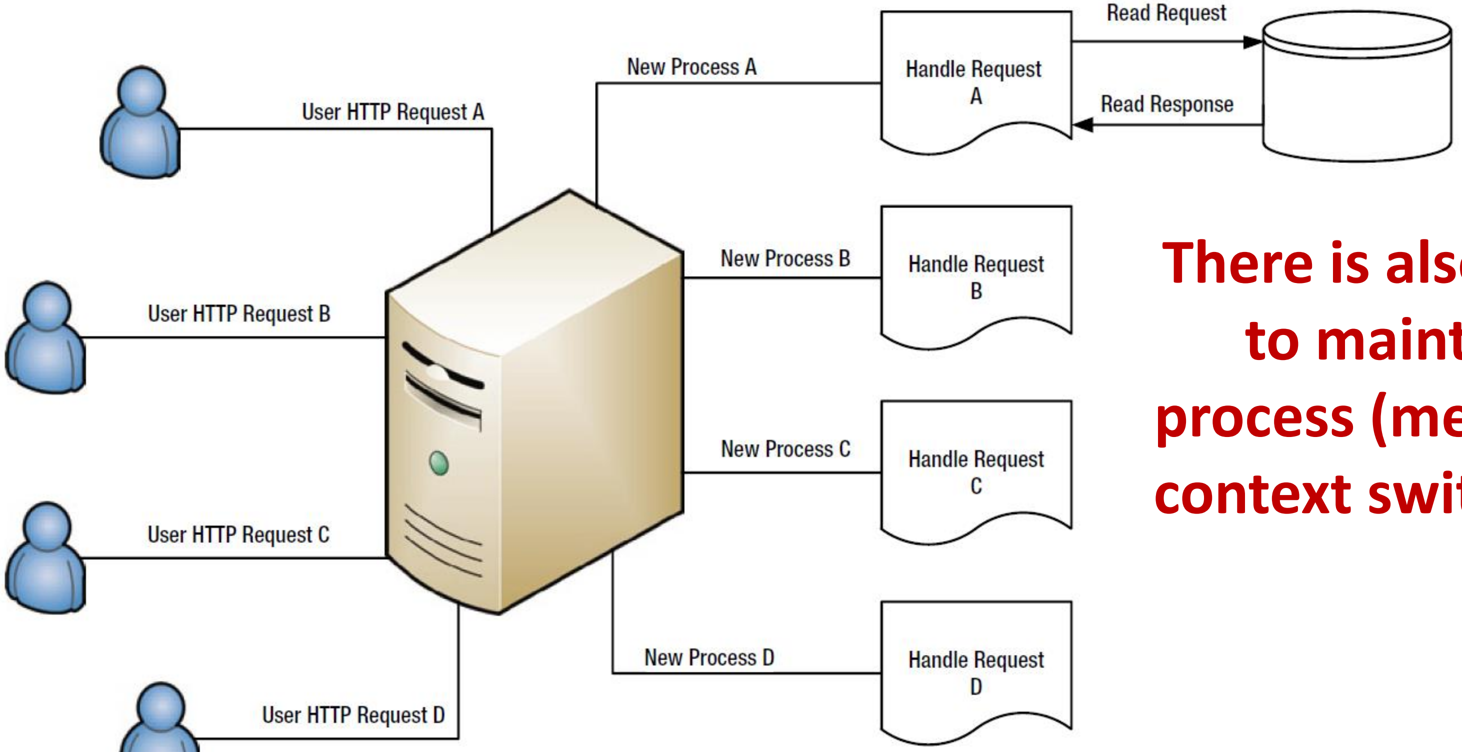
Spawning a new process is a relatively slow task with significant overhead.

Figure 2-2. *Traditional web server using Processes*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp24.

Concurrent vs. Asynchronous Servers

A classic architecture for web servers – a process is spawned for every request



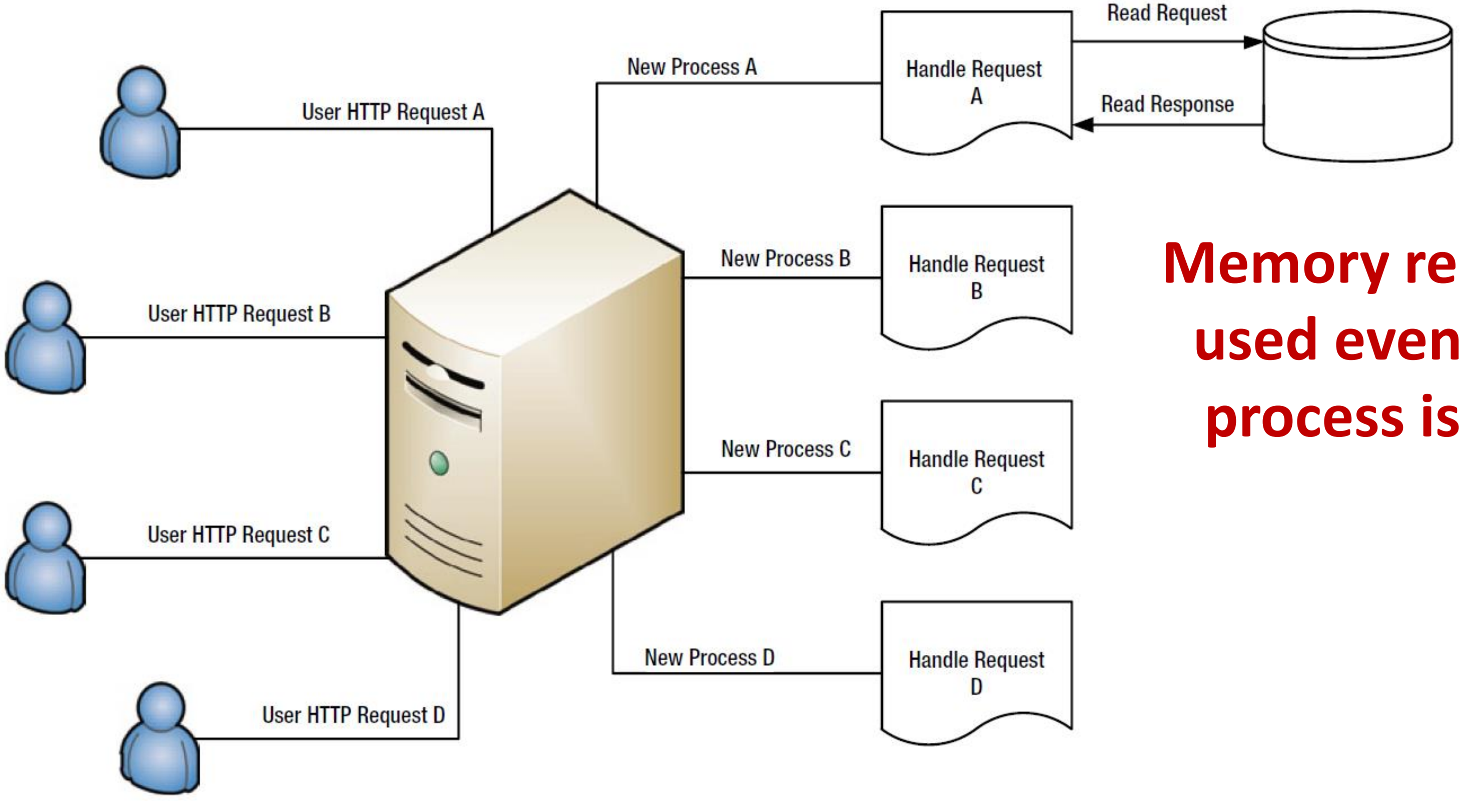
There is also overhead to maintain that process (memory, CPU, context switching, etc.)

Figure 2-2. *Traditional web server using Processes*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp24.

Concurrent vs. Asynchronous Servers

A classic architecture for web servers – a process is spawned for every request



Memory resources are used even when the process is waiting…

Figure 2-2. *Traditional web server using Processes*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp24.

Concurrent vs. Asynchronous Servers

An improved architecture uses a thread pool, where each request is handled by a single thread

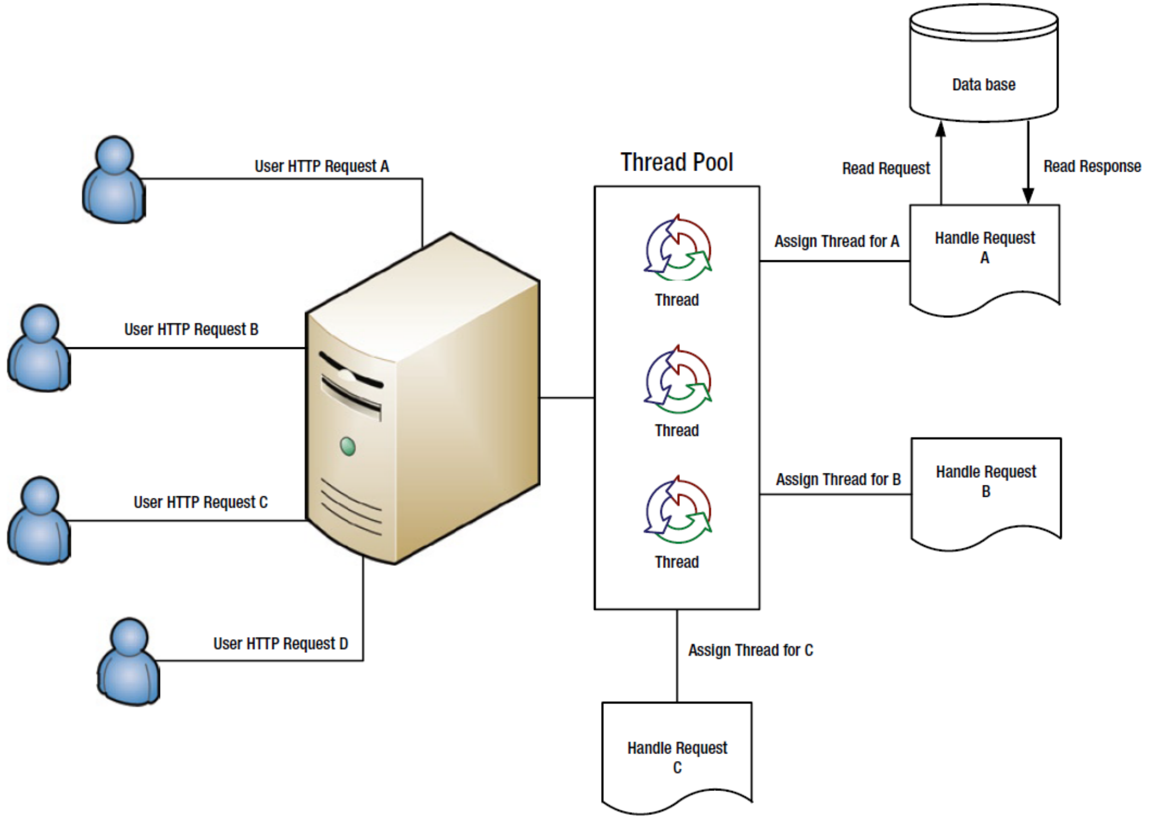


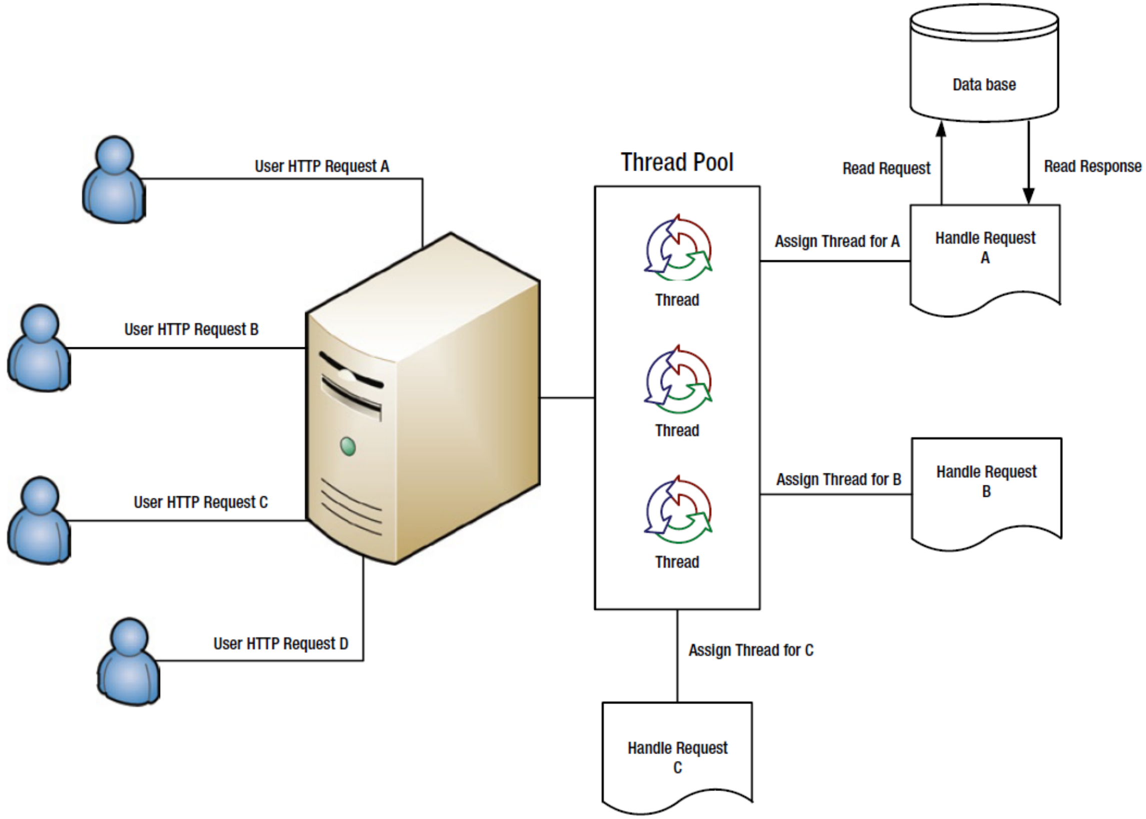
Figure 2-3. *Traditional web server using a thread pool*

Threads require less resources than processes and don't need to be newly created on each request

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp25.

Concurrent vs. Asynchronous Servers

An improved architecture uses a thread pool, where each request is handled by a single thread



There is still overhead involved with 'context switching' when many threads are running at one time

Figure 2-3. *Traditional web server using a thread pool*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp25.

Concurrent vs. Asynchronous Servers

Node.js uses a single thread asynchronous event-based architecture, made popular by NGINX

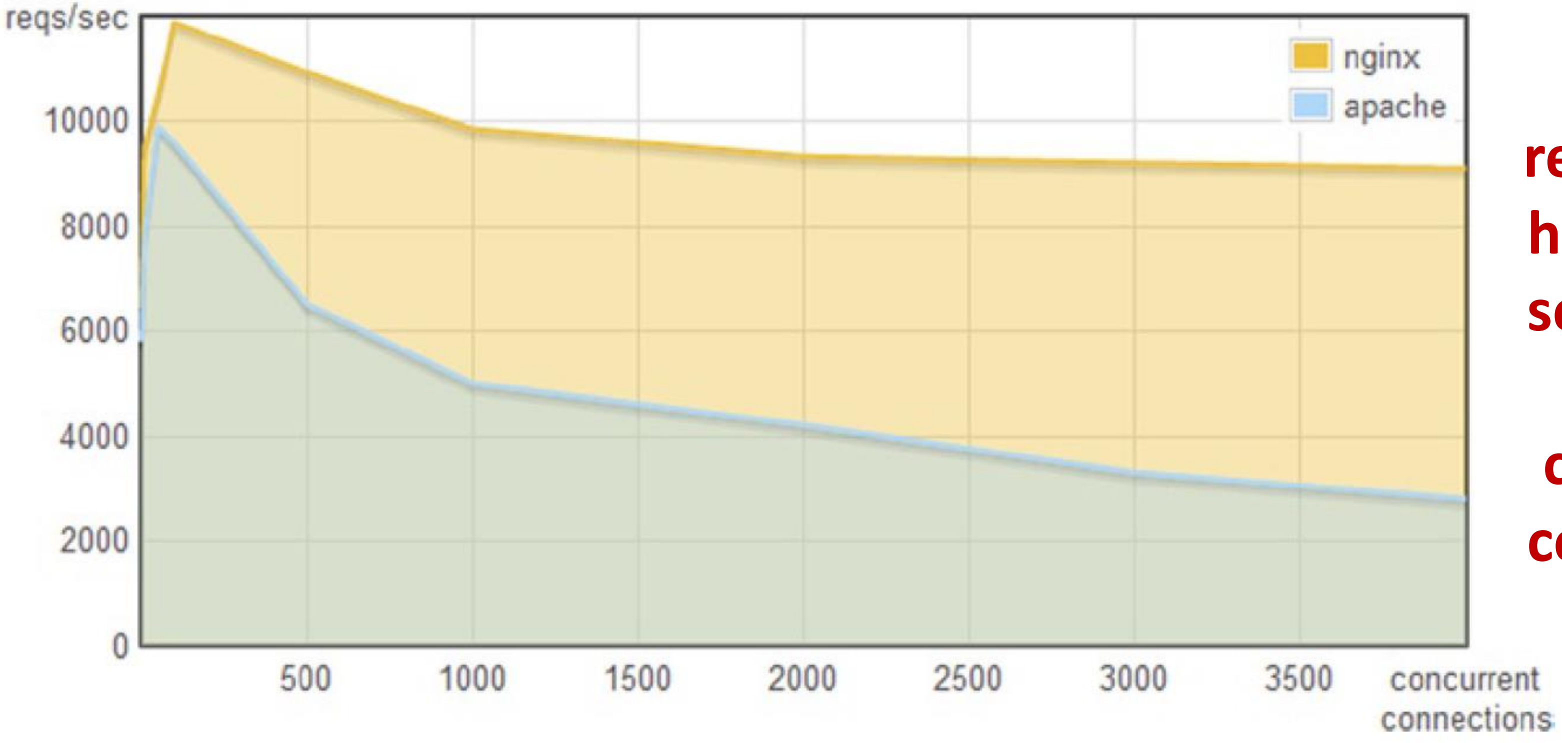


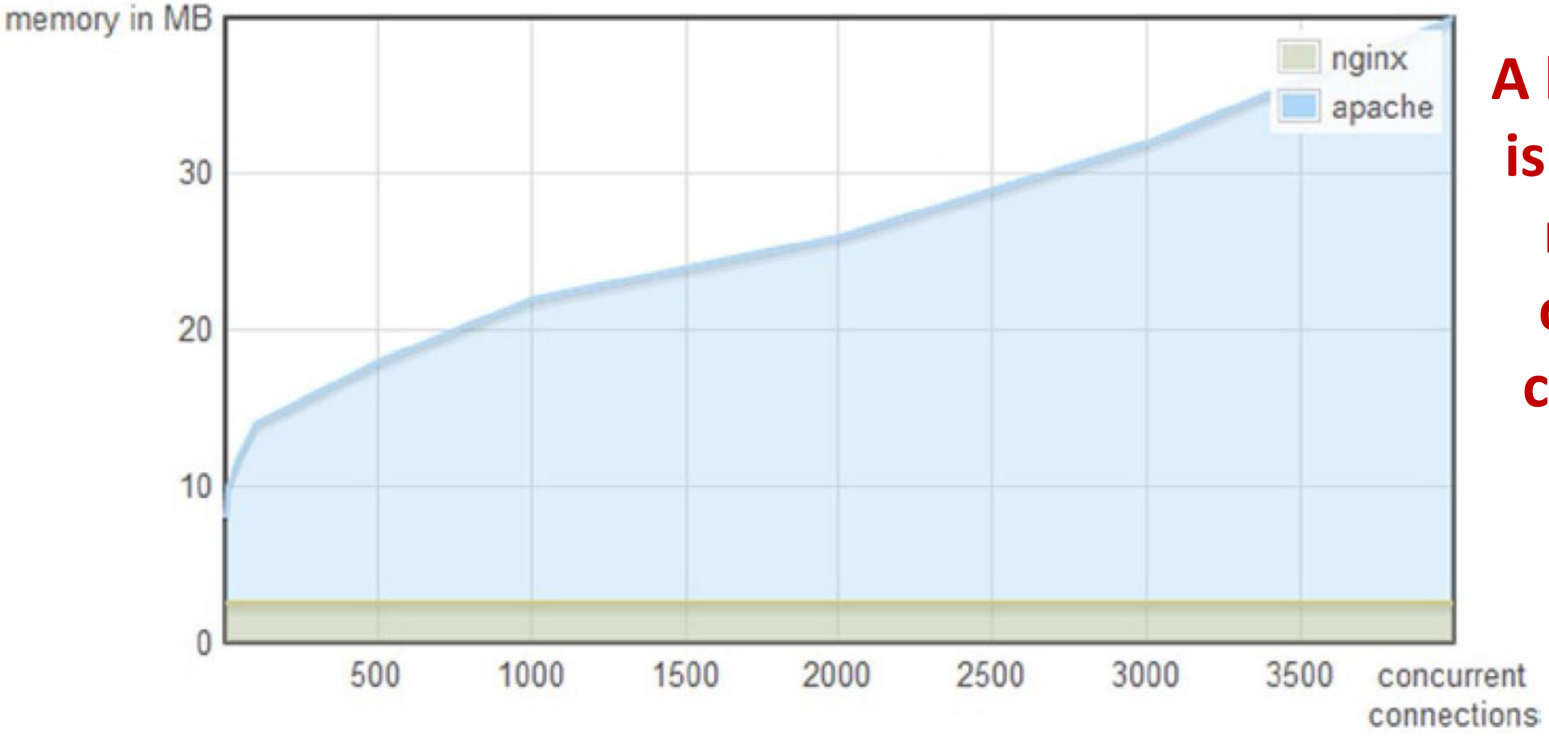
Figure 2-4. *Nginx vs. Apache requests/second vs. concurrent open connections*

More requests are handled per second with many concurrent connections

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp26.

Concurrent vs. Asynchronous Servers

Node.js uses a single thread asynchronous event-based architecture, made popular by NGINX



A lot less RAM is used as the number of concurrent connections increases

Figure 2-5. *Nginx vs. Apache memory usage vs. concurrent connections*

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp26.

How Does It Work?

A single thread runs our code, just like in the browser

Input and output are asynchronously handled by a lower-level system

We define callback functions to handle whatever result is produced by the input/output operations (and closures can give us access to anything we need)

How Does It Work?

As examples, we can look at the asynchronous code defined in 06-timer.js and 06-file-read.js

The structure of the file read example will be a common pattern we see throughout Node.js…

How Does It Work?

We are essentially saying ‘go perform this operation, and once the result is available in RAM, do ___’

Our server thread is free to continue handling other events while the input/output is occurring

How Does It Work?

This all happens seamlessly, as it is a core principle of Node.js’s architecture

We get the same feel of multi-threaded computing, without the hassle of managing/maintaining a more complex architecture

This is a similar idea to the ‘layers’ abstraction we saw when discussing HTTP and the web

How Does It Work?

This event-driven callback model is made possible with an ‘event loop’

This is a common idea seen in GUI/browser programming (which are also often event-based)

To understand how the event loop works, we will look at a similar browser example

The Event Loop

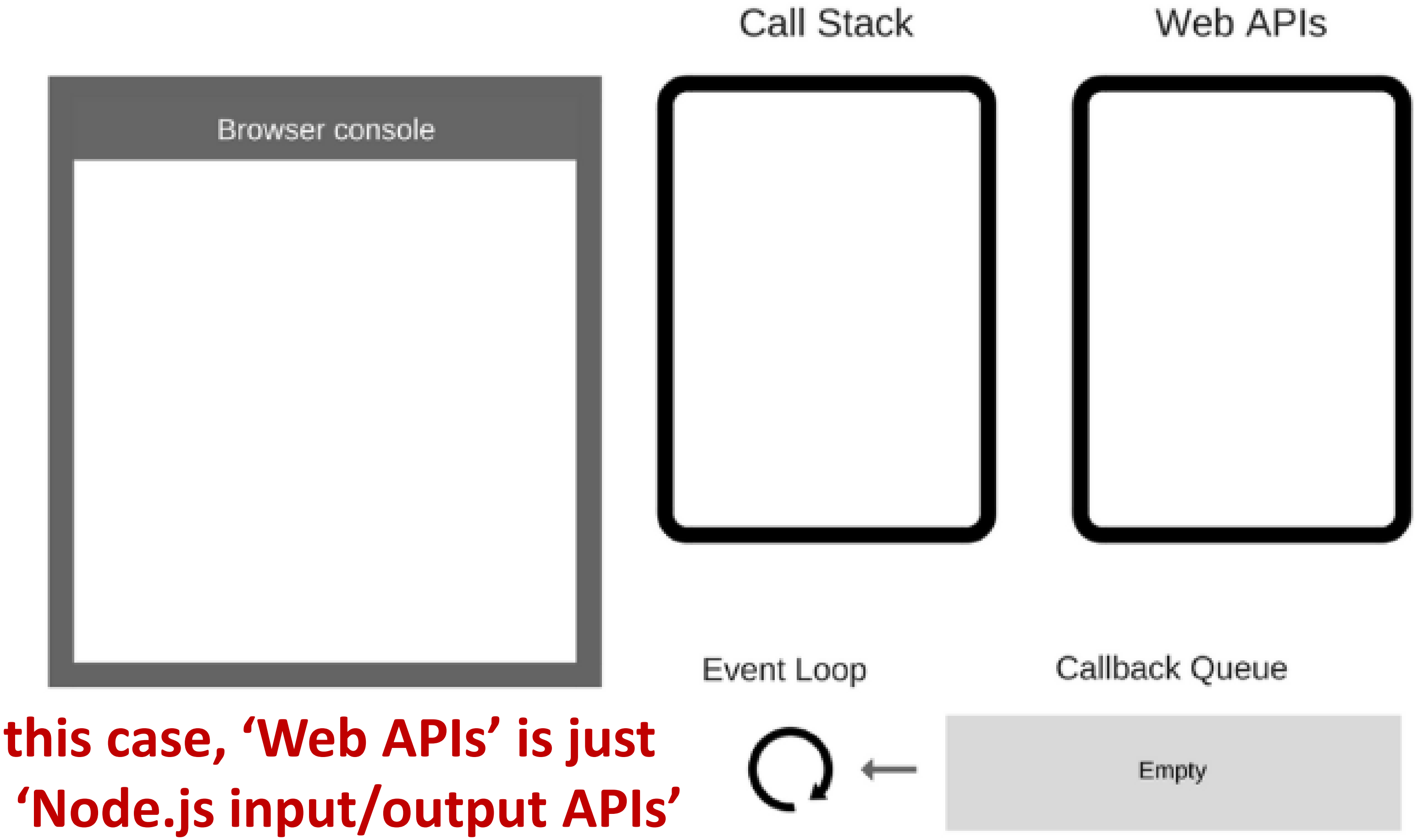
Assume we execute this code block in the browser:

```
console.log('Hi’);
setTimeout(function cb1() {
    console.log('cb1’);
}, 5000);
console.log('Bye');
```

https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

Initially everything is in an empty state

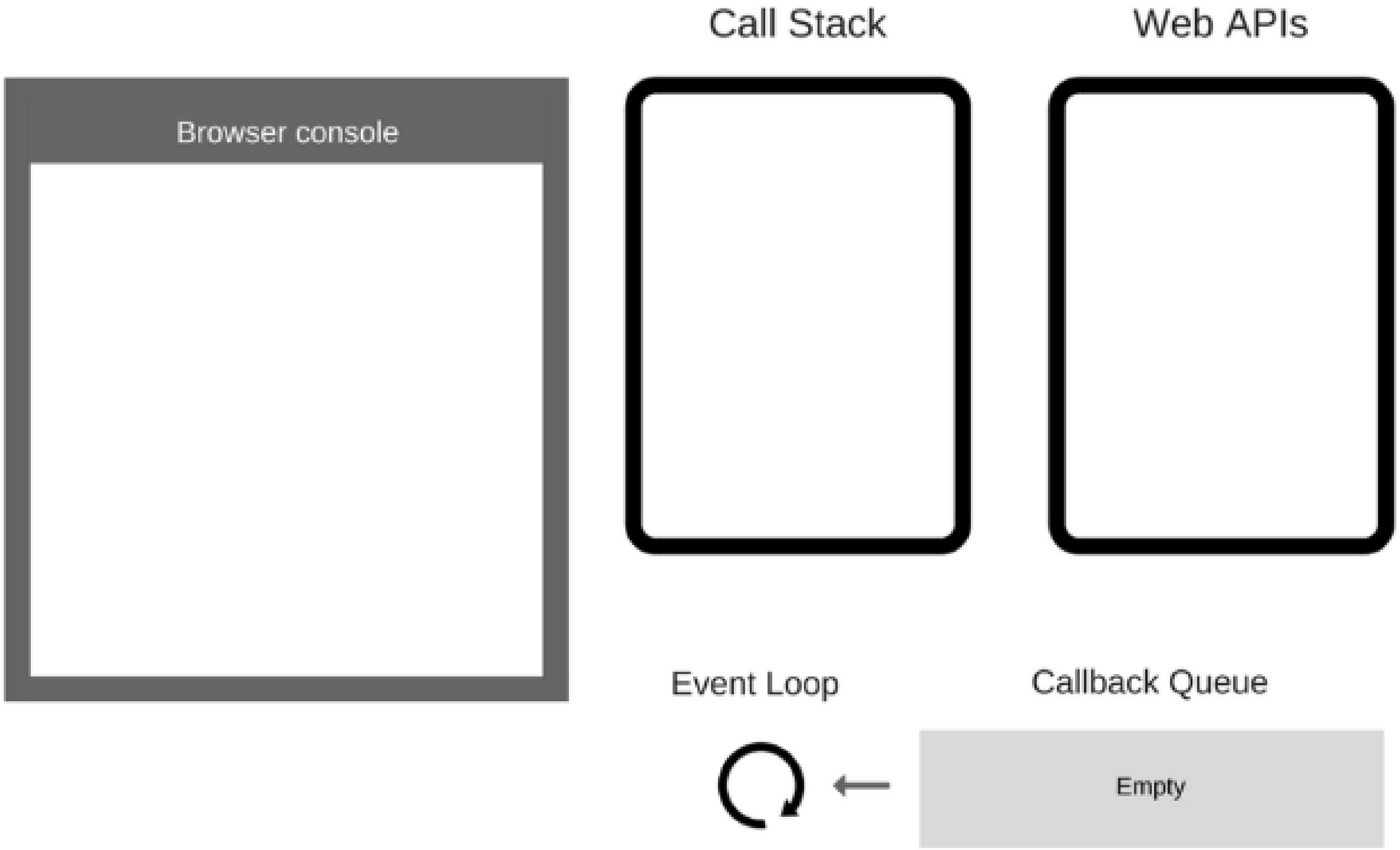


In this case, ‘Web APIs’ is just like ‘Node.js input/output APIs’

https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

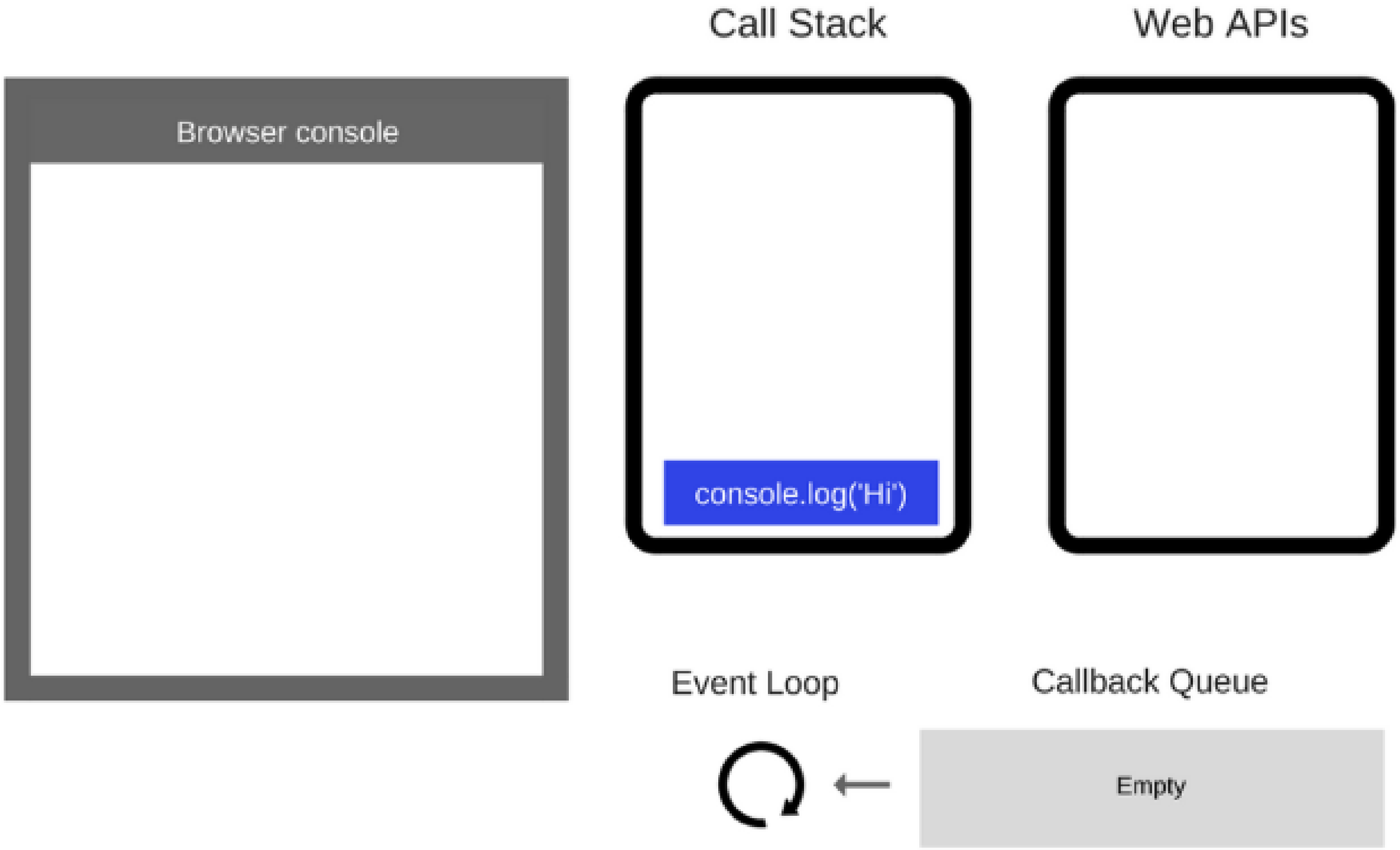
We execute: `console.log(‘Hi’);`



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

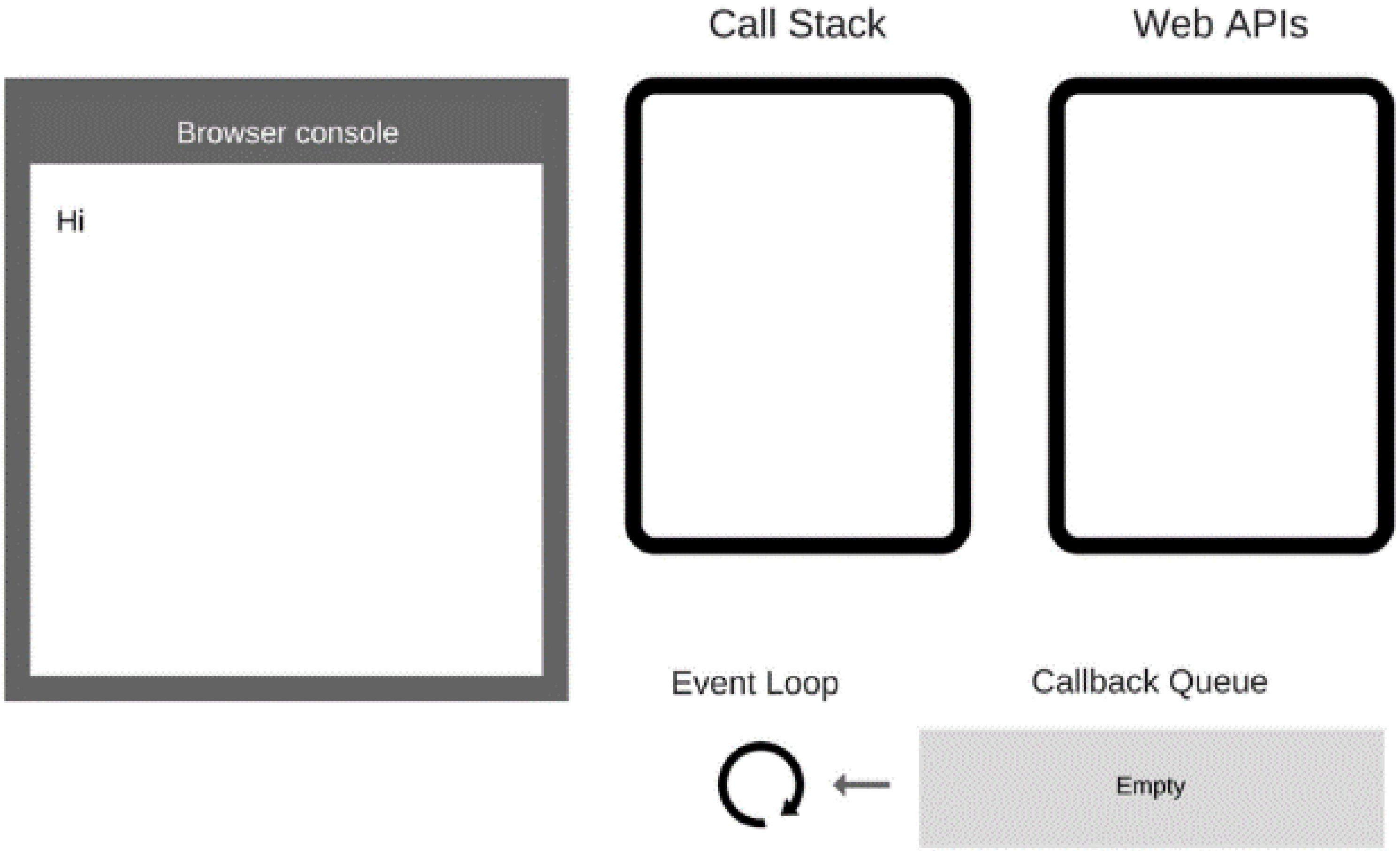
`console.log('Hi');` is added to the call stack



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

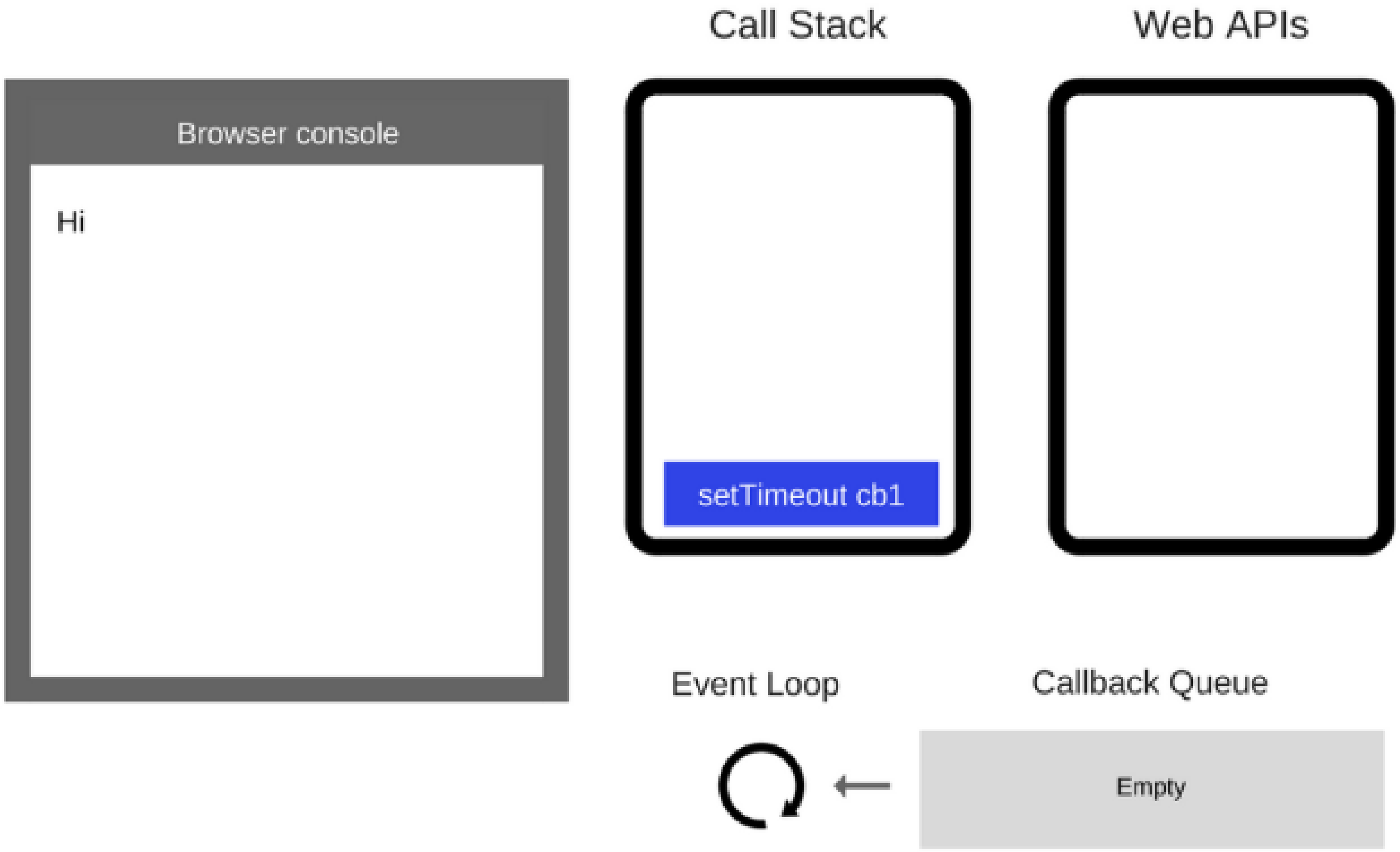
`console.log(‘Hi’);` is executed/removed



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

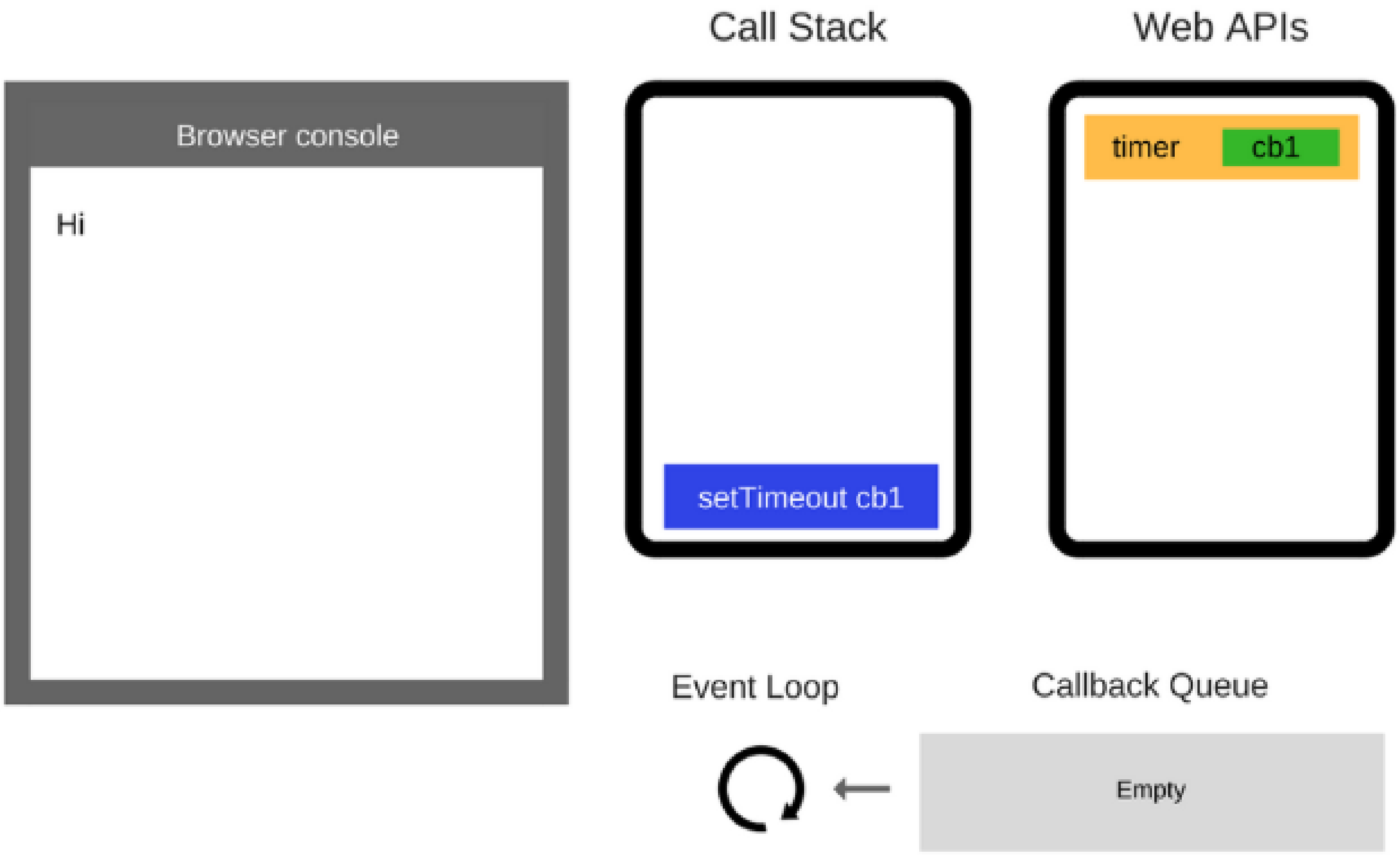
`setTimeout(function cb1() { ... })` is called



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

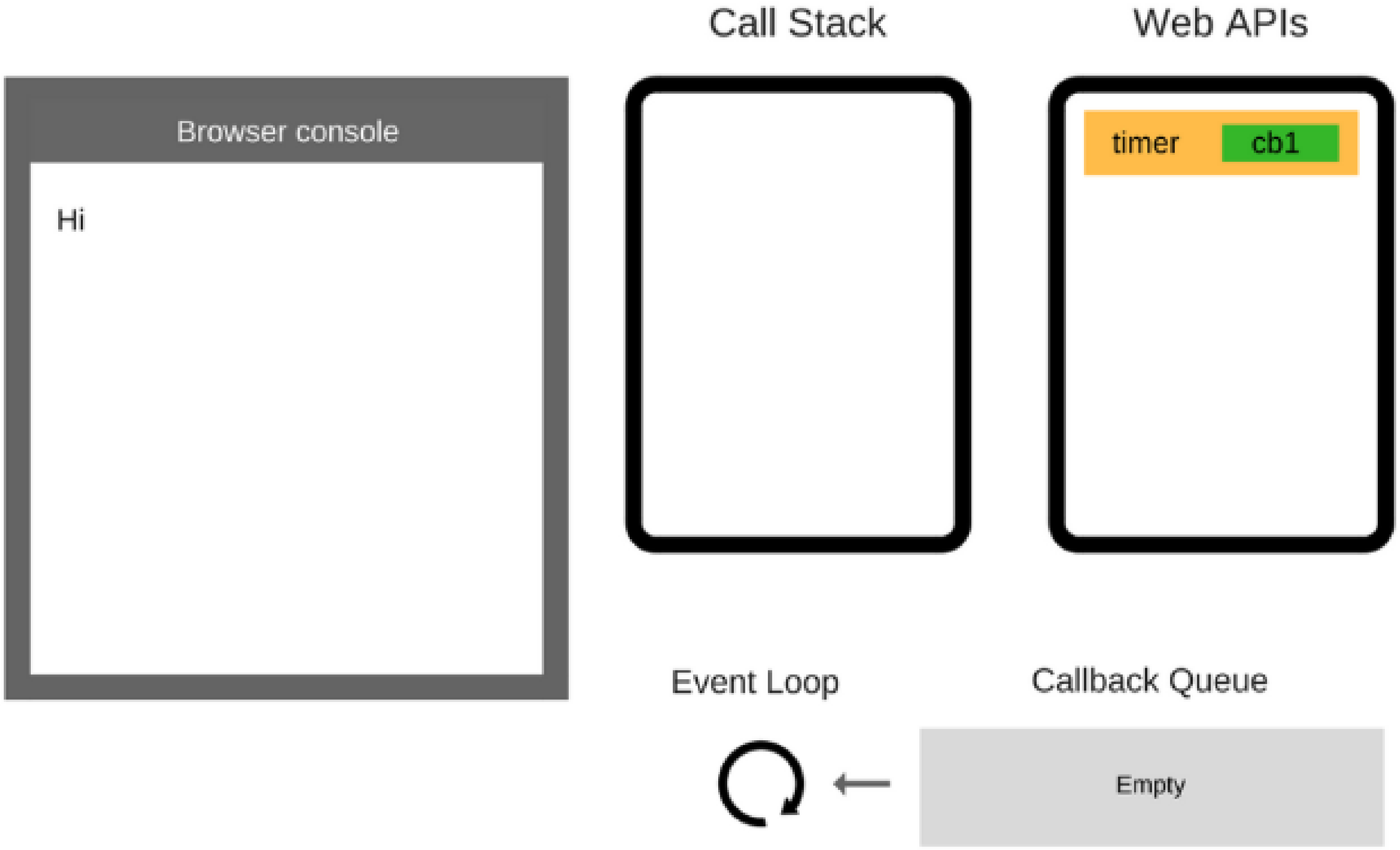
The API is called and given the callback function



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

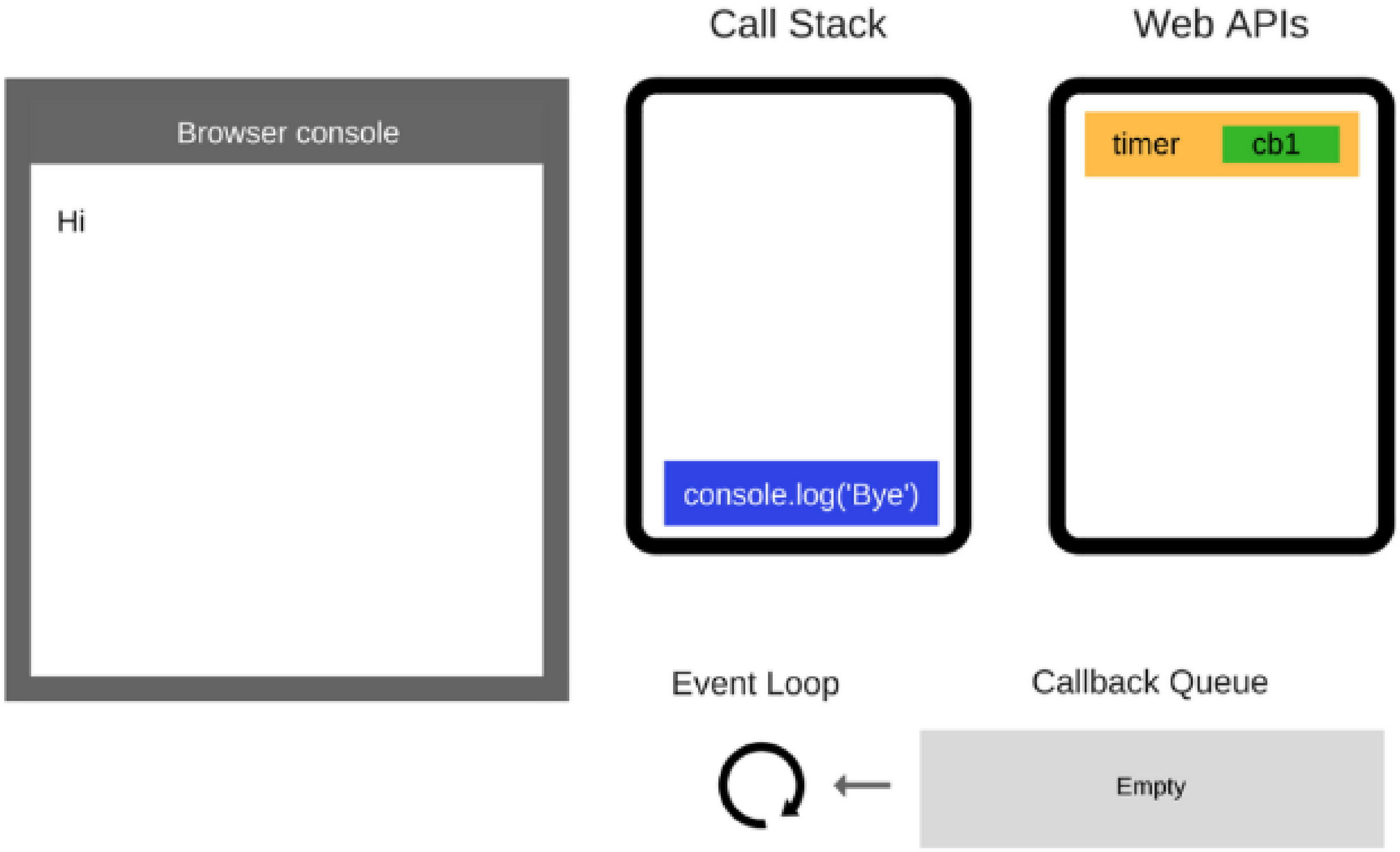
`setTimeout(…)` is completed and removed



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

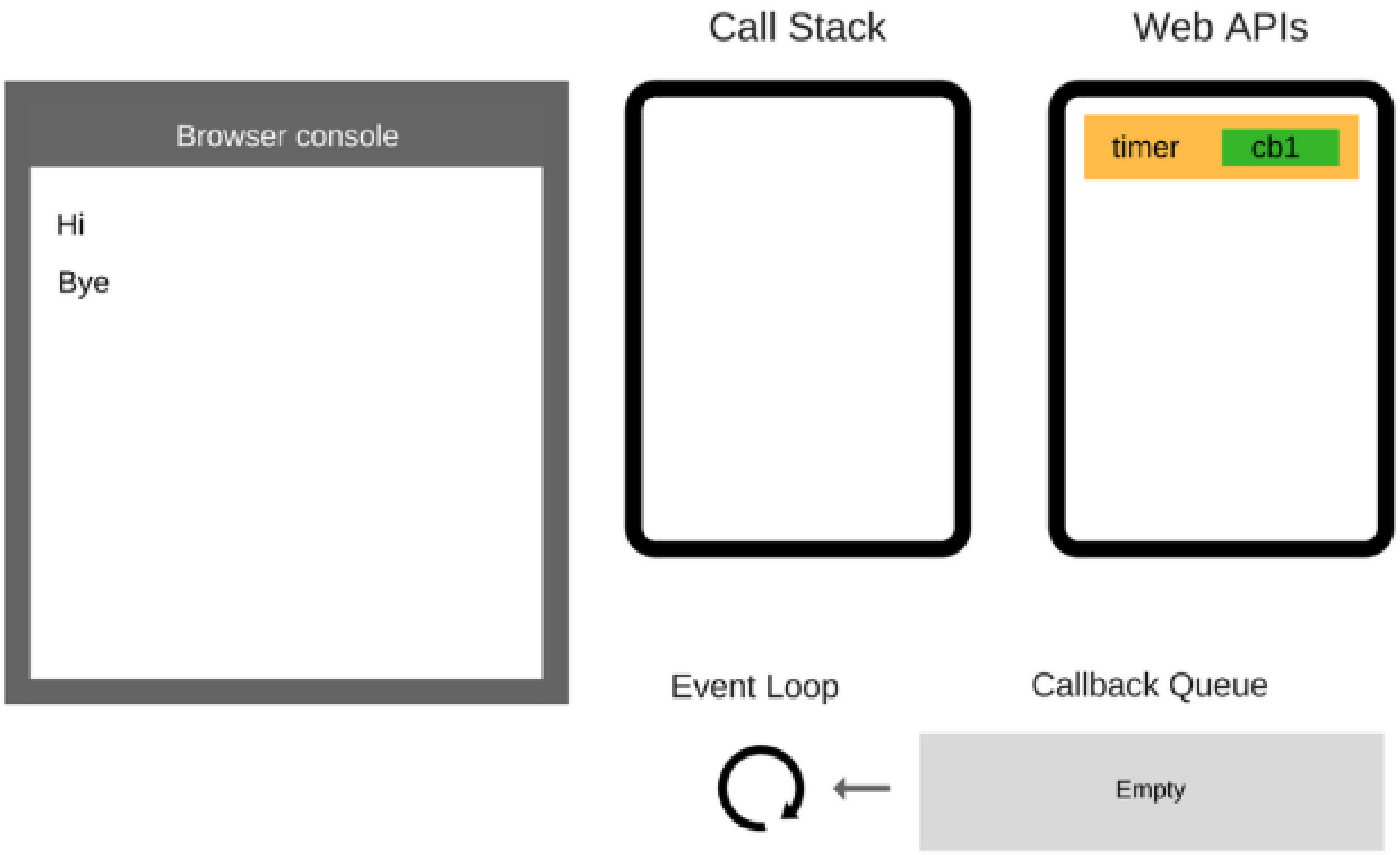
`console.log('Bye');` is added to the call stack



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

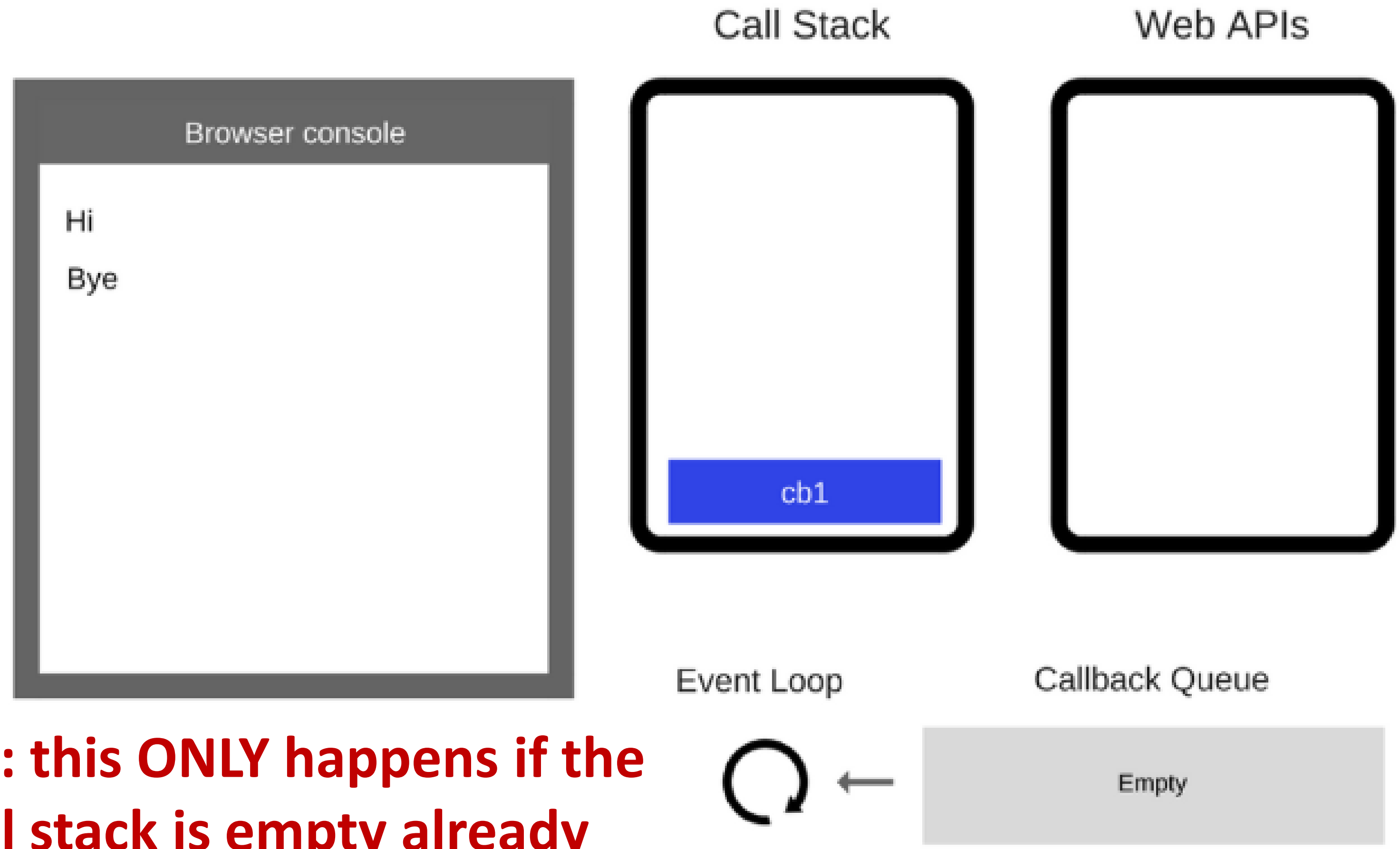
`console.log('Bye');` is executed/removed



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

The event loop takes cb1 and adds it to the call stack

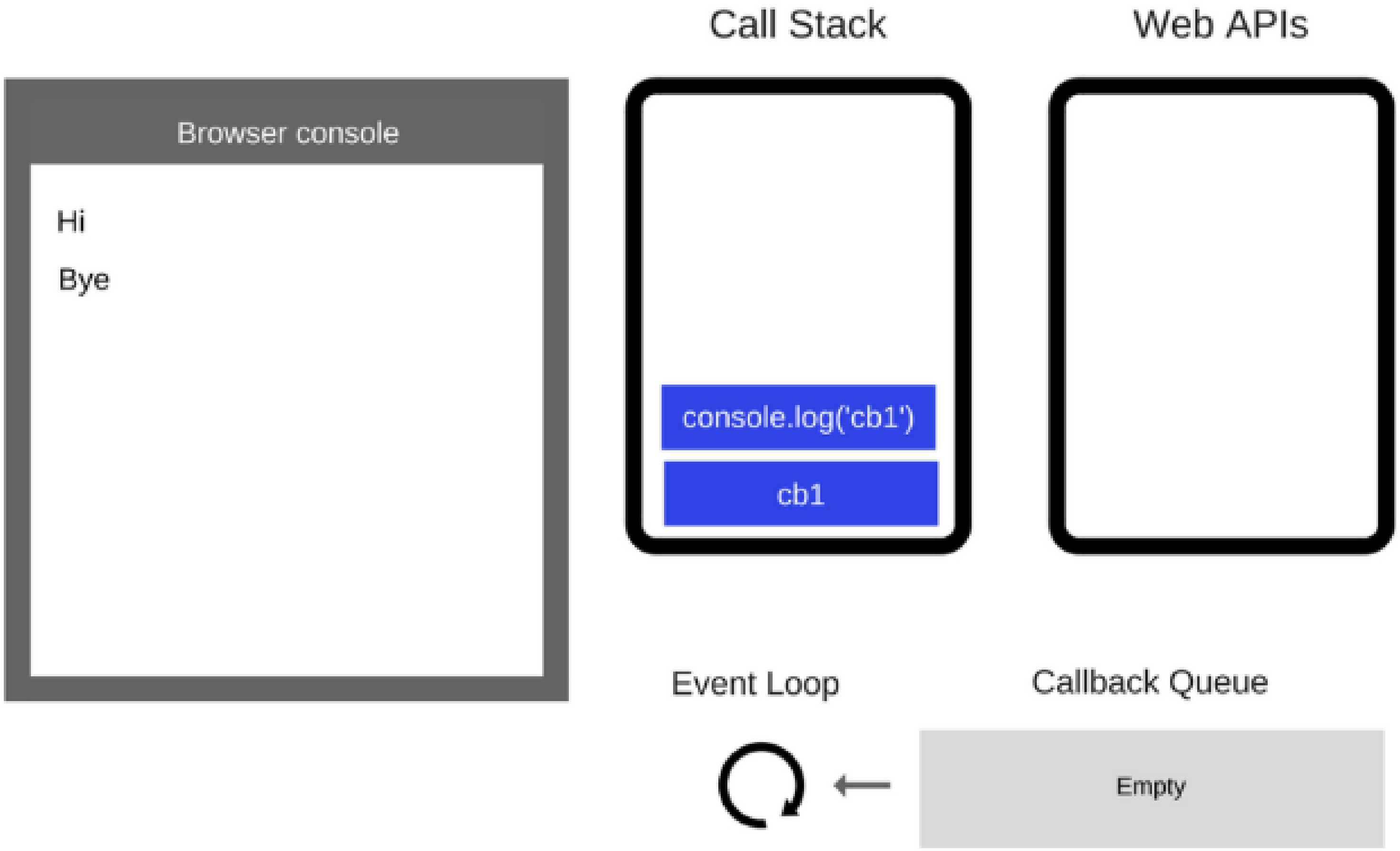


Note: this ONLY happens if the call stack is empty already

https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

The Event Loop

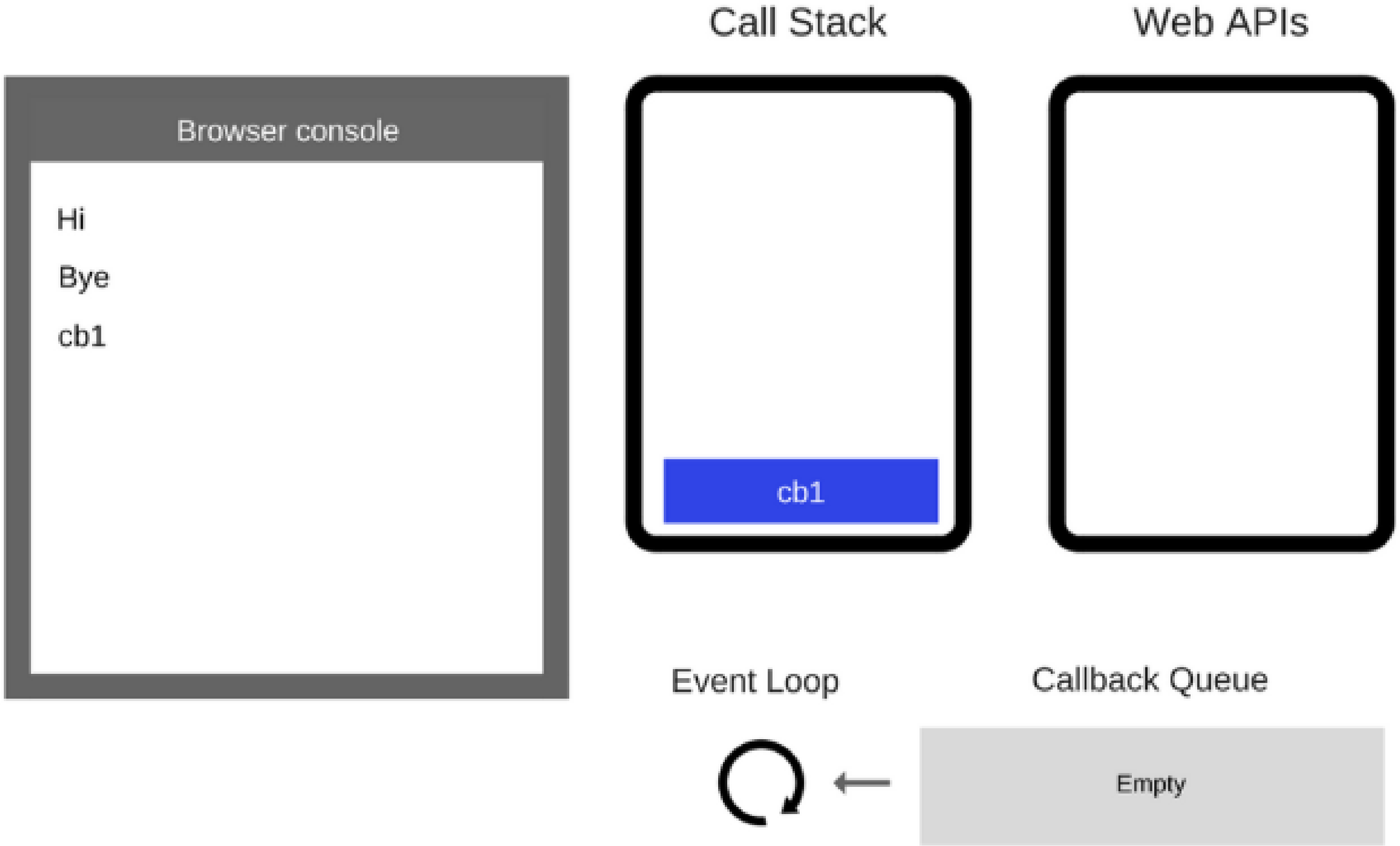
cb1 is executed and adds log to call stack



https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

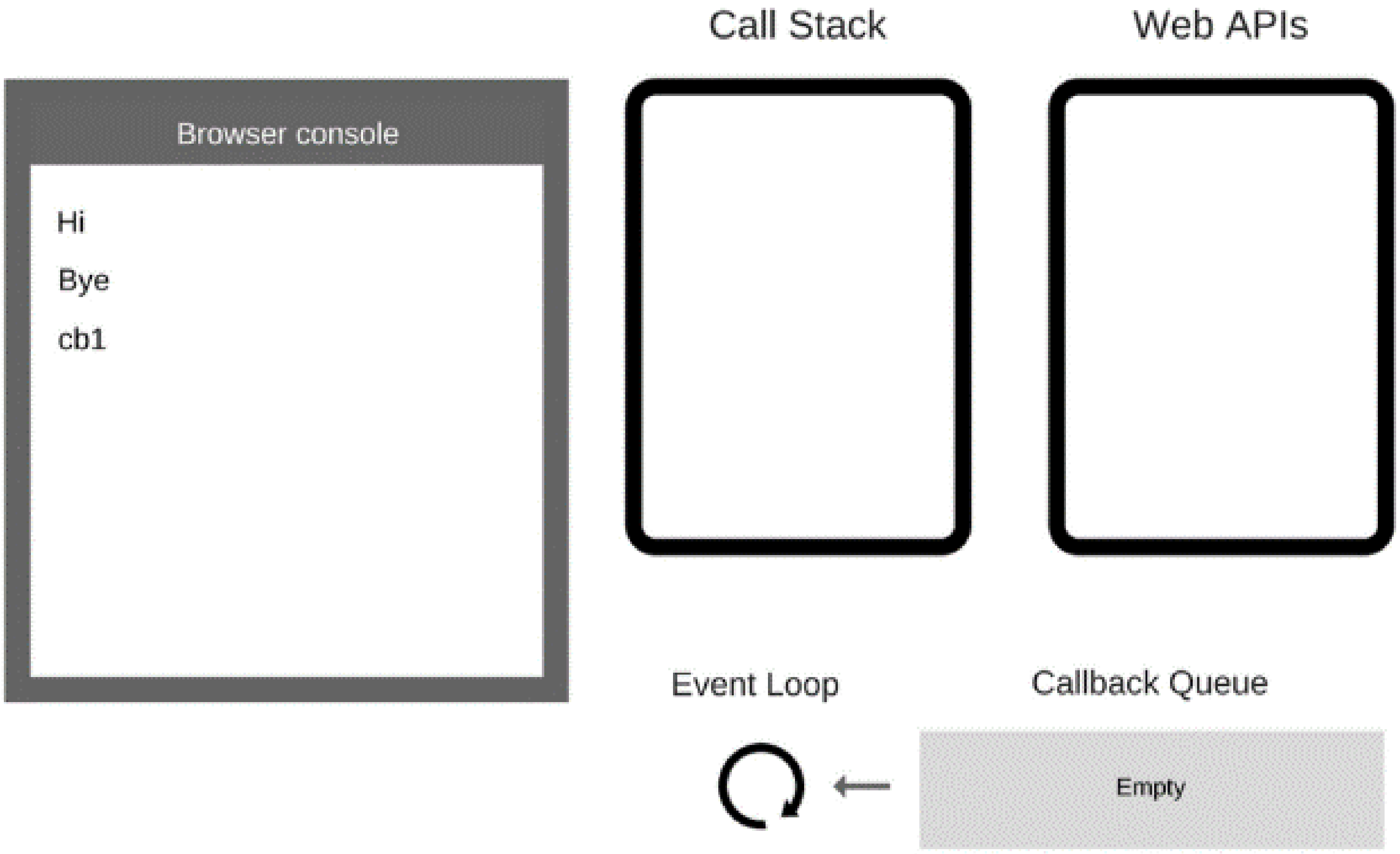
The Event Loop

`console.log('cb1')` is executed/removed



The Event Loop

cb1 has finished and is removed from call stack



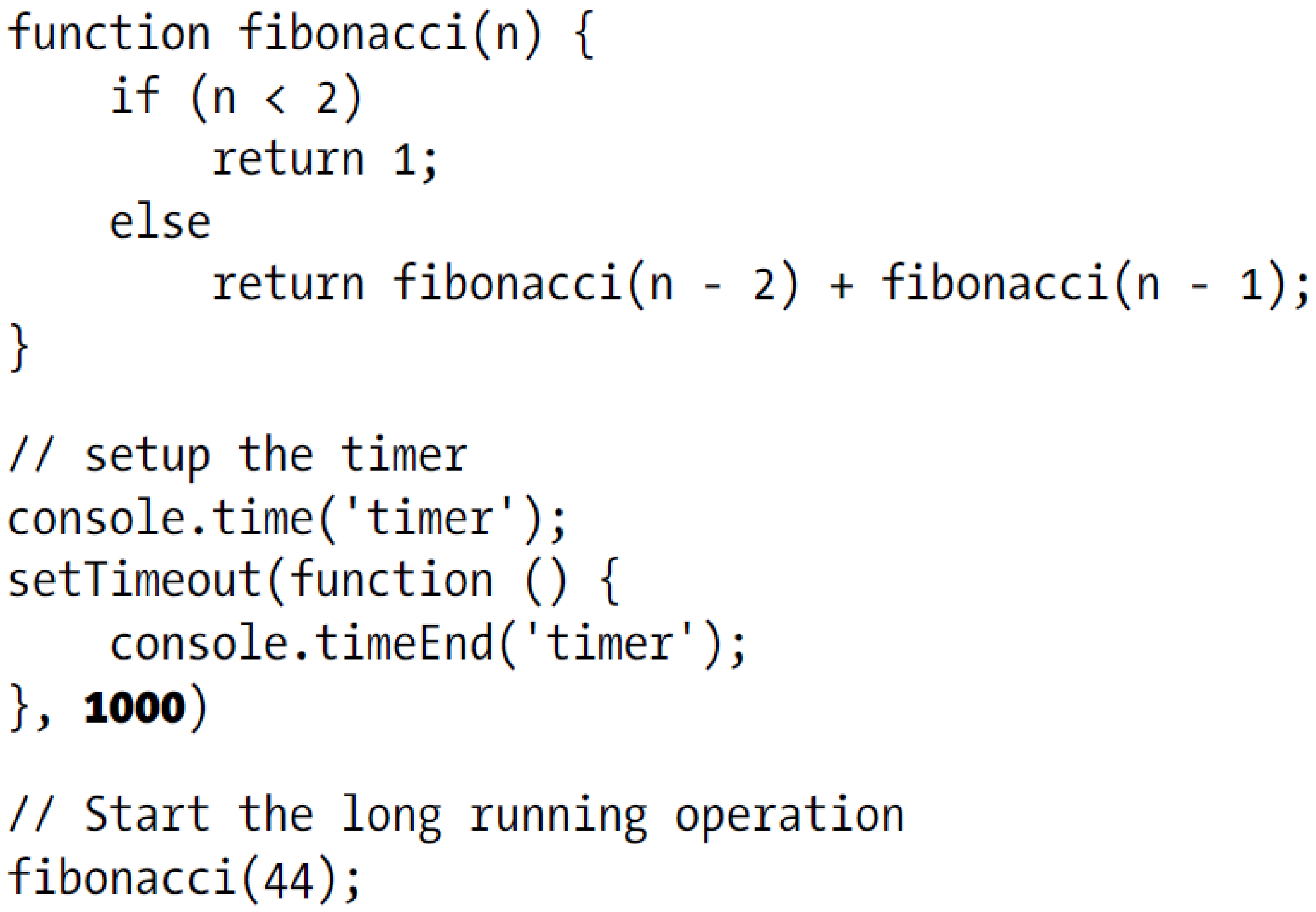
Browser console

```
Hi
Bye
cb1
```

https://blog.sessionstack.com/how-javascript-works-event-loop-and-the-rise-of-async-programming-5-ways-to-better-coding-with-2f077c4438b5

What Happens?

What happens when we run this code?



```
function fibonacci(n) {
    if (n < 2)
        return 1;
    else
        return fibonacci(n - 2) + fibonacci(n - 1);
}

// setup the timer
console.time('timer');
setTimeout(function () {
    console.timeEnd('timer');
}, 1000)

// Start the long running operation
fibonacci(44);
```

Syed, Basarat. *Beginning Node. js*. Apress, 2014, pp29.

Beware of Long-Running Operations

It is important to understand that the callback queue is only accessed when the call stack is empty

If you call a function that takes a long time to complete, all asynchronous operations that finish will not be handled until that function is finished

A good rule: anything that isn't asynchronous should complete 'immediately'

Node.js Module System

Another important aspect of the Node.js architecture to understand is the module system

Modules in Node.js can be:

1. Built-in modules (included)
2. File modules (local files)
3. External modules (from NPM)

Creating Your Own Module

Each file is considered to be its own module

Within a file, you have access to a *module* object

A property of interest within this object is *exports*

Creating Your Own Module

The *module.exports* object represents the values/functions/objects that this module exposes

Essentially, it is the object returned when you ‘require’ the module from somewhere else

For example, consider the example in 06-ex1-simple-module-exports.js

Creating Your Own Module

Modules can be a good way of organizing your code and using common functionality across different projects

So if it makes sense, you should separate your code into different modules

Requiring a Module

To include a module within another file, you use the *require* directive

When we specify a relative path as the argument for require, it tries to load a local module

What happens when Node.js executes:
let x = *require(“./06-ex1-simple-module-exports”);*

Requiring a Module

Node.js runs the specified file in a new scope and returns its *module.exports* object

Why is the ‘new scope’ important?

Requiring a Module

Node.js runs the specified file in a new scope and returns its *module.exports* object

Why is the 'new scope' important?

It eliminates the possibility of 'clobbering' existing definitions and does not pollute the global namespace

Requiring a Module

So after we call:
let X = ***require(“./06-ex1-simple-module-exports”);***

X points to the *module.exports* object defined in the specified file

Modules and Caching

An important note: require is a blocking (i.e., non-synchronous) function call

But, required modules are cached - subsequent requires of same module are loaded from memory

This has important implications if adding an object to module.exports – why?

Modules and Caching

Remember – object variables are references

So if we have *module.exports = { someKey : someVal};*

When we require this module in two different places, they will be pointing to the same object

Requiring Core Modules

To use a built-in module within node, you require it the same way

Just don’t specify a relative path:

require(‘somemod’);

Useful Core Modules

Some useful core modules to get started:

path – working with file path names

fs – reading/writing/manipulating files

http – works with the HTTP protocol
(e.g., to create a web server)

Consult the documentation for details…

External Modules

There are MANY external modules, organized through the Node Package Manager (NPM)

We will look at this in more detail later, once we start making use of external modules

Our First Server!

The *http* module has a *createServer(function)* method to easily create a web server

Function you pass the createServer function is a handler used to handle requests the server receives

Handler function needs a signature with two arguments representing request/response objects (again, documentation will be your friend!)

Our First Server!

Consider the code in 06-ex2-simple-server.js

This is a template for creating a basic server

Within the handler function, you can add any logic you want to handle requests and send responses

The Request Object

The default request object has useful properties:

request.method – the HTTP method of the request

request.url – the URL of the request

request.headers – an object containing all headers

We can use these to decide what to do with the request

Handling Get Requests

Remember “GET” requests do not have a body

Data we are interested in is included in the query string of the URL

For example:
http://localhost:3000/problems?arg1=1&other=2

This is in an easy to parse format for a reason…
(see 06-ex3-request-details.js)

Requests with Body Data

Some requests, however, do contain information in the HTTP request body (e.g., POST/PUT)

Extracting this data is not as straightforward

The request object our handler receives can be treated as a ‘ReadableStream’

(i.e., it implements the ReadableStream interface)

Reading Data from a ReadableStream

A ReadableStream has two important events we can create handlers for:

‘data’ – triggered when a new chunk of data is ready

‘end’ – triggered when there is no more data

We can add handlers to these to read in the entire request body (‘data’) and then handle it (‘end’)

Reading Data from a ReadableStream

So we can handle a plain text body like this:

```
let body = “”;
request.on('data', (chunk) => {
  body += chunk;
});
request.on('end', () => {
  // at this point, ‘body’ has the entire
  // request body stored in it as a string
});
```

See 06-ex4-extracting-body-data.js and 06-ex4-page.html

The Response Object

The response we send will have two components: the headers and the body

The response object has a number of properties and methods that we can use to set these values and send the response

The Response Object

We can set the status code of the response:

```
response.statusCode = 200;
```

The Response Object

We can set/remove header values:

```
response.setHeader(‘header-name’, ‘value’);

response.removeHeader(‘header-name’);
```

Setting the ‘Content-Type’ Header

One important header will be the ‘Content-Type’

This specifies the Multipurpose Internet Mail Extensions (MIME) type of the data

This gives the receiver information about how to process the data (e.g., JSON vs. HTML)

Setting the ‘Content-Type’ Header

The general structure of a MIME type is:

type/subtype;parameter=value

(parameters are optional)

For example:

text/html

application/json

text/plain;charset=UTF-8

Common MIME Types

Some common MIME types we will use:

```
const MIME_TYPES = { css: "text/css",
gif: "image/gif",
html: "text/html",
ico: "image/x-icon",
jpeg: "image/jpeg",
jpg: "image/jpeg",
js: "application/javascript",
json: "application/json",
png: "image/png",
svg: "image/svg+xml",
txt: "text/plain“ }
```

The Response Object

You can send the headers and status code manually:

```
response.writeHead(statusCode, {moreHeaders: value, ...});
```

There is an optional second part to include more headers that haven't already been set

The Response Object

If you don’t send the headers manually, they will be sent when the first of the following two occur:

response.write(data) – sends data to the requester

(useful for sending data as it is available)

response.end(*data*) – marks the end of the response

(can optionally be given data to include)

Extending the Simple Server

So we can:

1) Receive and parse requests
2) Read files locally (fs and path modules)
3) Send responses

We can combine these to create a server that serves static HTML content easily…
(e.g., 06-ex5-static-page-server.js)

Building a To-Do List Server

Consider the code in 06-todo-server.js. Add functionality to this file so the server can respond with the to-do list HTML/Javascript.

Building a To-Do List Server

We may want to share the list data among multiple clients. In this case, the server will act as a centralized store of the information.

Clients can request the list data (i.e., with a GET request) or request to change the list data (i.e., with a POST or PUT request)

Building a To-Do List Server

This involves at least three main steps on the server:

1. Create variable to store the list data on the server
2. Add route handler for GET requests to list URL (e.g., /list)
3. Add route handler for POST requests to list URL (e.g., /list)

Remember: you can JSON.stringify(obj) any object to send in response

Building a To-Do List Server

The client will also require some changes:

1. When new items are added to the list, use a POST request to send that new item to the server
2. Intermittently (e.g., every X seconds) make a GET request for the list data and update the page contents

These steps are facilitated by the XMLHttpRequest.

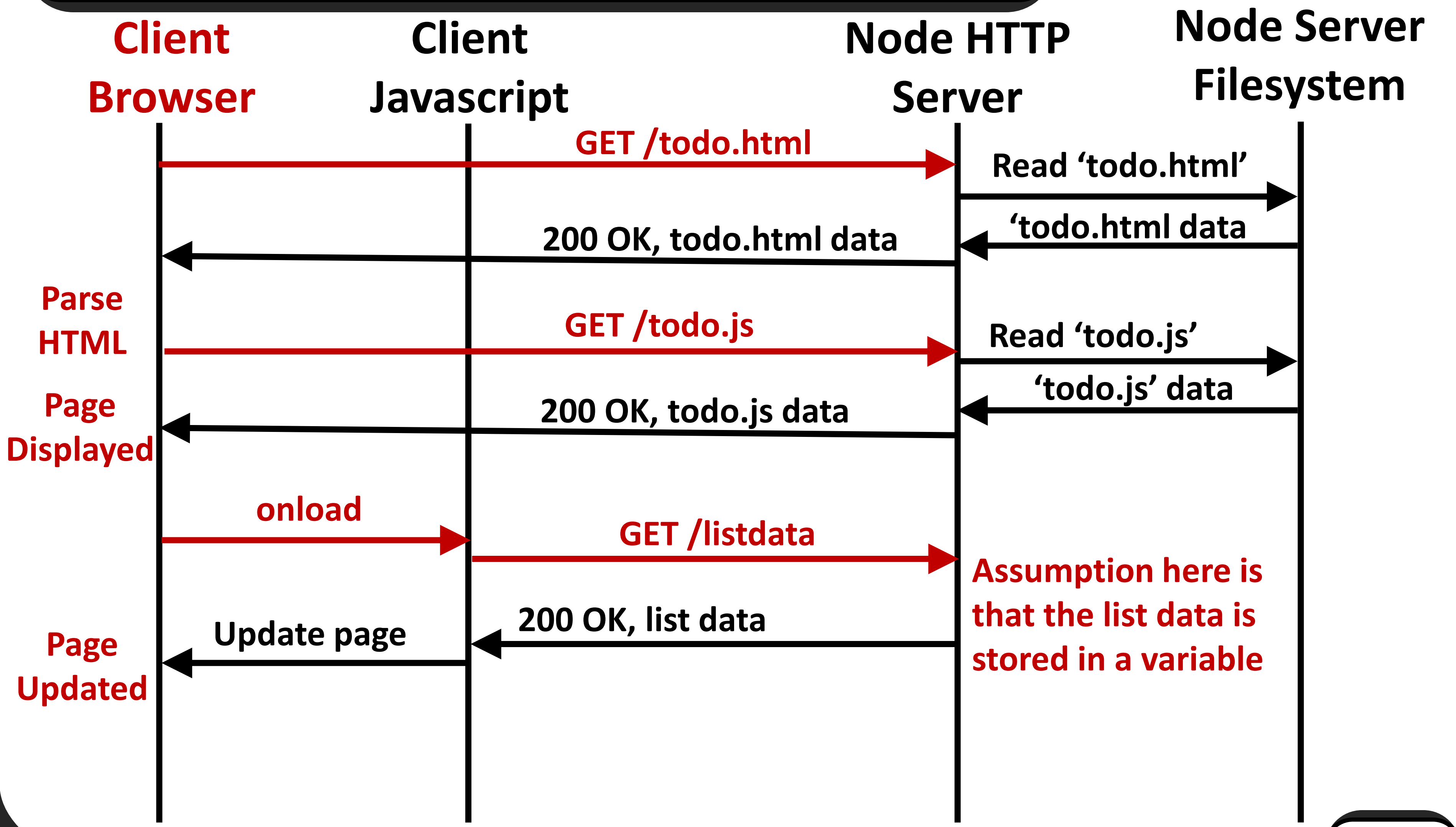
Building a To-Do List Server

If you have a good design, the client changes should be minimal

For example, if you are rendering the page contents from a single object, implement your server so it sends the same object structure to the client

This will be the essence of tutorial #4

The To-Do List Interactions



Client Browser

Client Javascript

Node HTTP Server

Node Server Filesystem

GET /todo.html

Read ‘todo.html’

‘todo.html data

200 OK, todo.html data

Parse HTML

GET /todo.js

Read ‘todo.js’

‘todo.js’ data

200 OK, todo.js data

Page Displayed

onload

GET /listdata

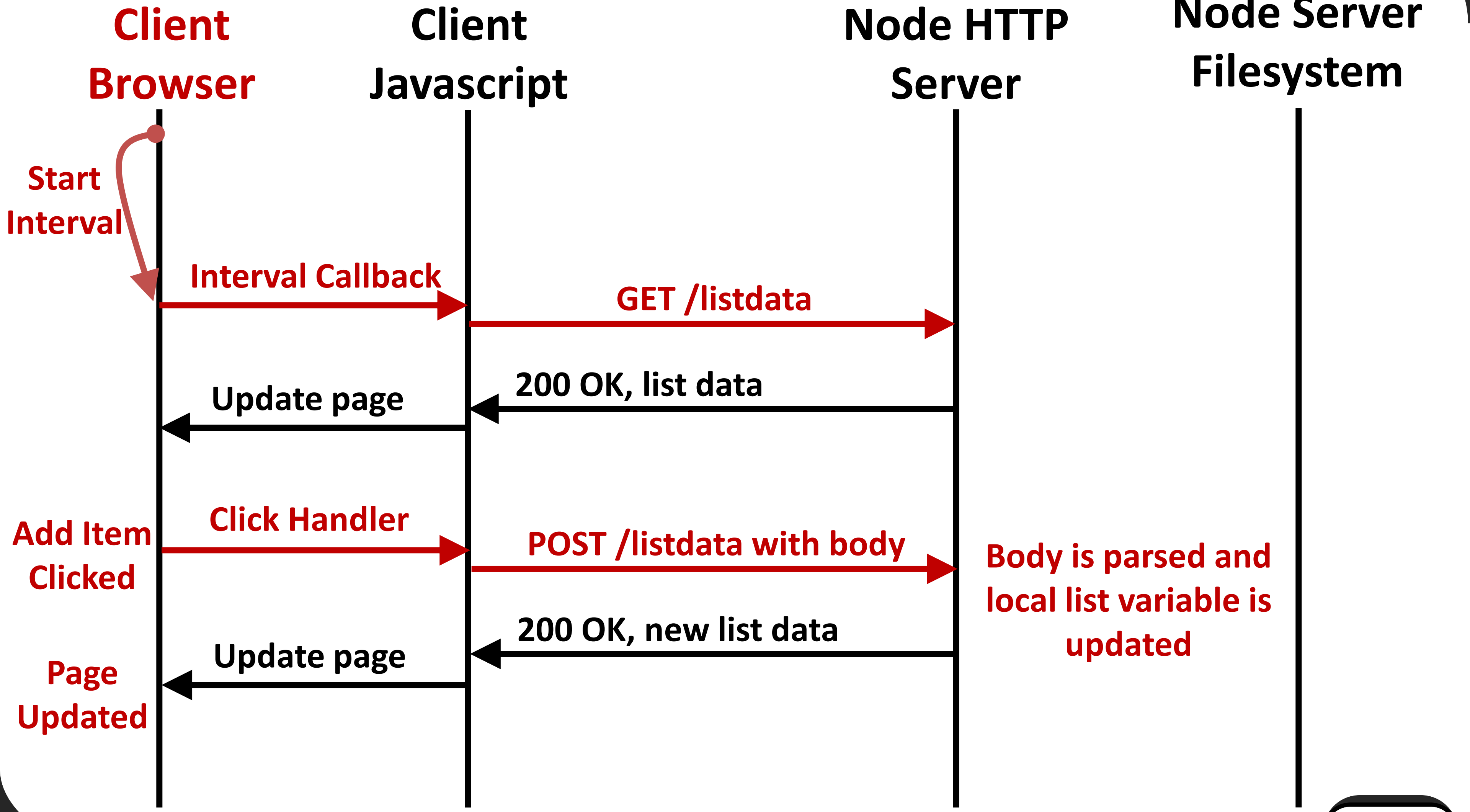
Assumption here is that the list data is stored in a variable

200 OK, list data

Update page

Page Updated

The To-Do List Interactions



Building a To-Do List Server

An important question: how will you indicate what operation you want to perform on the server

e.g., add an item, remove items, etc.

Different routes/URLs? Different HTTP methods?

Summary

We have a way of accepting requests, handling them, and sending responses

We can add as much logic as we need into the handler function to build a complex web system

This would get messy quickly and involve a lot of manual work on our part

Summary

Throughout the course, we will look at some ways to build these systems in a more efficient, scalable, and extensible way

But it is good to understand what is happening in the basic sense before we get into those details

Questions?

Social Network Example

The social network example is BAD coding, but it accomplishes something using tools we have covered (requests, responses, strings, JS objects)

One thing to consider because it is related to the next tutorial(s): what if we wanted the front page or messages section to update automatically?

How could we accomplish this with what we have available to us so far?