Software Engineering

A Faculty of Engineering Course: CSEN 303

Asynchronous Programming: The Concurrency Model

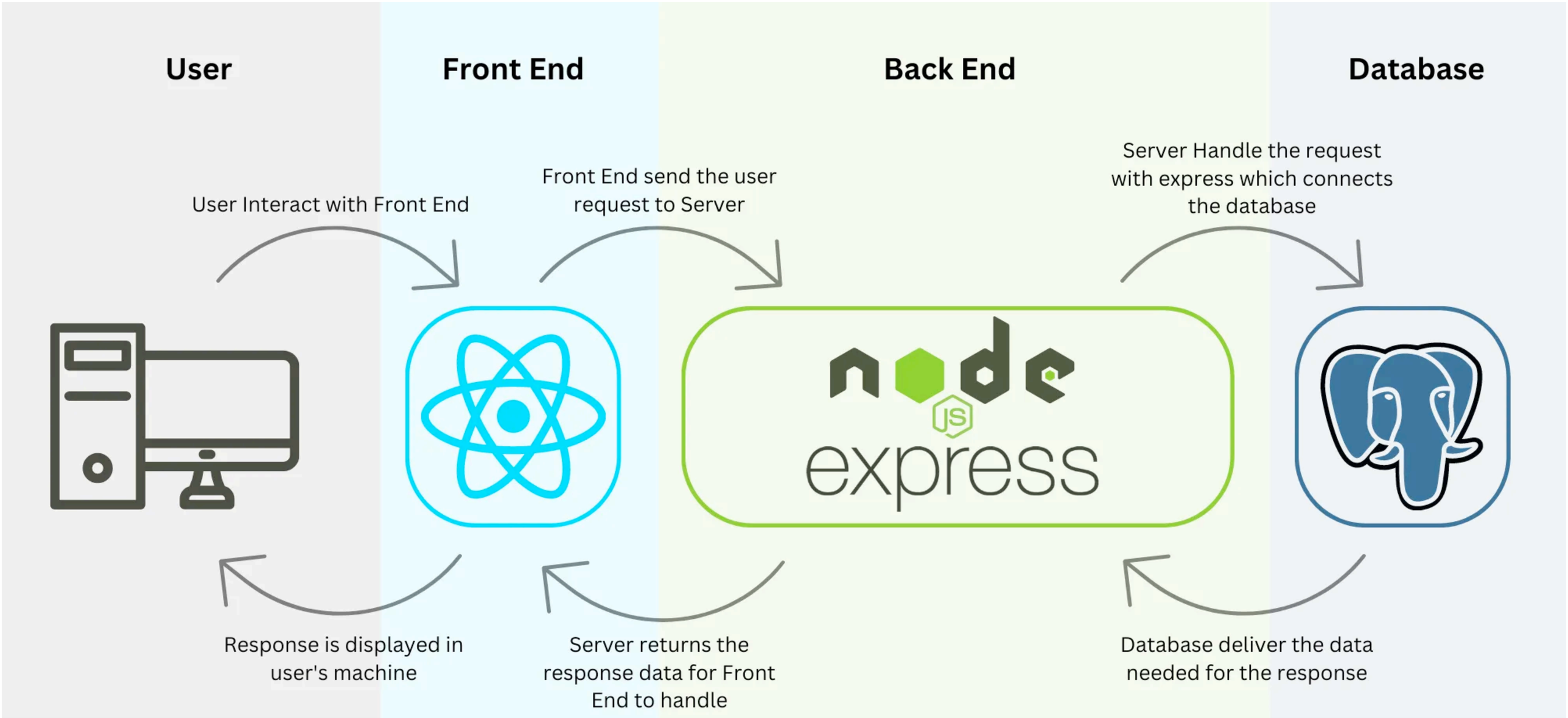
7

Dr. Iman Awaad

iman.awaad@giu-uni.de

GIU GERMAN INTERNATIONAL UNIVERSITY
الجامعة الألمانية الدولية

The PERN stack



https://medium.com/@snehadissanayaka56/exploring-the-pern-stack-build-robust-and-dynamic-web-applications-1bf04d6e5c82

Acknowledgments

The slides are **heavily** based on the **slides** by **Prof. Dr. John Zaki**.

His contribution is gratefully acknowledged.

Any additional sources are referenced.

The concurrency model

- Javascript
- Concurrency
- Processes and threads
- The JS event-loop model
- Call Stack
- Callback Queue

How does JavaScript deal with synchronisation… calls to databases, handling web requests, asynchronous processing, …?

Concurrency?

… in JavaScript, refers to the ability to **manage** and **handle** **multiple tasks** or **operations *seemingly* at the same time**.

How?

Through an **asynchronous**, **event-driven model** **rather than true parallelism** (simultaneous execution on multiple CPU cores)…

Concurrency?

… in JavaScript, refers to the ability to **manage** and **handle** **multiple tasks** or **operations** ***seemingly*** **at the same time**.

…provides the illusion of simultaneous execution

How?

Through an **asynchronous**, **event-driven model** **rather than true parallelism** (simultaneous execution on multiple CPU cores)…

What is **JavaScript** (**JS**)?

A programming language that runs in your web browser

Provides interactivity,
animation, RT content

…used mainly for
building **dynamic websites**

Many frameworks are based on JS:

Frontend: Angular, React, VueJS….

Backend: NodeJS

Desktop: Electron

Mobile: Ionic and React Native

Games: Phaser, Unity, ThreeJS

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

A high-level language

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

Machine language

No abstraction: 01010101000111

Low-level language

One level of abstraction: Assembly code

High-level language

C, Java, JS, Python, LISP,…

i.e. it provides a significant level of abstraction from the underlying computer hardware and its low-level operations…

Asynchronous

Javascript is a…

High-level

Asynchronous

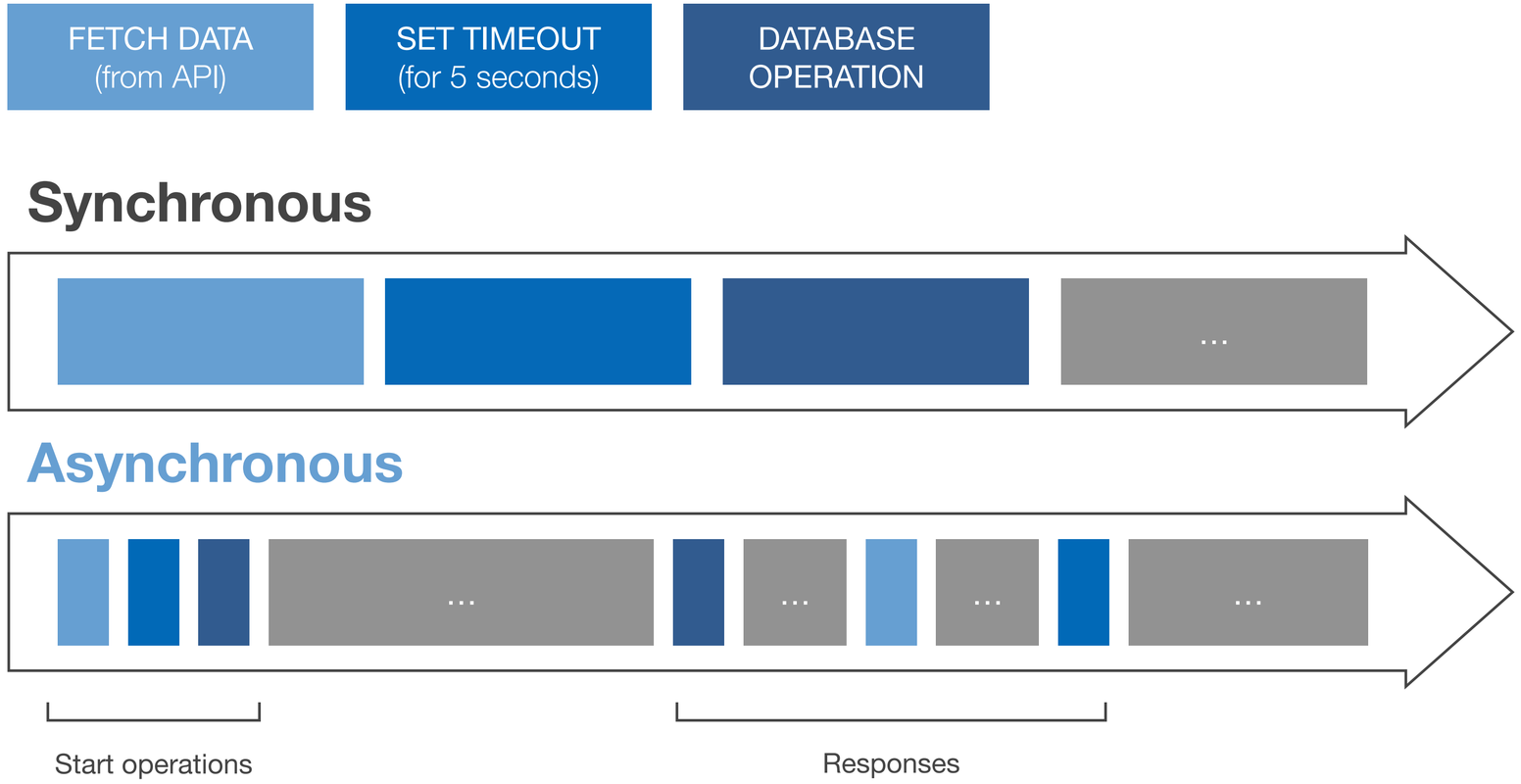
Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language



Asynchronousity is enabled by **callbacks**

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

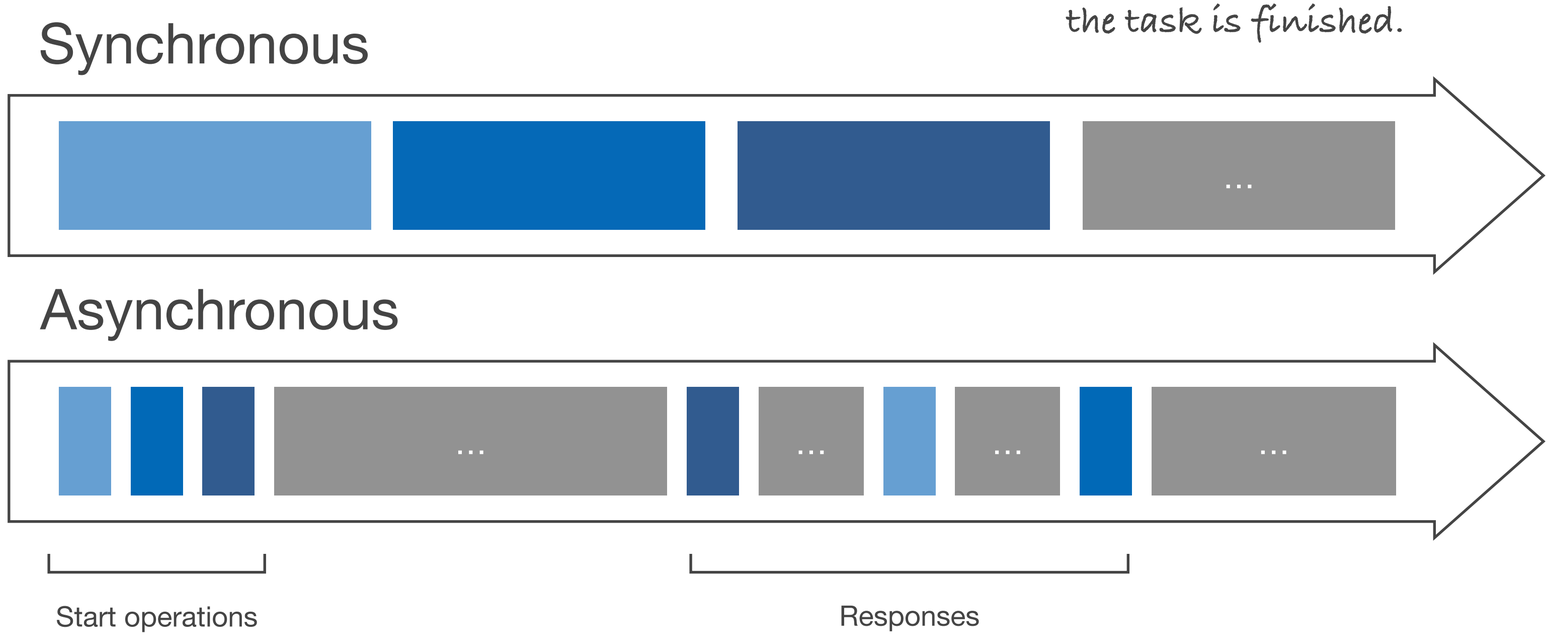
…are **functions** passed as arguments to **other functions**, to be executed later when an asynchronous operation **completes**: a foundational pattern for handling asynchronous events!

FETCH DATA (from API)

SET TIMEOUT (for 5 seconds)

DATABASE OPERATION

The function that receives the callback will "call back" the provided function when the task is finished.



Asynchronous: **Callbacks**…

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

…are **functions** passed as arguments to **other functions**, to be executed later when an asynchronous operation **completes**: a foundational pattern for handling asynchronous events!

JavaScript

```
function fetchData(callback) {
  // Simulate an asynchronous operation, like fetching data from an API
  setTimeout(() => {
    const data = "Some fetched data";
    callback(data); // Execute the callback with the data
  }, 2000); // Simulate a 2-second delay
}

function processData(data) {
  console.log("Processing:", data);
}

fetchData(processData); // Pass processData as the callback
console.log("Data fetching initiated...");
```

`fetchData` is an asynchronous function: doesn't immediately return the data;

instead, it starts a process that will eventually yield data.

`processData` is the callback function that `fetchData` will invoke once the data is ready.

The `console.log("Data fetching initiated...")` executes immediately, demonstrating the **non-blocking** nature of JS!

Non-blocking

Many operations, *e.g.* network requests, file I/O, or user interactions, can take **time to complete**…

Javascript is a…

High-level

Asynchronous

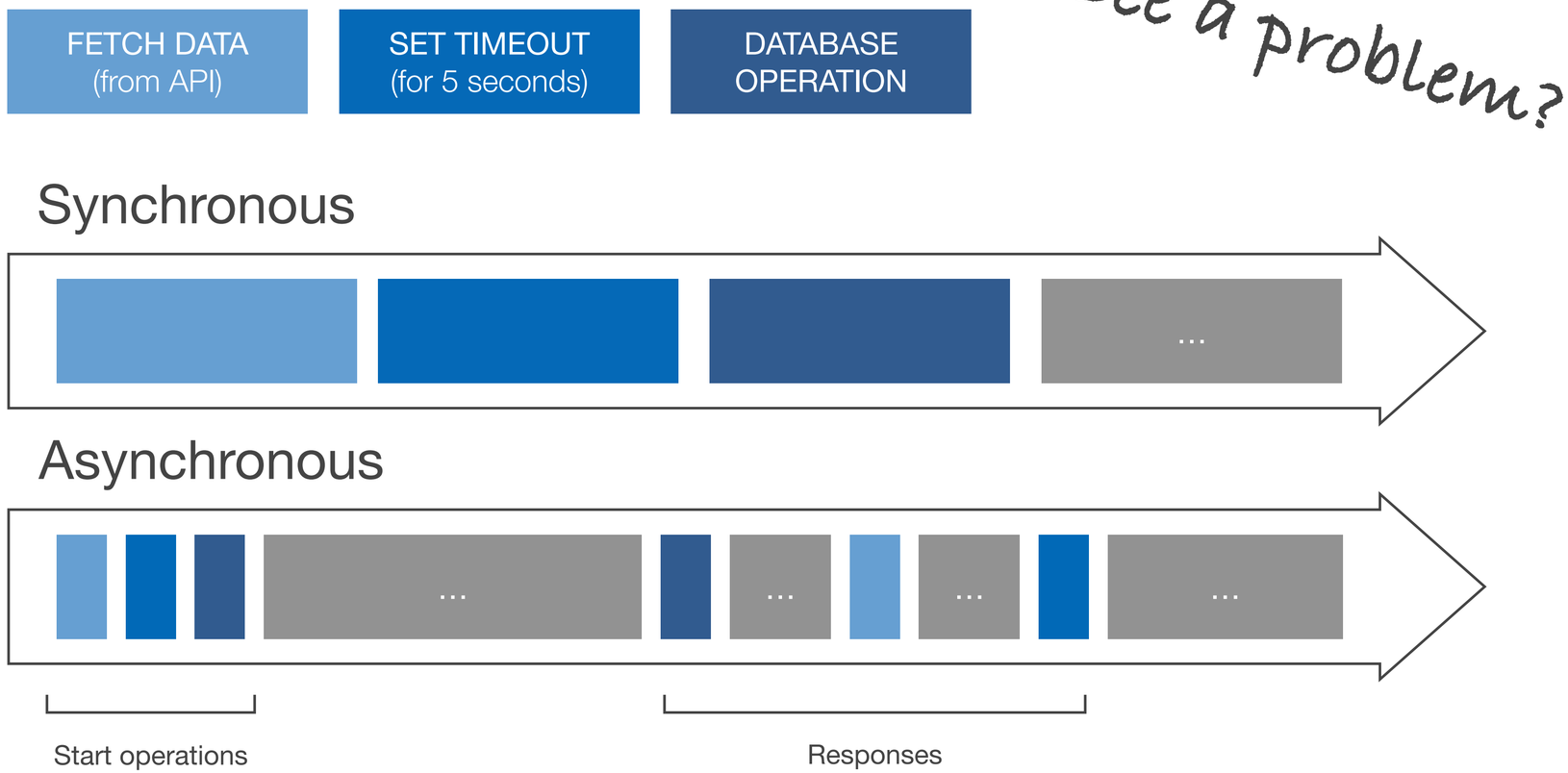
Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language



If these operations were **synchronous** (**blocking**), **the entire program would halt until they finished**, leading to a frozen user interface and poor responsiveness! UNLESS we could execute them **concurrently** or in **parallel**!

Non-blocking

Javascript is a…

High-level

Asynchronous

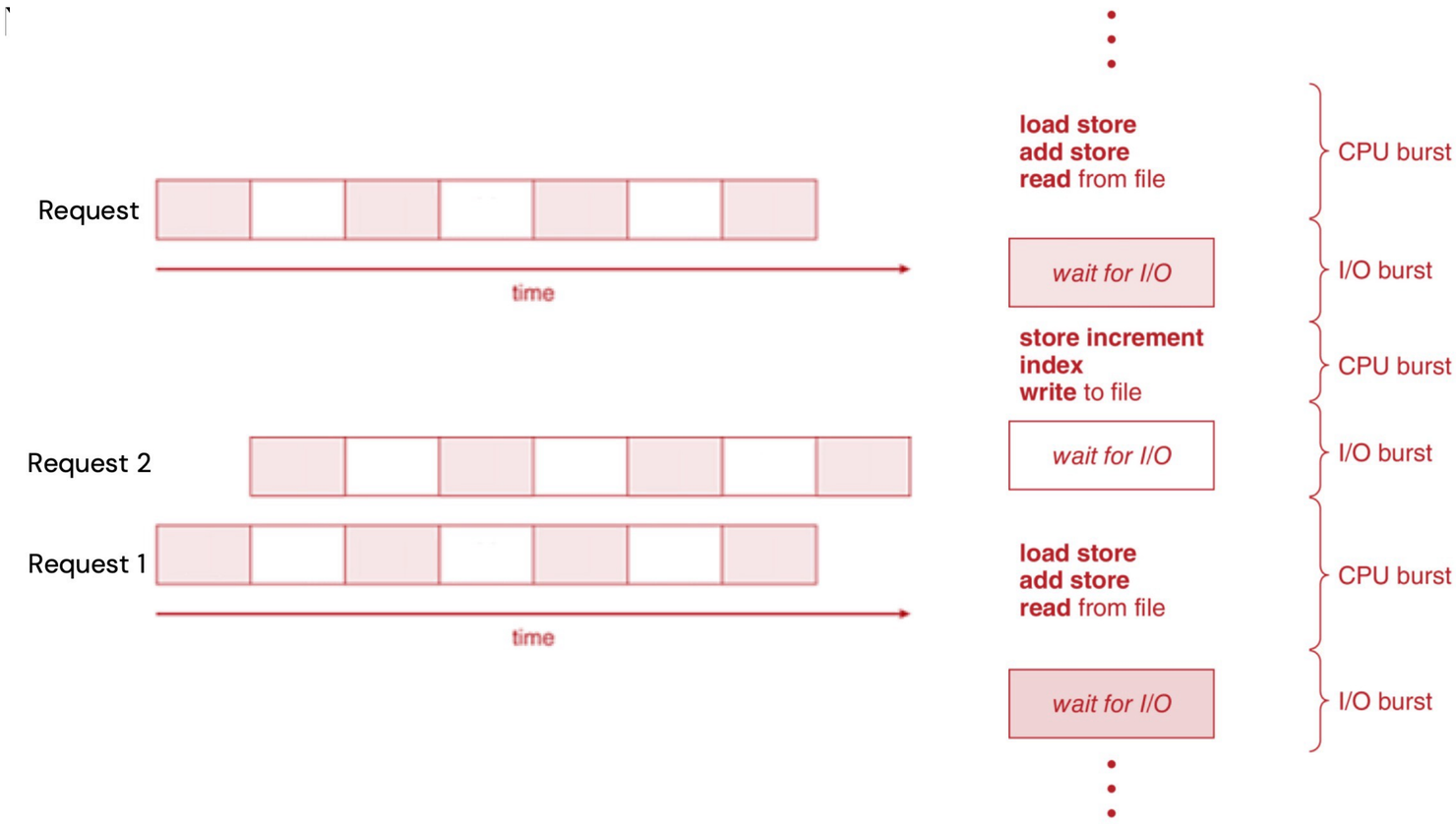
Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language



While the request is doing IO, the thread is not stuck waiting for the request to finish…

Non-blocking

Many operations, *e.g.* network requests, file I/O, or user interactions, can take **time to complete**…

Javascript is a…

High-level

Asynchronous

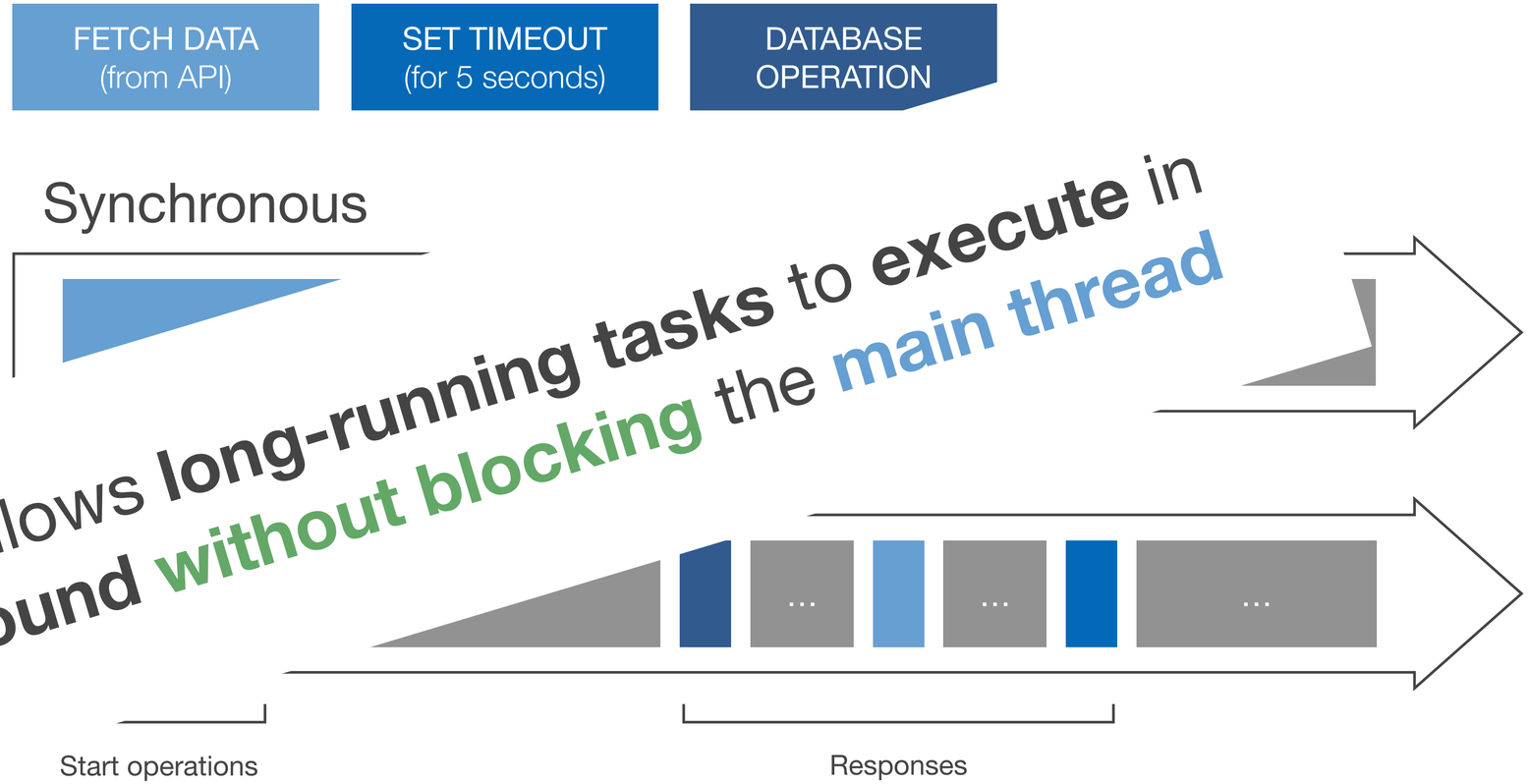
Non-blocking

Single-threade

Interpr

Conc

langu



If these operations were **synchronous** (**blocking**), **the entire program would halt until they finished**, leading to a frozen user interface and poor responsiveness! UNLESS we could execute them **concurrently** or in **parallel**!

Single-threaded

Javascript is a…

High-level

Asynchronous

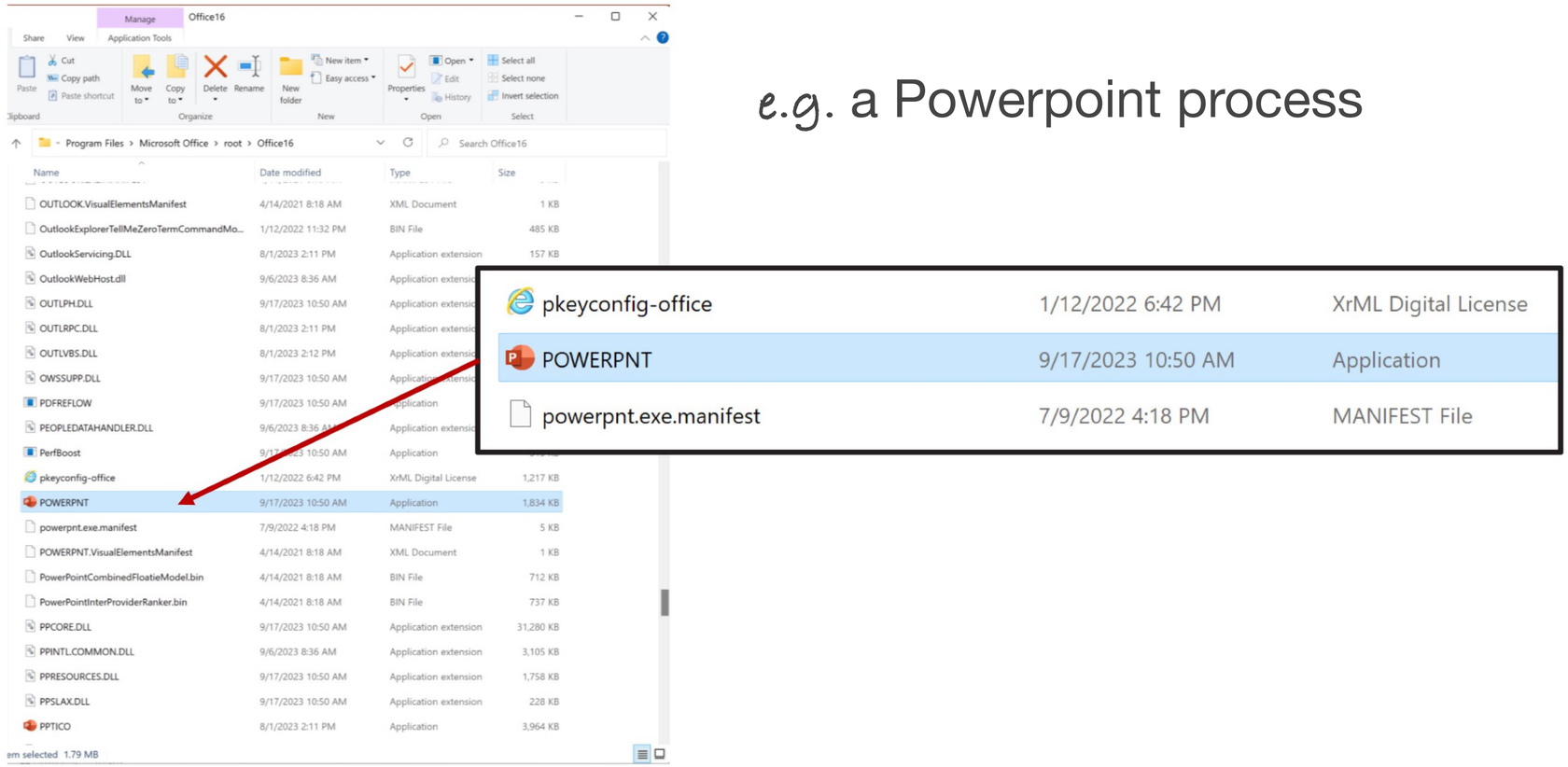
Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language



e.g. a Powerpoint process

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

How does it manage?

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

Process

An executing program loaded into memory

Each process has its **own distinct memory space and resources**, operating in isolation from other processes

Processes are considered "heavyweight" operations, meaning **they require more time and resources to create and terminate** (*i.e.* resource intensive)

Due to their isolation, **a failure in one process is unlikely to affect others** (*i.e.* fault tolerant)

Thread

The basic unit of execution **within a process** (*aka* “lightweight process”)

Threads belonging to the **same process share the same memory** address space and **resources**

This shared memory allows for **faster** and **more efficient communication** and data sharing between threads

Because they share resources, an error in one thread can potentially **crash the entire parent process**.

e.g. in a browser, one thread might load the page content while another handles scrolling and user input simultaneously…

i.e. a **process** is an entire program's environment,
while a **thread** is a specific sequence of execution within that environment…

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

How does it manage?

It does all these within a single process!

i.e. it executes one piece of code at a time on a single call stack….

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

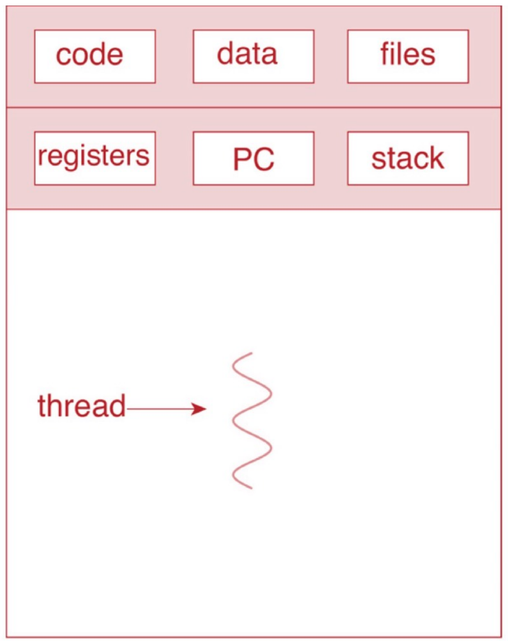
Single-threaded

Interpreted & JiT

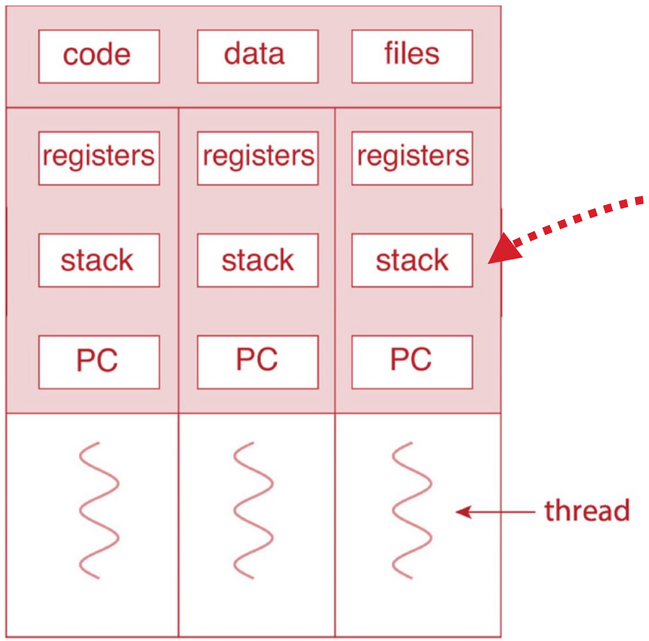
Concurrent

language

It does all these within a single process!



single-threaded process



multithreaded process

Thread is a smaller unit of execution **within a process**

Processes are **independent**, having their **own memory space**, while **threads within a single process share that memory**, enabling efficient communication but requiring careful synchronisation.

i.e. it executes one piece of code at a time on a single call stack….

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

Does **only** one thing at a time
Maintains a **call stack**: LIFO
Runs: executes one line at a time

```
1  //  Single Threaded
2
3  console.log("log1");
4  console.log("log2");
5  console.log("log3");
```

You know stacks…

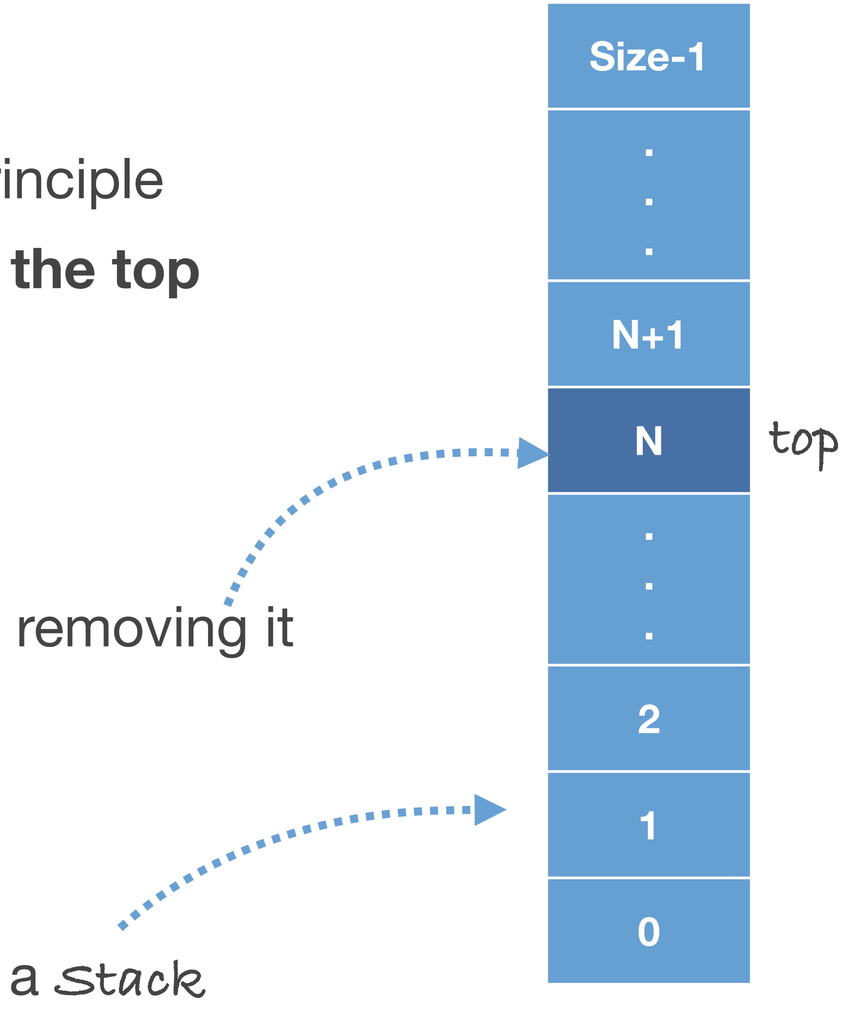
Hard to process

Easy to process

Credit: Bohdan Kalchenko

Stack operations

- Operates on the "**last-in-first-out**" (aka **LIFO**) principle
- Elements can be **added** and **removed only from the top**
- Manage the data using:
  - `push`: add a new piece of data to the stack
  - `pop`: remove a piece of data from the the stack
  - `top`: peek/look at the top-piece of data without removing it
  - `isEmpty`: check whether the stack is empty



Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

Does **only** one thing at a time

Maintains a **call stack**: LIFO

Runs: executes one line at a time

Things seem to be synchronous!

Fetching data, displaying documents,… take time!!!

What will happen?

```
1  //  Single Threaded
2
3  console.log("log1");
4  console.log("log2");
5  console.log("log3");
```

```
1  //  Single Threaded
2
3  console.log("log1");
4  setTimeout(() => {
5    console.log("log2");
6  }, 1000);
7  console.log("log3");
```

Single-threaded

Javascript is a…

High-level

Asy

Si

Inte

C urrent

language

Does **only**

…ying

… take time!!!

Single-threaded nature could lead to **blocking** issues where a **time-consuming task** **freezes** the whole application!!

What will happen?

Browsers have **Web APIs**
(**A**pplication **P**rogramming **I**nterfaces)

Executed within the
JS Runtime Environment

```
e Threaded

.log("log1");
console.log("log2");
console.log("log3");
```

```
1  //  Single Threaded
2
3  console.log("log1");
4  setTimeout(() => {
5    console.log("log2");
6  }, 1000);
7  console.log("log3");
```

Enabling concurrency

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

When an **asynchronous operation** is started, JavaScript **doesn't wait** for it to complete…

It **continues executing** the rest of the code…

Once the **asynchronous operation** **finishes**, it places its **result** (**or** an **error**) and the **associated callback function** into an **event queue**…

The JavaScript **event loop** then **picks up** these **callbacks** from the **queue** and **executes** **them** when the **call stack** is **empty**.

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

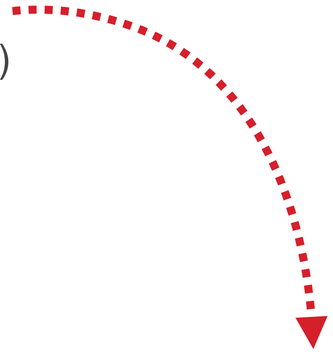
Interpreted & JiT

Concurrent

language

Browsers have **Web APIs**
(**Application Programming Interfaces**)

Executed within the
JS Runtime Environment



- Guarantees **availability** of the operations
- Guarantees **input-** and **output-types** of the operations

Interfaces exposed by web browsers that **allow JavaScript code to interact with various browser functionalities and system resources… NOT** part of the core JavaScript language but are provided by the browser environment, **extending** what JavaScript can do in a web application!

Examples of browser-provided functionality

Web APIs

Document Object Model (**DOM**) **API** allows JS to interact with the structure, style, and content of a web page *e.g.* `document.getElementById()`, `element.addEventListener()`

Fetch API provides a flexible way to make **network** requests *e.g.* fetching data from a server

Web Storage APIs (LocalStorage & SessionStorage) allow web applications to store data locally within the user's browser

Multimedia APIs for working with audio, video, and other media elements

Geolocation API enables web pages to request and use a user's geographical location

Notification API enables web applications to display system notifications to the user

Clipboard API provides access to the system clipboard

The JS Event Loop: Web APIs/Node.js APIs

…when a **long-running operation** (like `setTimeout` or a network fetch) is encountered, it's handed off to the **browser's** or **Node.js**'s underlying environment's **Web APIs**…

Single-threaded

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

```
> console.log("log1");
  console.log("log2");
  console.log("log3");
  log1
  log2
  log3
< undefined
```

```
> console.log("log1");
  setTimeout(() => {
  console.log("log2");}, 5000);
  console.log("log3");
  log1
  log3
< undefined
  log2
```

Set the time to 0 instead of 5 seconds and look at the output…

`setTimeout` & `setInterval`: Functions that **schedule the execution** of a **callback** after a specified delay or at regular intervals, respectively…**handled by the browser's** or **Node.js**'s **internal mechanisms** and placed in the **Callback Queue** when their time is due.

JavaScript Runtime Environment

Javascript is a…

High-level

Asynchronous

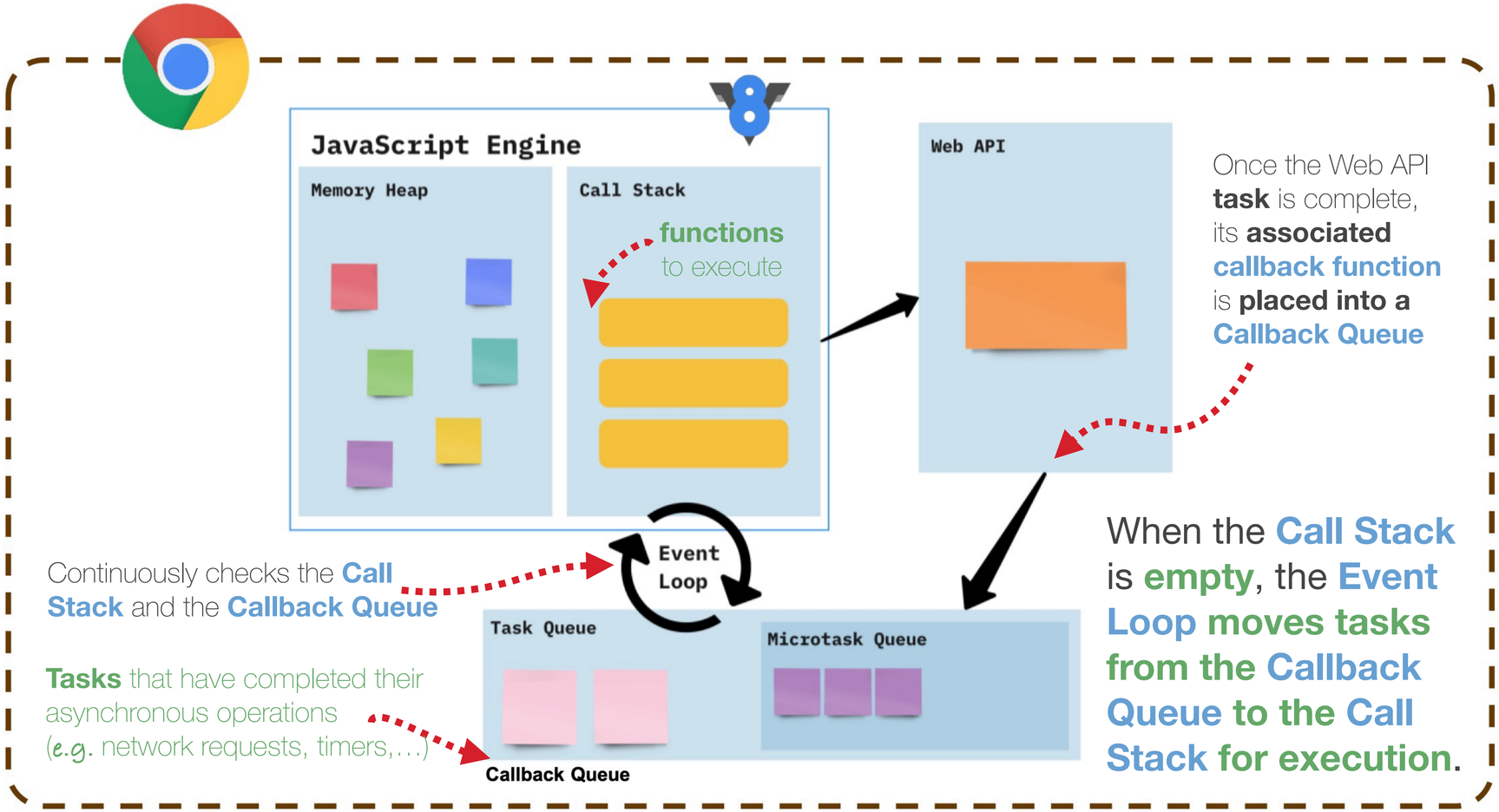
Non-blocking

Single-threaded

Interpreted & JiT

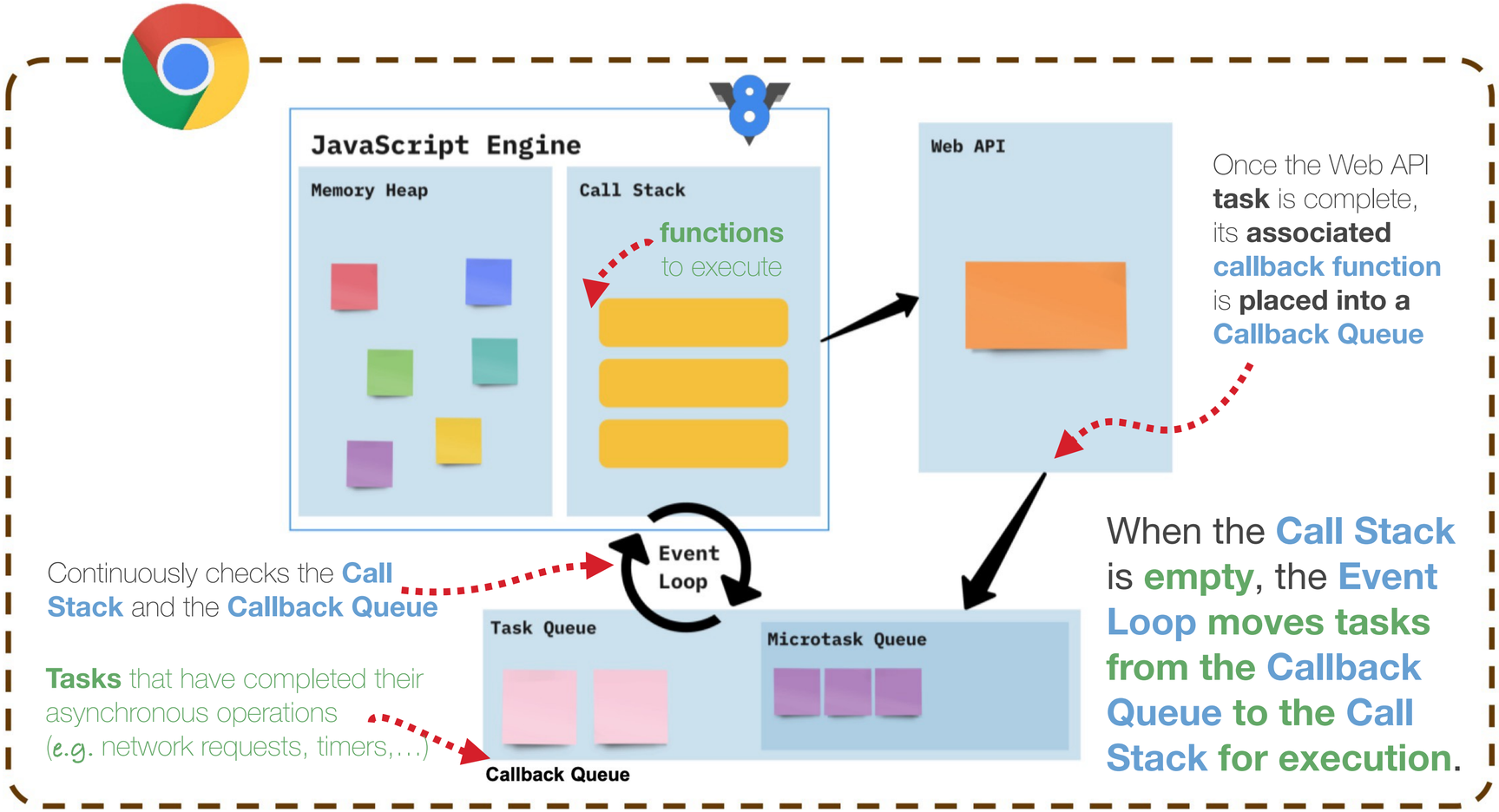
Concurrent

language



The JS Event Loop

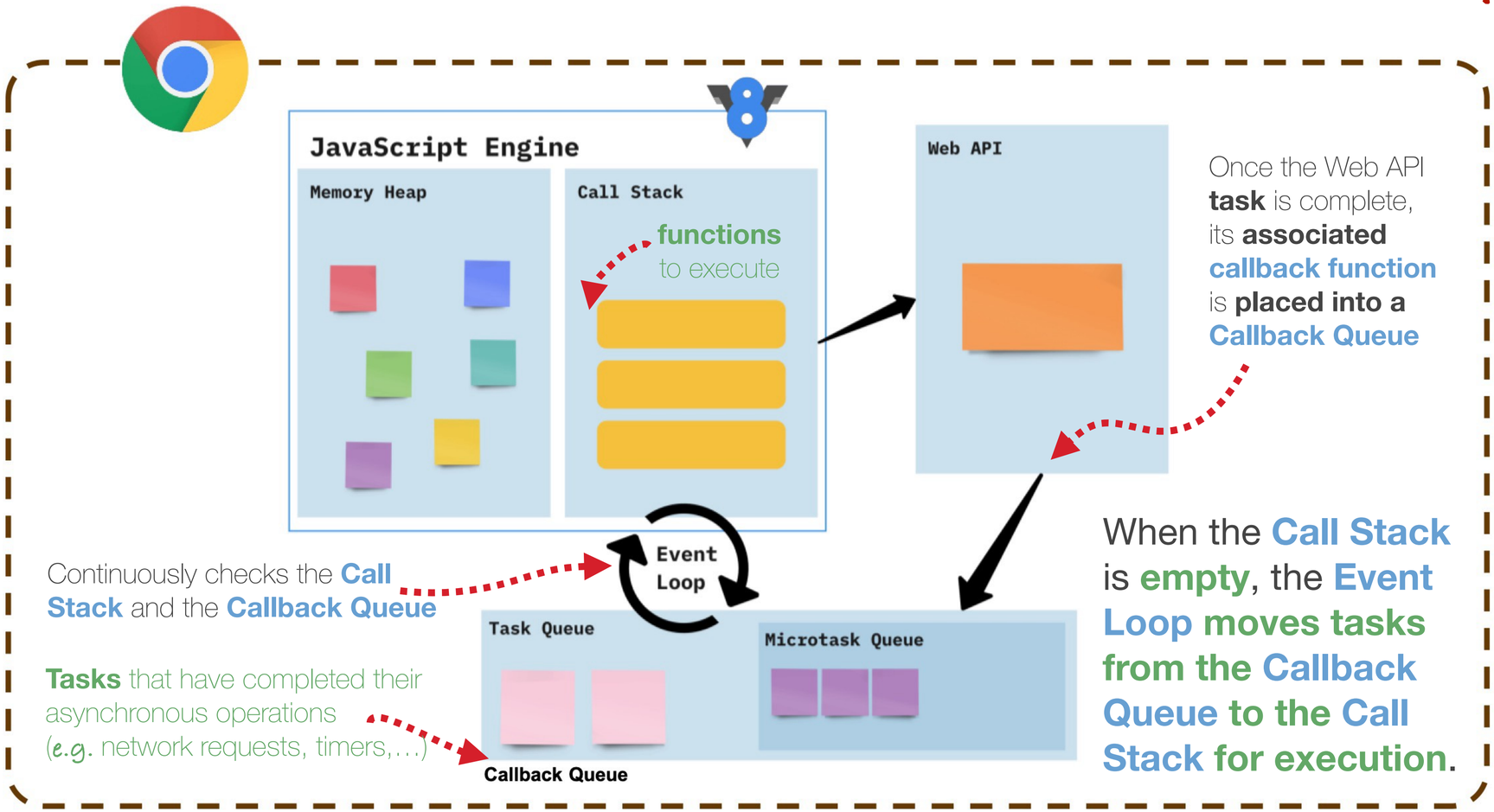
…is the **mechanism** that allows JS to **handle multiple tasks concurrently**



The JS Event Loop: Callbacks

Remember callbacks?

… are **functions** passed as arguments to **other functions**, to be executed later when an asynchronous operation **completes**:
a foundational pattern for handling asynchronous events!

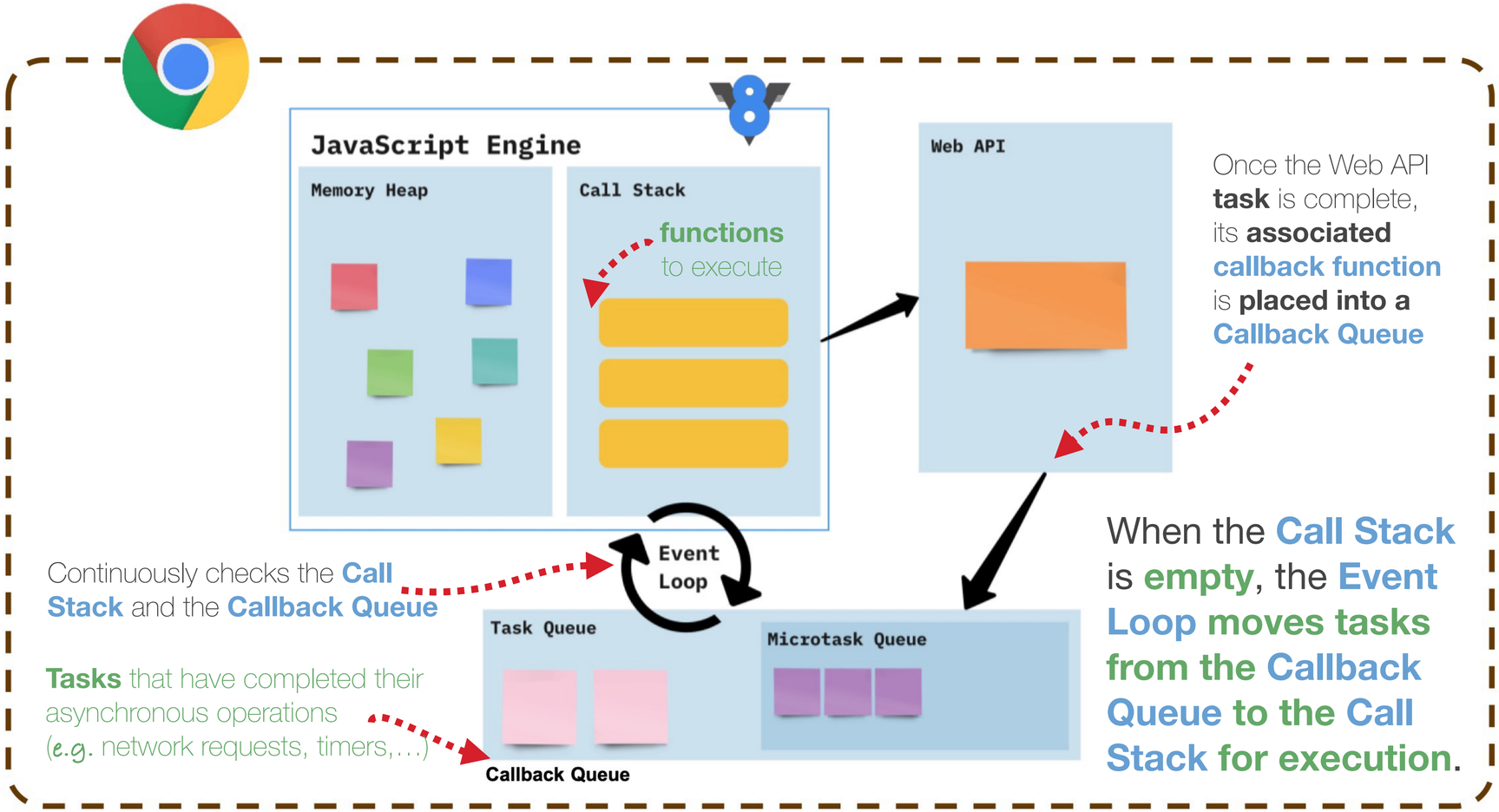


The function that receives the callback will "call back" the provided function when the task is finished.

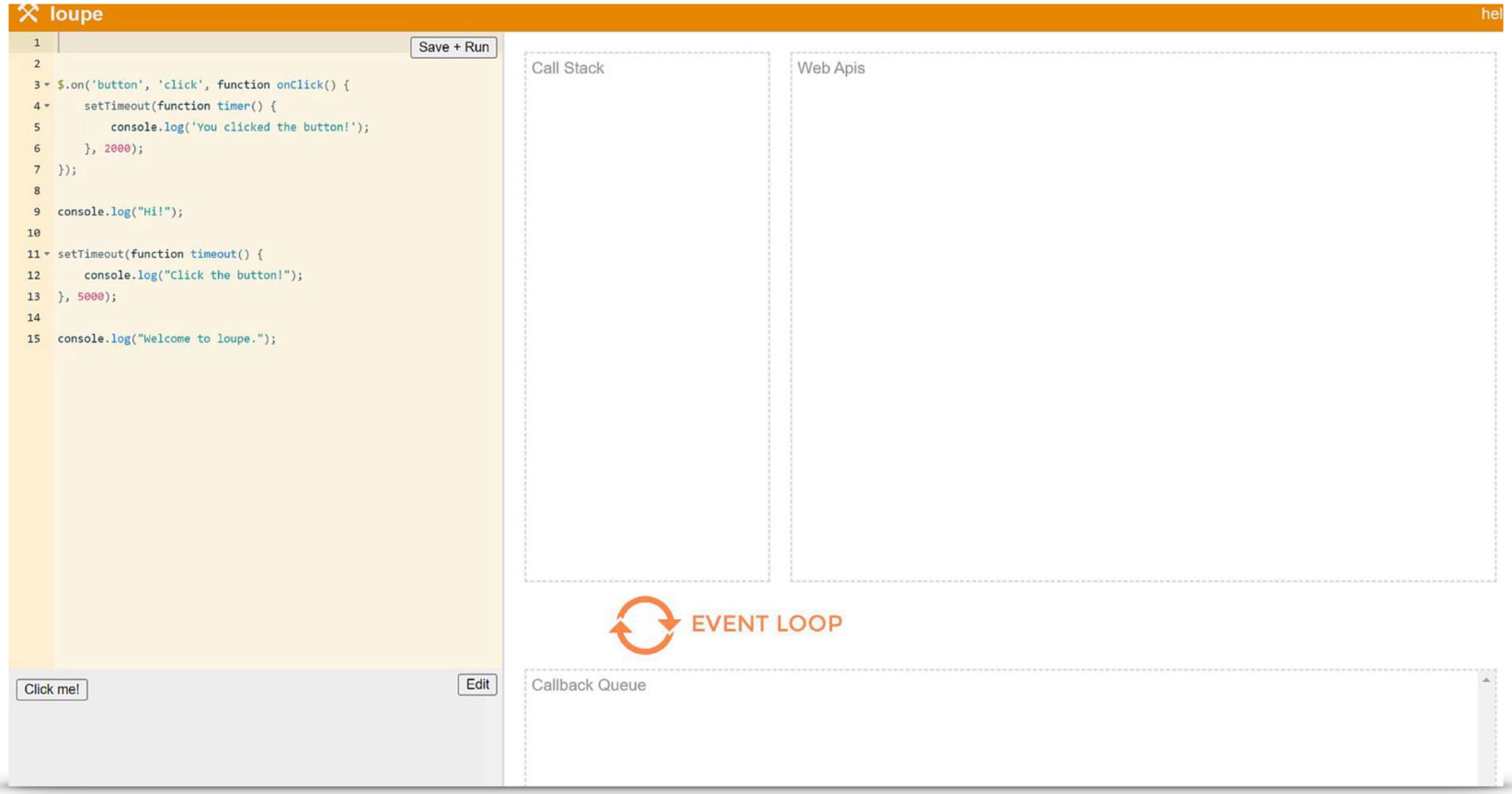
The JS Event Loop: **Promises**

…are **Objects** representing the **eventual completion** (**or failure**) of an **asynchronous operation** and its resulting **value**…

Provide a more structured and readable way to **handle asyn. code** compared to nested callbacks (*aka* callback hell)



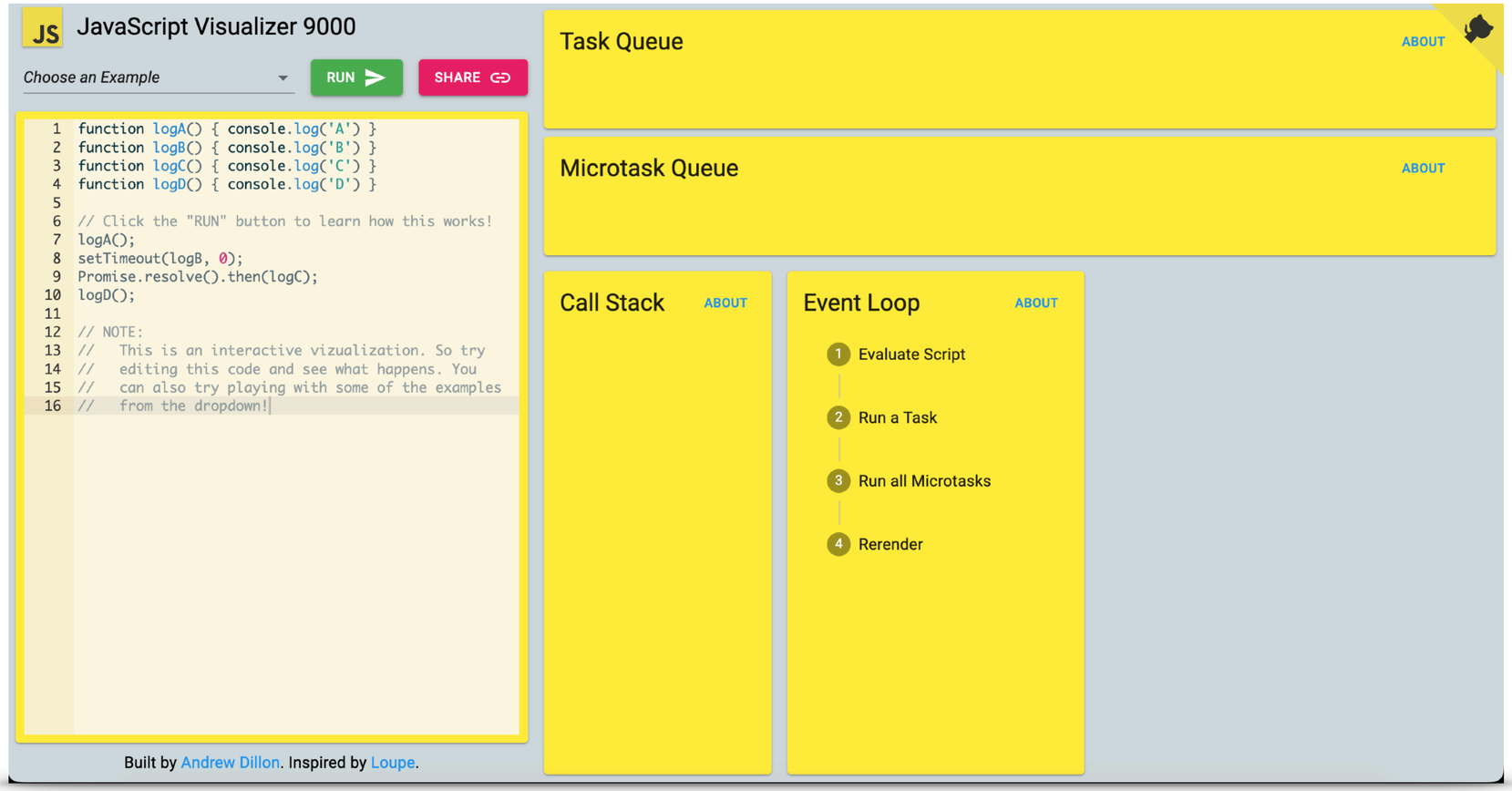
The JS Event Loop



"*Loupe* is a little visualisation to help you understand how *JavaScript*'s call stack/event loop/ callback queue interact with each other"

https://github.com/latentflip/loupe

The JS Event Loop



https://www.jsv9000.app/

The JS Event Loop: Callback Queue

i.e. a response to some action/event (*e.g.* Mouse click, receiving response to an HTTP request)

…holds **callbacks** from standard **asynchronous** operations *e.g.* `setTimeout()`, `setInterval()`, `setImmediate()` (in Node.js), and I/O operations, *e.g.* network requests, file system operations…

Once a `setTimeout` timer expires, or an I/O operation completes, its associated **callback function** is **moved** from the **Web APIs** (or Node.js C++ APIs) environment to the **Callback Queue**

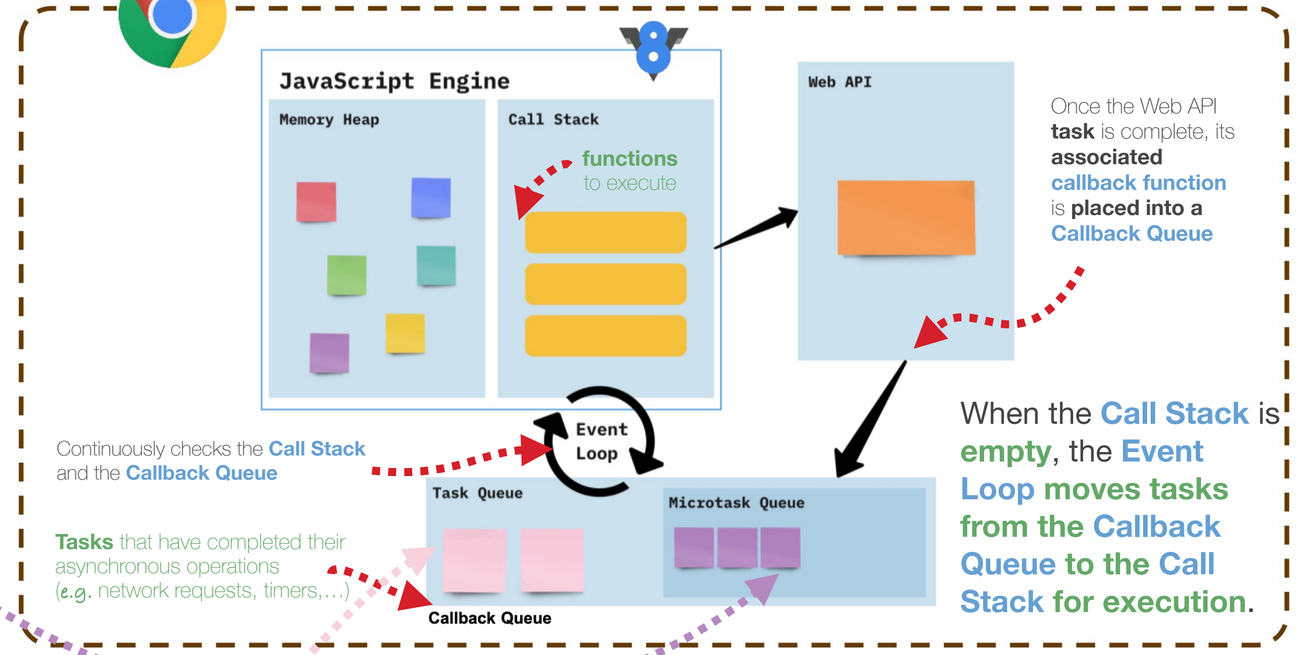
The Event Loop processes tasks from the Callback Queue one by one, only after the **Microtask Queue** has been completely emptied in the current iteration!

serve as a response to some action/event
Mouse click
Receiving response to an HTTP request

aka Macrotask Queue / Task Queue

```
> console.log("log1");
  console.log("log2");
  console.log("log3");
log1
log2
log3
undefined
```

```
> console.log("log1");
  setTimeout(() => {
  console.log("log2");}, 5000);
  console.log("log3");
log1
log3
undefined
log2
```

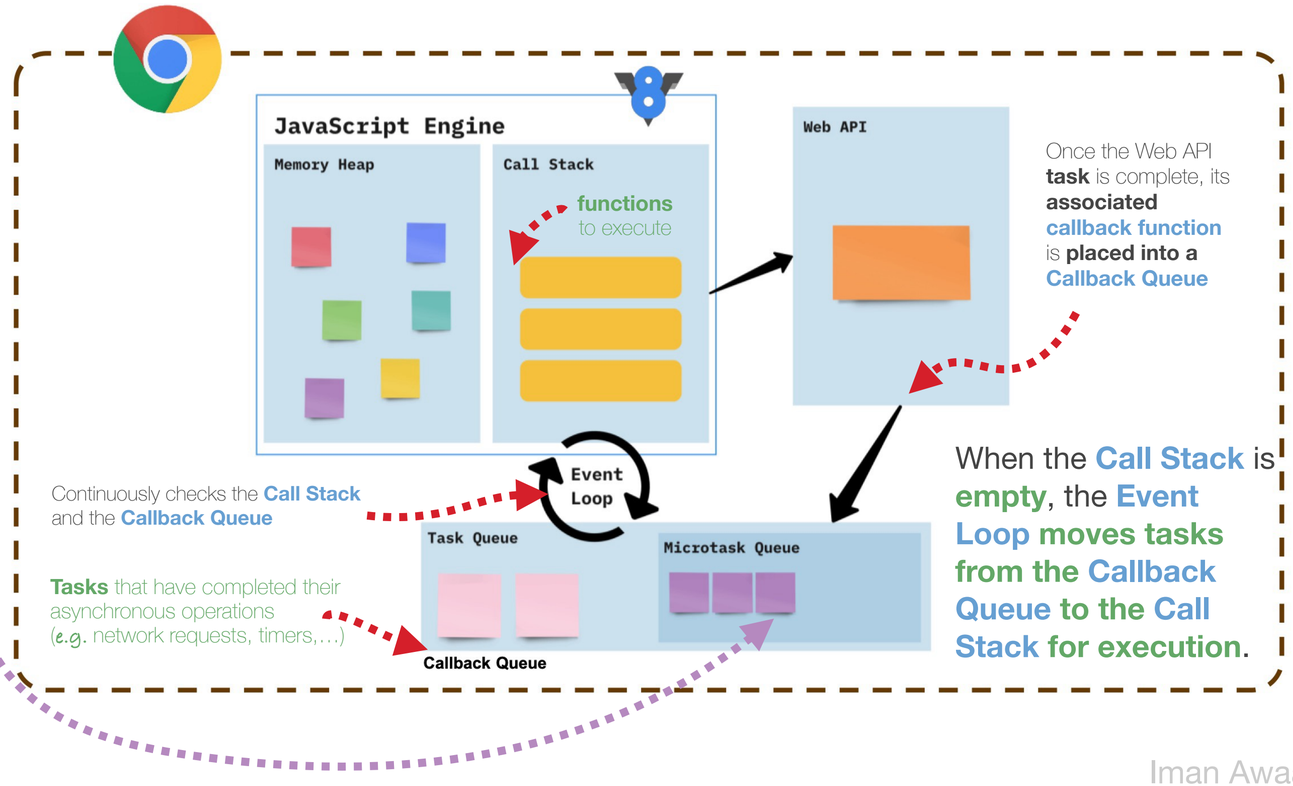


The JS Event Loop: Microtask Queue

…holds **callbacks** from **higher-priority asynchronous** operations, primarily **Promises** *e.g.* `.then()`, `.catch()`, `.finally()`) and `MutationObserver` callbacks… (`process.nextTick()` in Node.js also places its callbacks in the Microtask Queue)

When a **Promise resolves** or **rejects**, its associated `.then()`, `.catch()`, or `.finally()` **callback** is **immediately added** to the **Microtask Queue**

The **Event Loop** gives **priority** to the **Microtask Queue**. It will completely **empty** the Microtask Queue **before moving on** to process any tasks from the **Callback Queue** in the **same Event Loop iteration**.

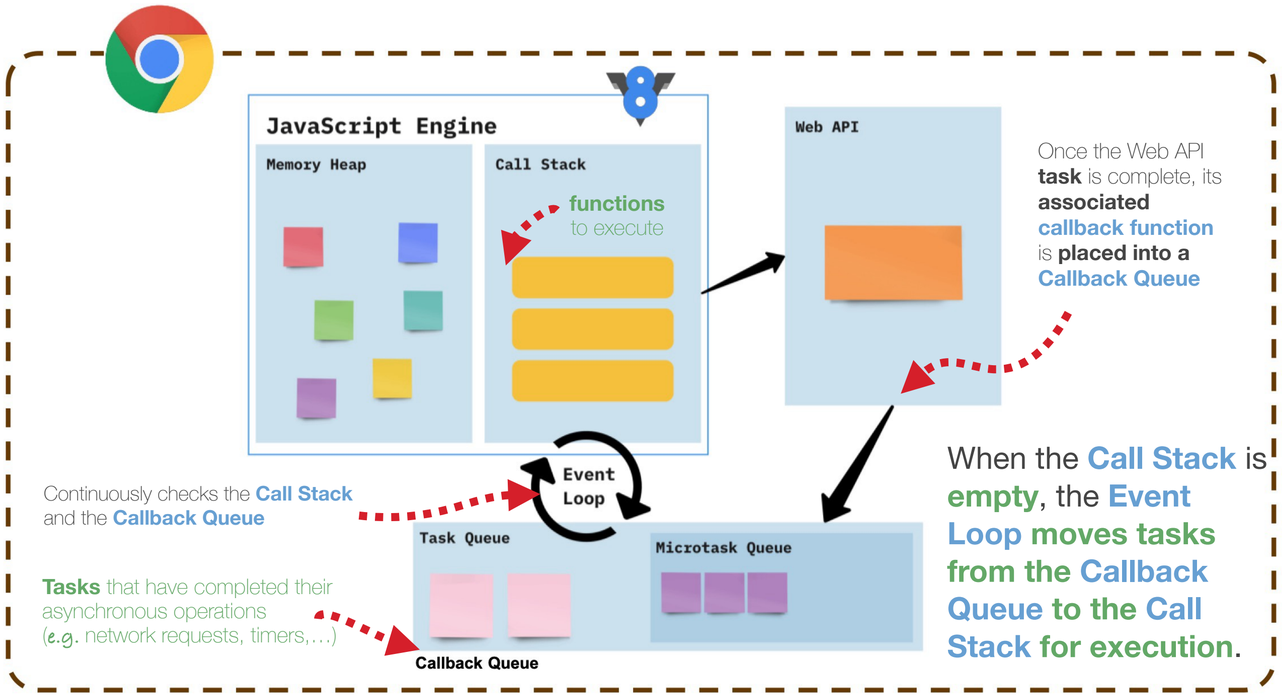


The JS Event Loop: differences between queues

Priority The **Microtask Queue** has a **higher** priority than the **Callback Queue**: all microtasks will be executed before any macrotasks are processed in a given **Event Loop** iteration

Contents The **Callback Queue** handles general **asynchronous** operations, while the **Microtask Queue** handles more **immediate**, often **Promise-related**, **asynchronous** operations

Execution Flow The **Event Loop** continuously checks if the **Call Stack** is **empty**. If it is, it first **drains** the entire **Microtask Queue**. Only after the Microtask Queue is empty does it pick one task from the **Callback Queue** to move to the **Call Stack** for execution… This cycle repeats…



The JS Event Loop: Priority in action

JavaScript

```
console.log("Start");

setTimeout(() => {
  console.log("Timeout callback (Macrotask)");
}, 0);

Promise.resolve().then(() => {
  console.log("Promise callback (Microtask)");
});

console.log("End");
```

Output:

Code

```
Start
End
Promise callback (Microtask)
Timeout callback (Macrotask)
```

The **Microtask Queue** has a **higher** priority than the **Callback Queue**: all microtasks will be executed before any macrotasks are processed in a given **Event Loop** iteration

Example shows the priority of the **Microtask Queue**, as the **Promise** **callback** is executed before the `setTimeout` **callback**, even though `setTimeout` was scheduled earlier with a 0ms delay!

Interpreted & JiT

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

JiT stands for **Just-In-Time compilation**:
…a compilation process where **code is translated into machine code during the execution** of a program, **rather than** being fully compiled **before execution** (ahead-of-time, *aka* AOT compilation), or simply **interpreted** line by line…

Advantage:
improved performance without sacrificing the **flexibility** and **dynamic nature** of Javascript… bridges the gap between **interpreted languages** (easy to run) and **compiled languages** (fast execution)

Concurrent

Javascript is a…

High-level

Asynchronous

Non-blocking

Single-threaded

Interpreted & JiT

Concurrent

language

The Event Loop The core of JavaScript's concurrency model: continuously checks the **Call Stack** for **functions** to execute and the **Callback Queue** for **tasks that have completed their asynchronous operations** (like network requests or timers). When the **Call Stack** is **empty**, the **Event Loop moves tasks from the Callback Queue to the Call Stack for execution**

Callbacks **Functions passed as arguments to other functions**, to be **executed later when an asynchronous operation completes**. This is a foundational pattern for handling asynchronous events

Promises **Objects representing the eventual completion** (or failure) **of an asynchronous operation** and **its resulting value**. They provide a more structured and readable way to handle asynchronous code compared to nested callbacks (*aka* callback hell)

async/await **Syntactic sugar** built on top of **Promises**, **making asynchronous code look and behave more like synchronous code**, improving **readability** and **maintainability**. **async functions** always **return a Promise**, and **await** **pauses** the **execution of an async function until a Promise settles**

Web Workers These provide a form of **multi-threading** **in the browser** by allowing **scripts to run in the background in separate threads**, **preventing them from blocking the main UI thread**. They communicate with the main thread via message passing

`setTimeout` and `setInterval`

Some stuff on implementation…

DOM: Document Object Model

Similar to HTML but written in another language…
If you change something in the DOM, the HTML changes automatically:

- represents HTML or XML in a tree structure
- has methods that allow the selection of specific HTML elements
- has objects and sub-objects, each with its own methods and properties

Most important are these mechanisms:

- query selector
- event listener
- style property

In a console, write the word `document.`

Try the DOM:

(viewer) https://software.hixie.ch/utilities/js/live-dom-viewer/

(parser)
https://www.w3schools.com/xml/tryit.asp?filename=try_dom_loadxmltext

Some stuff on implementation…

JQuery

jQuery is a JS library: . It makes handling HTML documents in different browsers simpler

You can use the API to access the libraries' methods

It improves the UI interface (*e.g.* data pickers, sliders, collapsible panels,…)

It can **dynamically** modify the CSS and handle AJAX requests without reloading the page…

Use it from your own server or through a jQuery CDN (Content Delivery Network). Included in the head of the webpage.

Starts with the $.

https://jquery.com/

Some stuff on implementation…
AJAX

Asynchronous Javascript and XML

Uses XML HTTP Request Object to interact with the server

A technique to exchanging data with a server without reloading the page

JQuery simplifes AJAX request

Go to twitter, inspect the page, go to network tab, and look for the automatic refresh...

AJAX request can be text file, JSON object, HTML…

Some stuff on implementation…

Node.js

Non-blocking behaviour is **crucial** for **responsive web applications** and **efficient server-side operations** in **Node.js**, where **callbacks** are extensively used **to manage the flow of asynchronous tasks**!

Effective, but, deeply **nested** **callbacks** can lead to "**callback hell**" or "**pyramid of doom**": difficult to read and maintain

Solution: asynchronous patterns like **Promises** and `async`/`await`…

Node.JS is Javascript runtime environment for the **server side**
(*i.e.* to run JS on the server)

NodeJS + HTTP ➔ EXPRESS framework

…used to develop backend applications

Node Package Manager (**NPM**) streamlines the process of building JS applications: provides a centralised system for discovering, installing, managing, and sharing reusable code…

online registry for JavaScript packages

Javascript concurrency

The Event Loop The core of JavaScript's concurrency model: continuously checks the **Call Stack** for **functions** to execute and the **Callback Queue** for **tasks that have completed their asynchronous operations** (like network requests or timers). When the **Call Stack** is **empty**, the **Event Loop moves tasks from the Callback Queue to the Call Stack for execution**.

Callbacks **Functions passed as arguments to other functions**, to be **executed later when an asynchronous operation completes**. This is a foundational pattern for handling asynchronous events!

Promises **Objects representing the eventual completion** (or failure) **of an asynchronous operation** and **its resulting value**. They provide a more structured and readable way to handle asynchronous code compared to nested callbacks (*aka* callback hell).

async/await **Syntactic suga**r built on top of **Promises**, **making asynchronous code look and behave more like synchronous code**, improving **readability** and **maintainability**. **async functions** always **return a Promise**, and **await pauses** the **execution of an async function until a Promise settles**.

Web Workers These provide a form of **multi-threading** **in the browser** by allowing **scripts to run in the background in separate threads**, **preventing them from blocking the main UI thread**. They communicate with the main thread via message passing.

`setTimeout` and `setInterval`

Why is what we covered today relevant (or important) for your projects?

Why is what we covered today relevant or important for your projects?

…anything that **needs to do multiple things "at the same time”**

Beware
concurrency (async/await/event loops)
is different than **parallelism** (threads/CPU cores)

You have to be careful about synchronisation and data races!

Maybe of interest: https://github.com/earthboundkid/flowmatic?tab=readme-ov-file
“a generic Go library that provides a structured approach to concurrent programming. It lets you easily manage concurrent tasks in a manner that is simple, yet effective and flexible.”

I mark the hours everyone
Nor have I yet outrun the sun
My use and value unto you
Are guaged by what you have to do.

- Inscription on time turner.