Async Programming And You

By Zain Afzal
Software Engineer @ Google

@zainafzal08
@blockzain

Recap

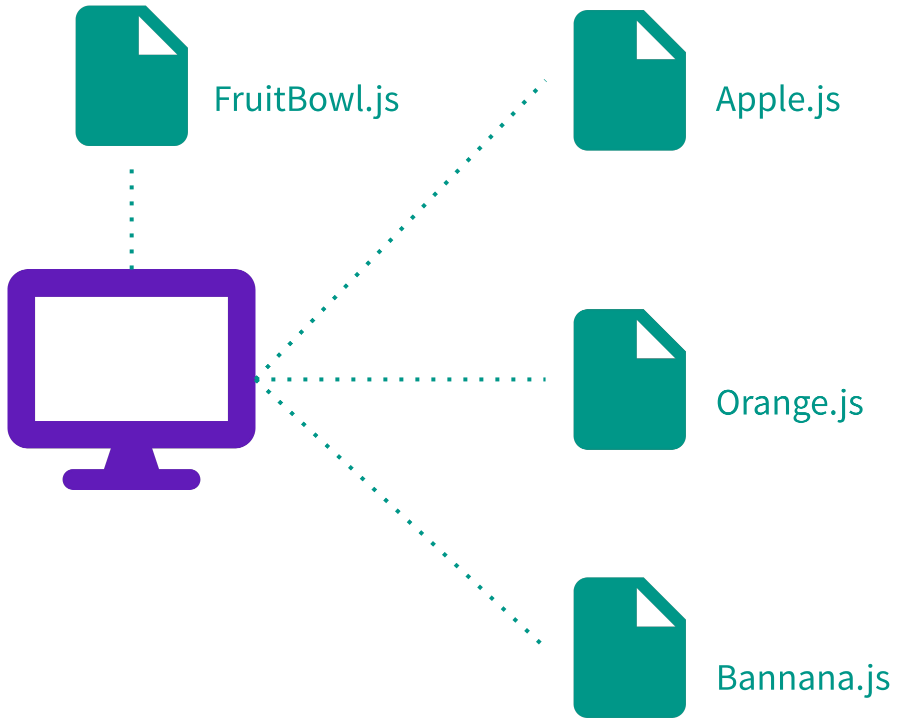
Introduction To Web
Alex Hinds

Dom and Events
Anna Azzam

Async And You
Zain Afzal

Week 1 **Week 2** **Week 3**

Quick Refresher on Modules



FruitBowl.js

```
import Apple from './Apple.js';
import Orange from './Orange.js';
import Bannana from './Bannana.js';

makeFruitBowl(Apple, Orange, Bannana)
```

Quick Refresher on Modules

```
Apple = {
    yummy: true
}
export Apple

Orange = {
    scurvy: null
}
export Orange

Banana = {
    ew: 'yes'
}
export default Banana
```

```
// Import apple and Banana, but rename Banana
to B
import {Apple, Banana as B} from 'fruits.js';
// import the default export from fruits.js
and name it O
import O from 'fruits.js;
```

DEMO
battery.js

Recap Of Basic Client Server Interactions

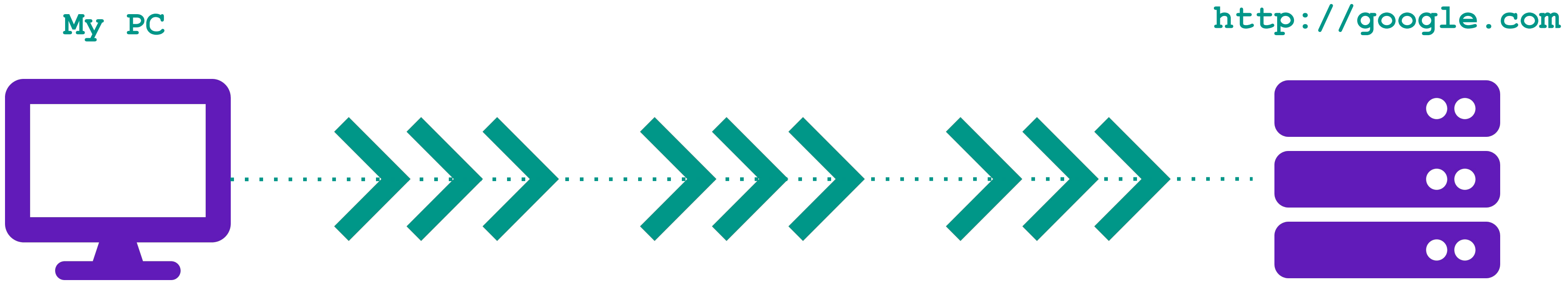
Client

Your home PC, laptop, phone, washing machine etc.

Server

A dedicated machine run by a website to receive and process requests

Recap Of Basic Client Server Interactions



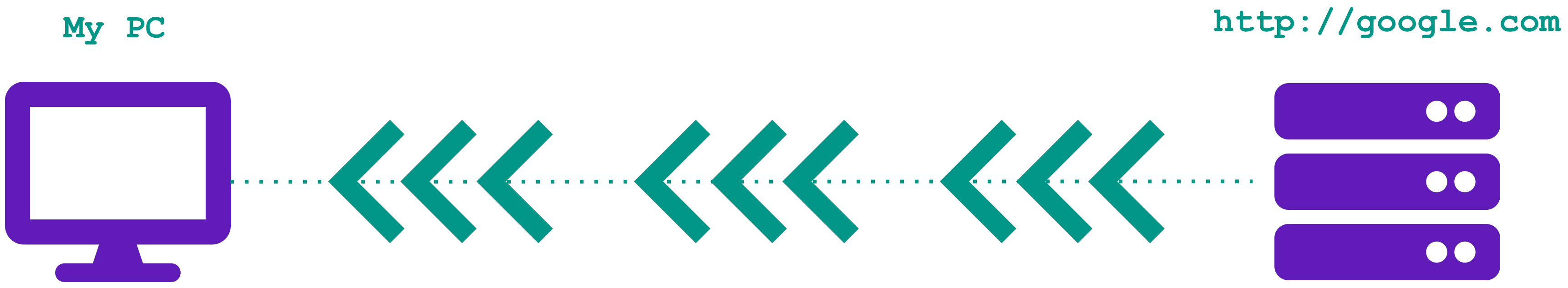
Client

Sends a request to a server for a page, lets say google.com

Server

Processes the request (what is this client asking for, are they allowed to see this?)

Recap Of Basic Client Server Interactions



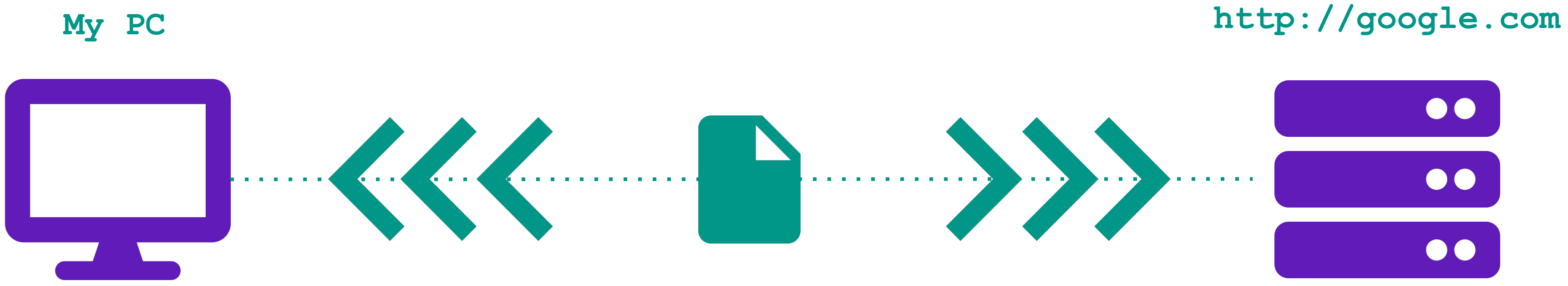
Client

Sends a request to a server for a page, lets say google.com

Server

Decides what to send back, creates a response of HTML, CSS and JS and shoots it off This is called **Server Side Rendering**

Recap Of Basic Client Server Interactions



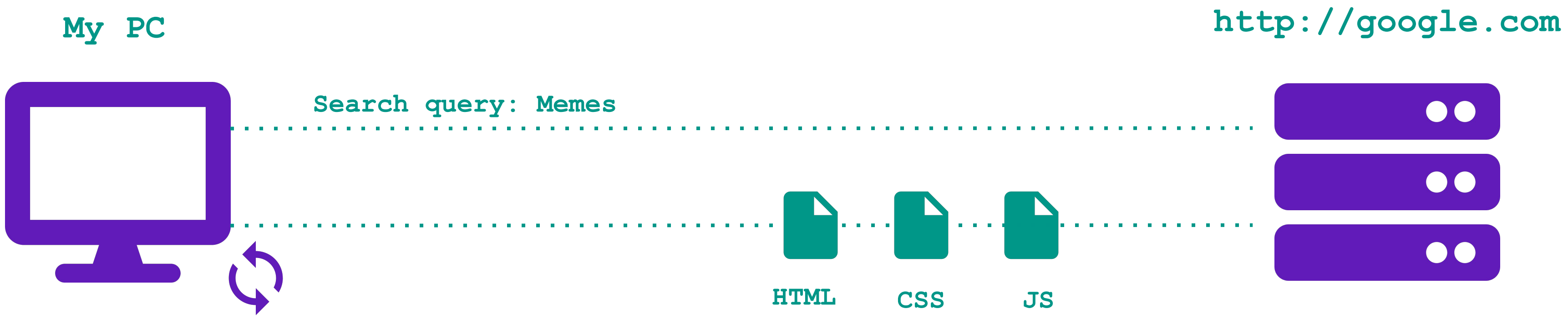
Client

Needs to spend time waiting for this (sometimes heavy) file to download and then spend time parsing and rendering it

Server

Needs to spend time and resources forming a full page response and sending it over

Forms With Full Page Reload



1. Reparse
2. Rerender

Client

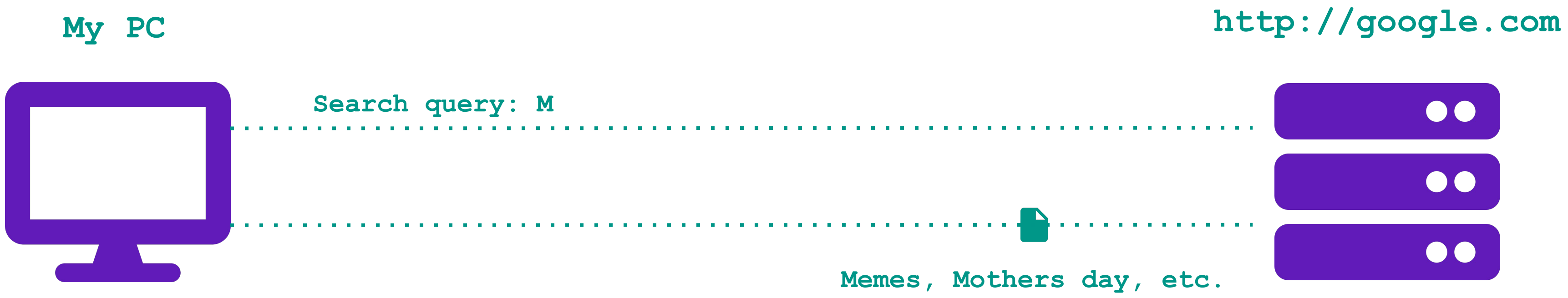
Forms and interactivity were built on top of this, a request with some information is sent and the response triggers a a reparse and rerender

Server

And the server would use this information to create a brand new page and return that

DEMO
google.com

The Magic of AJAX



Client
via js sends just a small request to autocomplete ‘M’ and adds the suggestions to the page without doing a full rerender

Server
Quickly processes this request and sends back just some suggestions

AJAX

A development technique in which a frontend client sends and receives data to the backend asynchronously (in the background) without interfering with the display and behavior of the existing page.

- Stands for Asynchronous Javascript And XML
- Smoothens UI
- Improves Speed
- Allows for offline web apps
- Separation frontend and backend responsibilities

Roadmap

1. The Event Loop
2. XMLHTTP And Callback Hell
3. Fetch and Promise Heaven
4. Promise++ And Async Await
5. Assignment
6. Beyond 2041

For the 'beyond 2041 lecture' we will cover some interesting things that you will most likely encounter if you look into frontend development in industry. This section will not be examinable so feel free to suggest any topics you'd like us to cover in the field of JS

Suggest Topics/Questions here
https://forms.gle/mqsehtzHFW7LQ9qo6

Roadmap

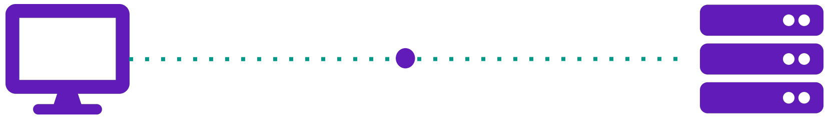
The Event Loop

What does asynchronous mean anyway?

Synchronous

Operations can only occur one at a time. You must wait for current action to finish before starting a new one.

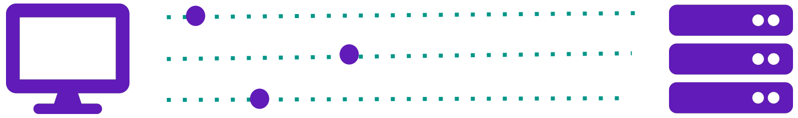
Ex: The old server client model where each request had to be completed before you sent a new one



Asynchronous

Multiple operations can occur at the same time. Programmers can parallelise their program and do many things at once.

Ex: The AJAX model where you can fire off 10 requests before the first one returns



Concurrency models

Doing multiple things at the same time requires a concurrency model, that is how do we distribute our work and reason about our code. There are 3 main ways

- Threads / Preemptive Multitasking
- Coroutines / Cooperative Multitasking
- Event Driven

Threads / Preemptive Multitasking

Here you write a bunch of code and spin up threads, each threads runs the code assuming that it’s the only thing running. At any random time the thread can be stopped, it’s state saved and the cpu tasked with some other bit of work.

We say that a thread can be preempted, here we need mutex's and semaphores to guard critical sections & variables to make sure we don’t get interrupted if we are doing something fiddily. This is the typical multi-tasking model found in many languages like Java and Python.

Do OS if you want to learn more, but this is not a model useful to us as all transformations to the dom **must happen in a predictable order** to avoid broken pages

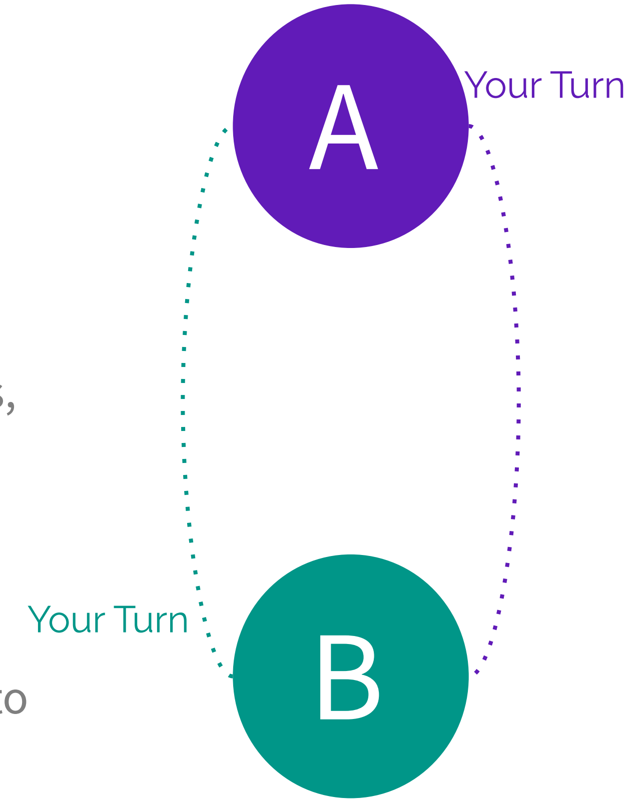
Coroutines / Cooperative Multitasking

Here we have co-routines, these are separated flows of execution which run in tandem.

co-routines can cooperatively **yield** to other coroutines.

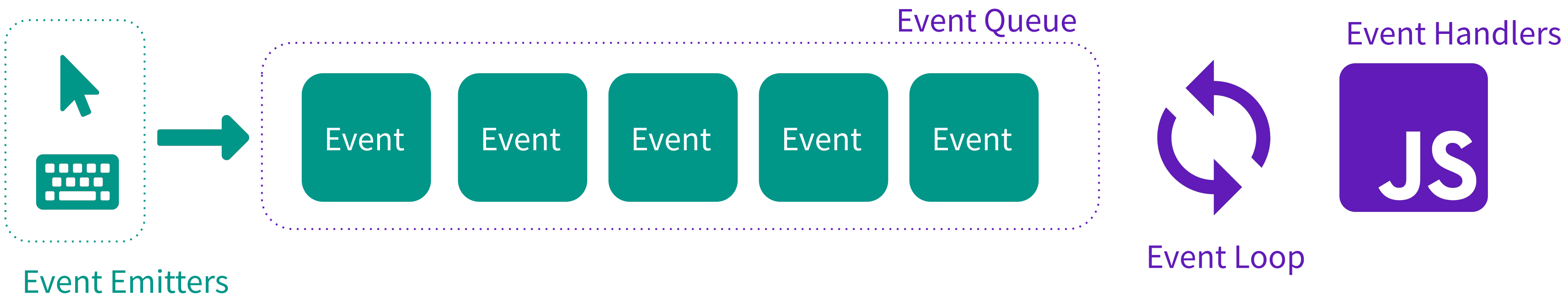
yield saves the state of co-routine A, and resumes B's state from it's previous yield point. No preemption between yields, so no need for mutex's or semaphores to guard critical sections.

This is better but often designed with the idea of having predefined processes that share the CPU and isn't as suited to the dynamic way events are generated on the web



Event Driven

This is the way JS handles concurrency, the core of this model is the idea of a **event loop**. This is a queue of events that have occured, events are popped off to be handled and pushed on as they occur



Each event is given to its respective handler and the loop waits for the handler to finish processing before handling the next event. In this way our javascript runs start to finish in a predictable way (no need for locks) but we can still have many events at the same time.

What is a callback?

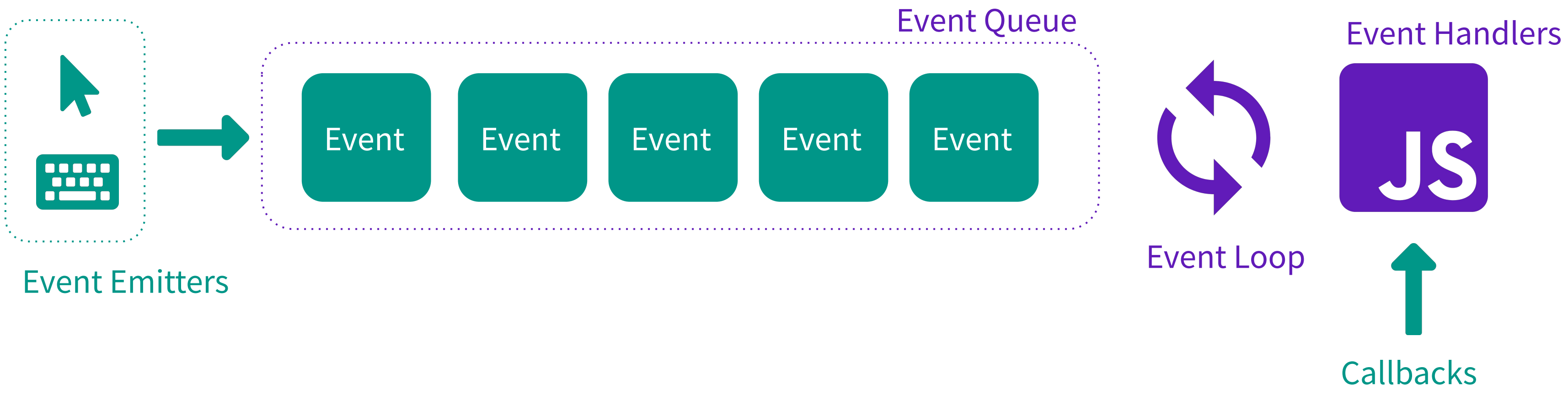
- A function that is called when the pending work has completed
- Can use either a named function or an anonymous lamdba

```
fs.readFile('foo.txt', (result, err) => {
 // Do something with result here
})
```

```
function onComplete(result, err) {
 // Do something with result here
}
fs.readFile('foo.txt', onComplete)
```

What is a callback?

Our callbacks are our event handlers, they get invoked with the response of some event



DEMO
The event Loop

DEMO

Timer

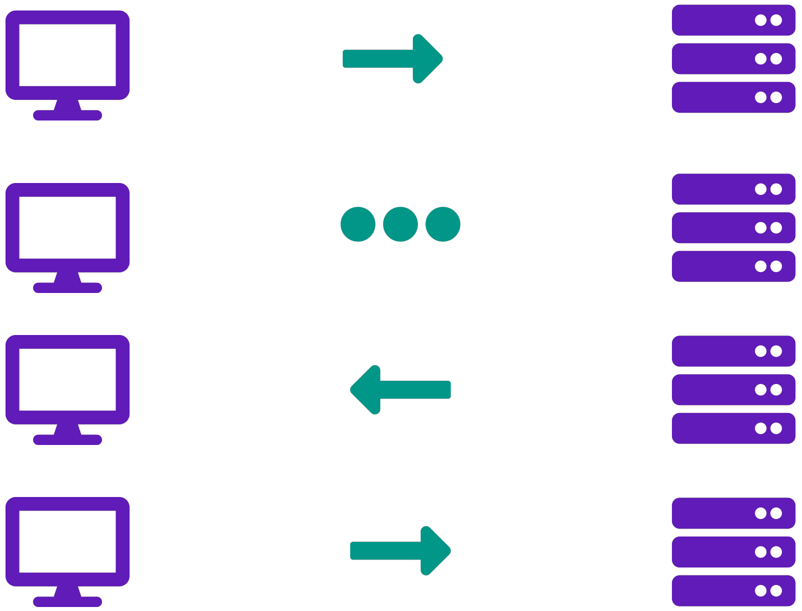
Network and Resource Fetching

only 1 task can run at a time so event loops suck for parallelized workloads, i.e CPU bound tasks like large calculations but it’s incredibly well suited to IO bound tasks, that is tasks that spend lots of time waiting for a server or device to respond.

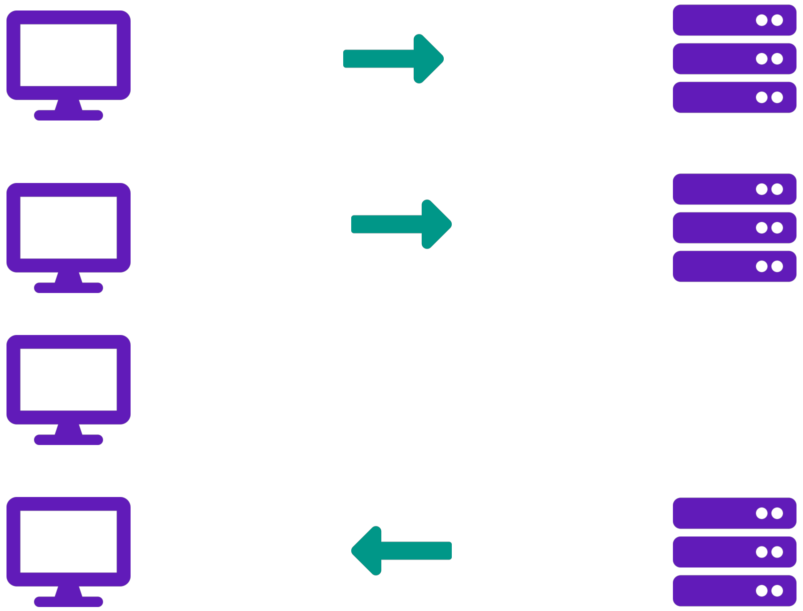
We can fire off a request to a server and then continue doing other work, when the server responds a event is generated which we can then handle.

Network and Resource Fetching

Synchronous Way



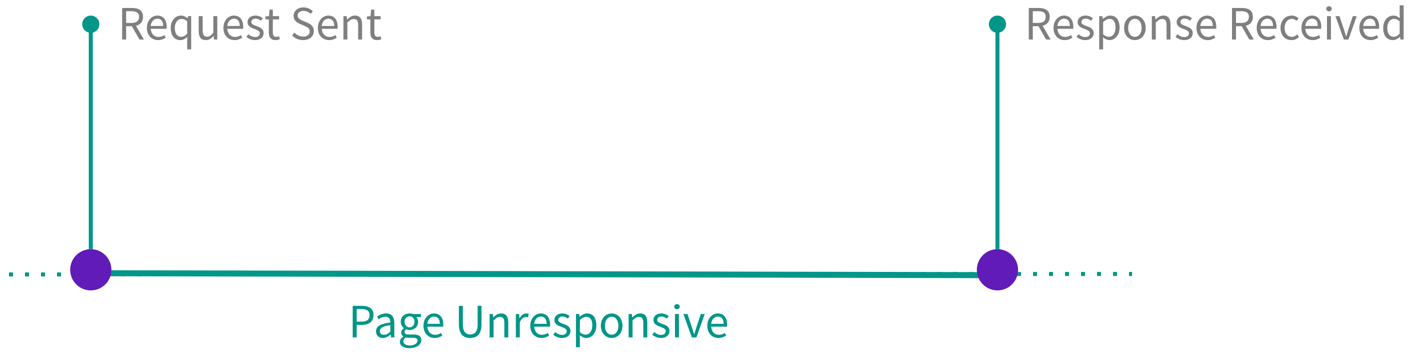
Asynchronous Way



DEMO
async

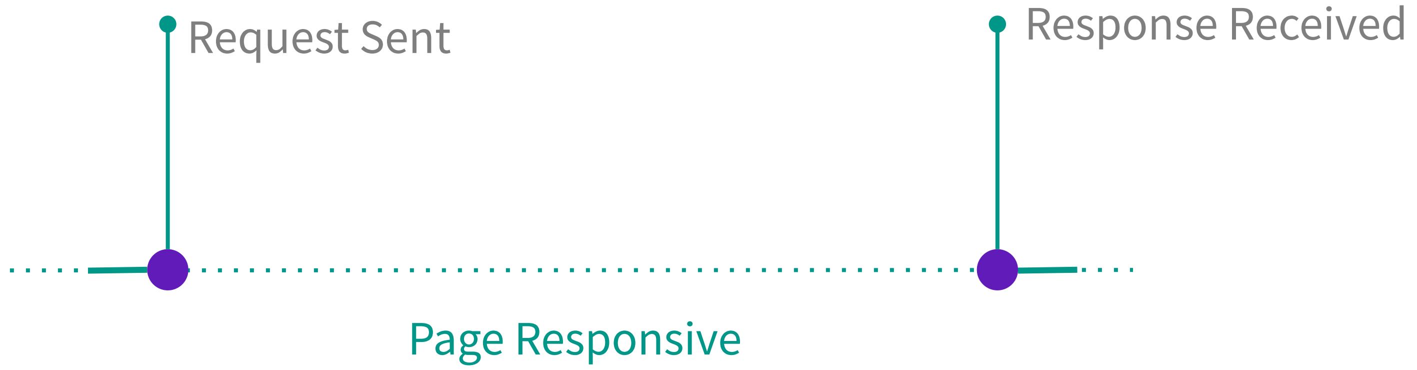
Blocking

Remember JS can only run 1 thing at a time so if that 1 thing takes 6 seconds then during that time things can be added to the queue but nothing gets popped off.



Blocking

By not waiting for a response and expecting the event loop to trigger a handler when the time is right we allow ourselves to keep the page running (we can also do heavy tasks in **webworkers** but we’ll cover that later)



Roadmap

1. The Event Loop
2. XMLHTTP And Callback Hell
3. Fetch and Promise Heaven
4. Promise++ And Async Await
5. Assignment
6. Beyond 2041

XMLHTTP And Callback Hell

Yes that’s the technical term

How do you actually do this?

Writing code that is async is a difficult problem and there have been many diff paradigms over the life of JS. The first is to use callbacks with XMLHTTP (Notice the XML in the name, very AJAX much wow)

```
console.log('Loading...');
const request = new XMLHttpRequest();
// `false` makes the request synchronous
request.open('GET', 'url', false);
request.send(null);
if (request.status === 200) {
 console.log(request.responseText);
}
```

```
console.log('Loading...');
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ipify.org?format=json"
true);
xhr.onload = function (e) {
 if (xhr.readyState === 4) {
   if (xhr.status === 200) {
     console.log(xhr.responseText);
   }
 }
};
xhr.send(null);
```

DEMO
XMLHTTP

Error Handling with callbacks

Errors are fairly common in web, maybe you weren’t logged in so your request failed, maybe your token expired, maybe the connection dropped. Your network interactions WILL fail so you have to be able to gracefully handle this.

DEMO
Handling errors in callbacks

Chaining

Another common thing in web is wanting to do things in a specific order, as a simple example lets say we want to animate some text being written.

There is no sleep in js that you should use (think about blocking), so how would you go about writing this with sleeps?

```
const output = document.getElementById('output')
output.innerText += 'h'
sleep(0.1)
output.innerText += 'e'
sleep(0.1)
output.innerText += 'l'
sleep(0.1)
output.innerText += 'l'
sleep(0.1)
output.innerText += 'o'
```

DEMO
become a hacker

Callback Hell

This was a simple example but it applies a lot to network requests where we need a bit of data from source A before we get the data from source B. So we end up nesting our callbacks to enforce a linear order of code execution.

```
getData(function(x){
    getMoreData(x, function(y){
        getMoreData(y, function(z){
        ...
        });
    });
});
```

Avoiding Callback Hell

Callback hell makes it

- difficult to reason about the code and what it’s doing
- difficult to debug
- difficult to extend / edit
- looks ugly
- fragile, a single missed callback in that mess causes a failure
- difficult to accurately catch and resolve errors

Avoid callback hell by

- Not nesting functions, give each one a name and place at the top scope
- Keep the code shallow wherever possible
- Avoid overuse of callbacks for things that can be syncronous
- Use promises

Avoiding Callback Hell

```
function sayHello(callback) {
   console.log('Hello!');
   callback();
}
sayHello(() => {
   console.log('We said hello.');
});
```

```
function sayHello() {
    console.log('Hello!');
}
sayHello();
console.log('We said hello.');
```

Roadmap

Fetch and Promise Heaven

Ok that’s not the technical term

Promises

An abstraction around async work giving us access to “future” values.

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

A Promise can be in a number of states

- Fulfilled (or Resolved): The action relating to the promise succeeded.
- Rejected: The action relating to the promise failed.
- Pending: Hasn't yet fulfilled or rejected.
- Settled: Has fulfilled or rejected.

DEMO

How to use promises