WEAVING WEBS OF WORKERS

@TRENTMWILLIS

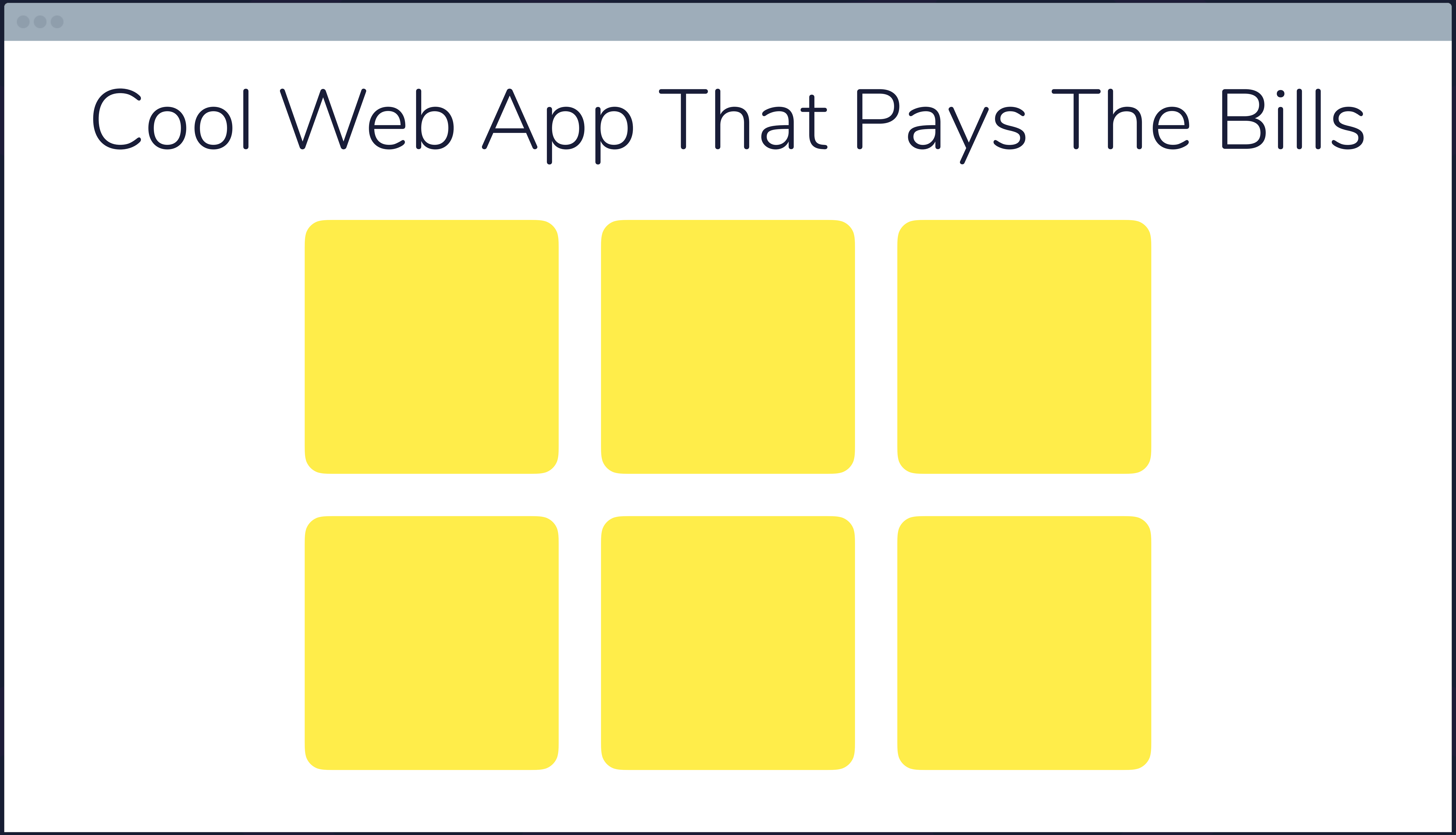
#JSKongress

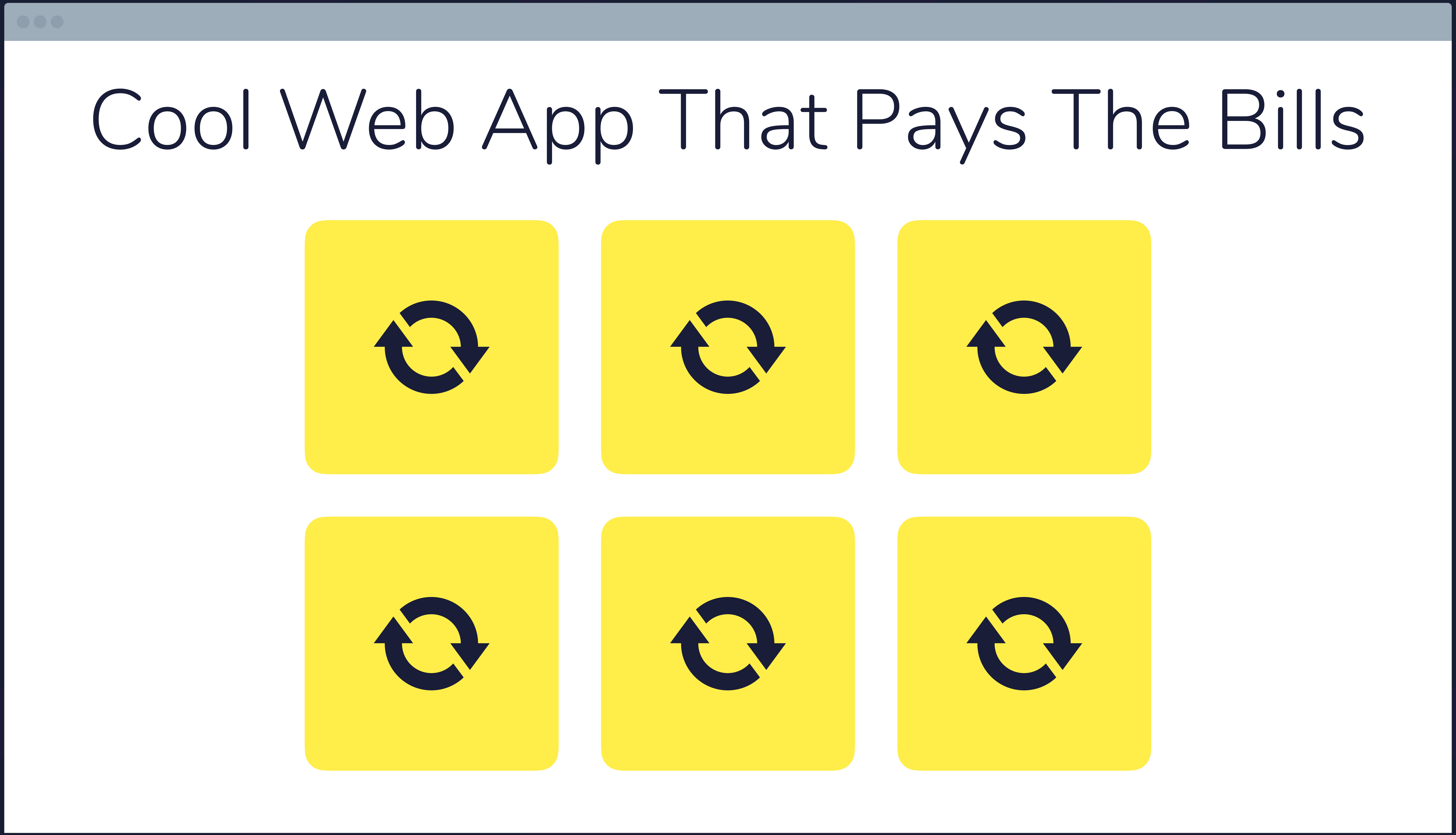
Hallo **JS Kongress**!

Hallo **JS Kongress**!
Raise your ✋ if you work on a web application.

Cool Web App That Pays The Bills

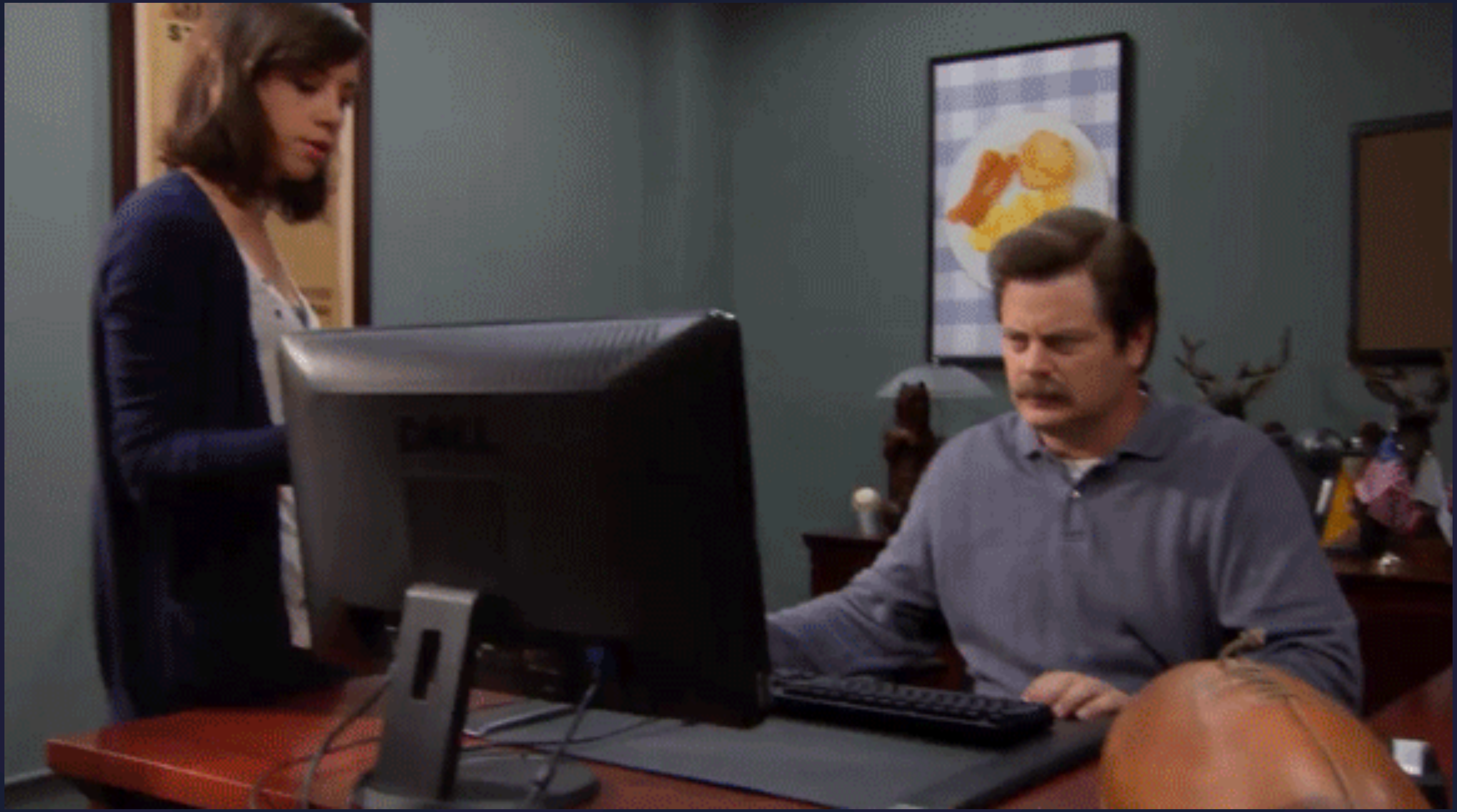
@TRENTMWILLIS #JSKongress

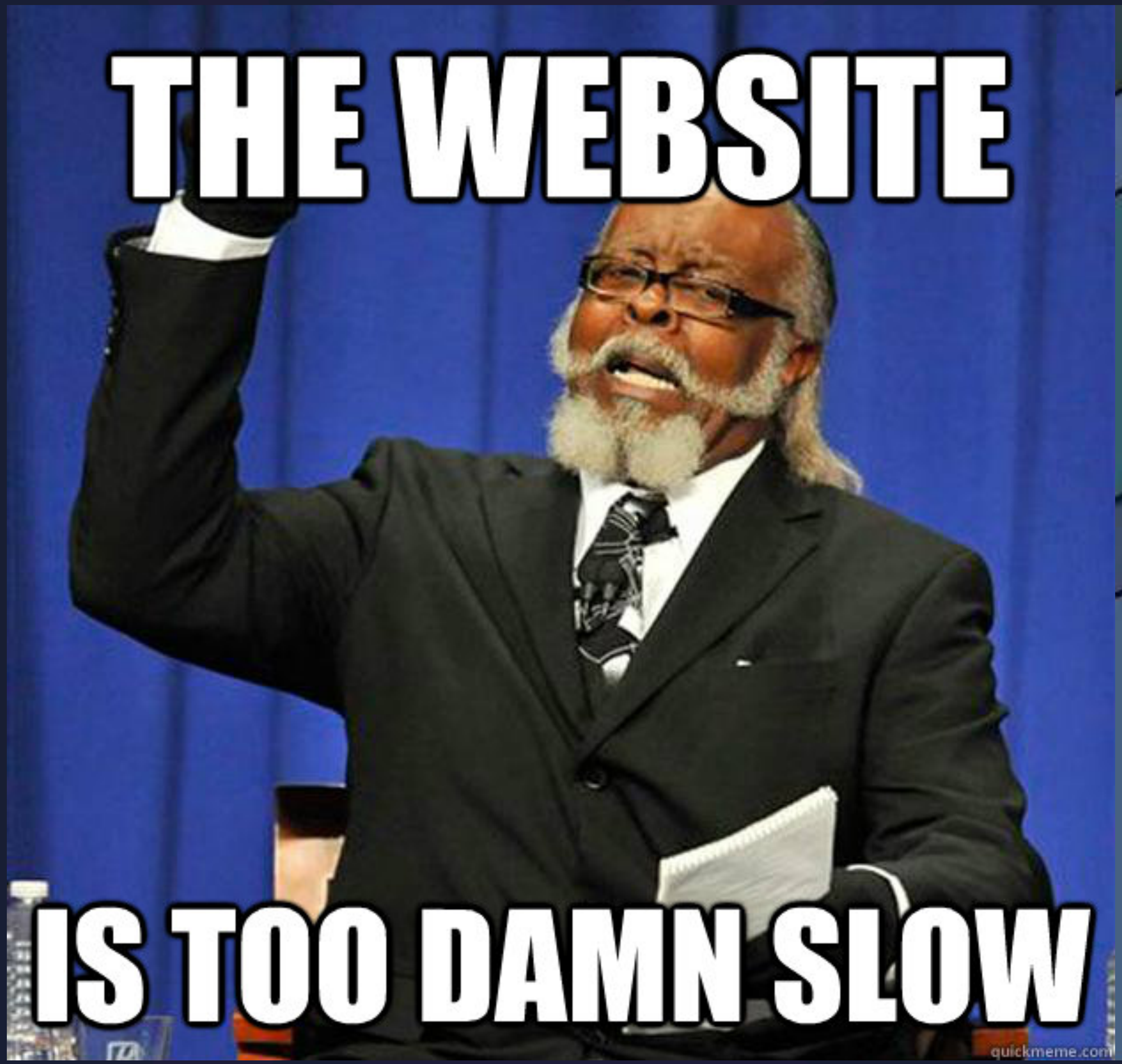
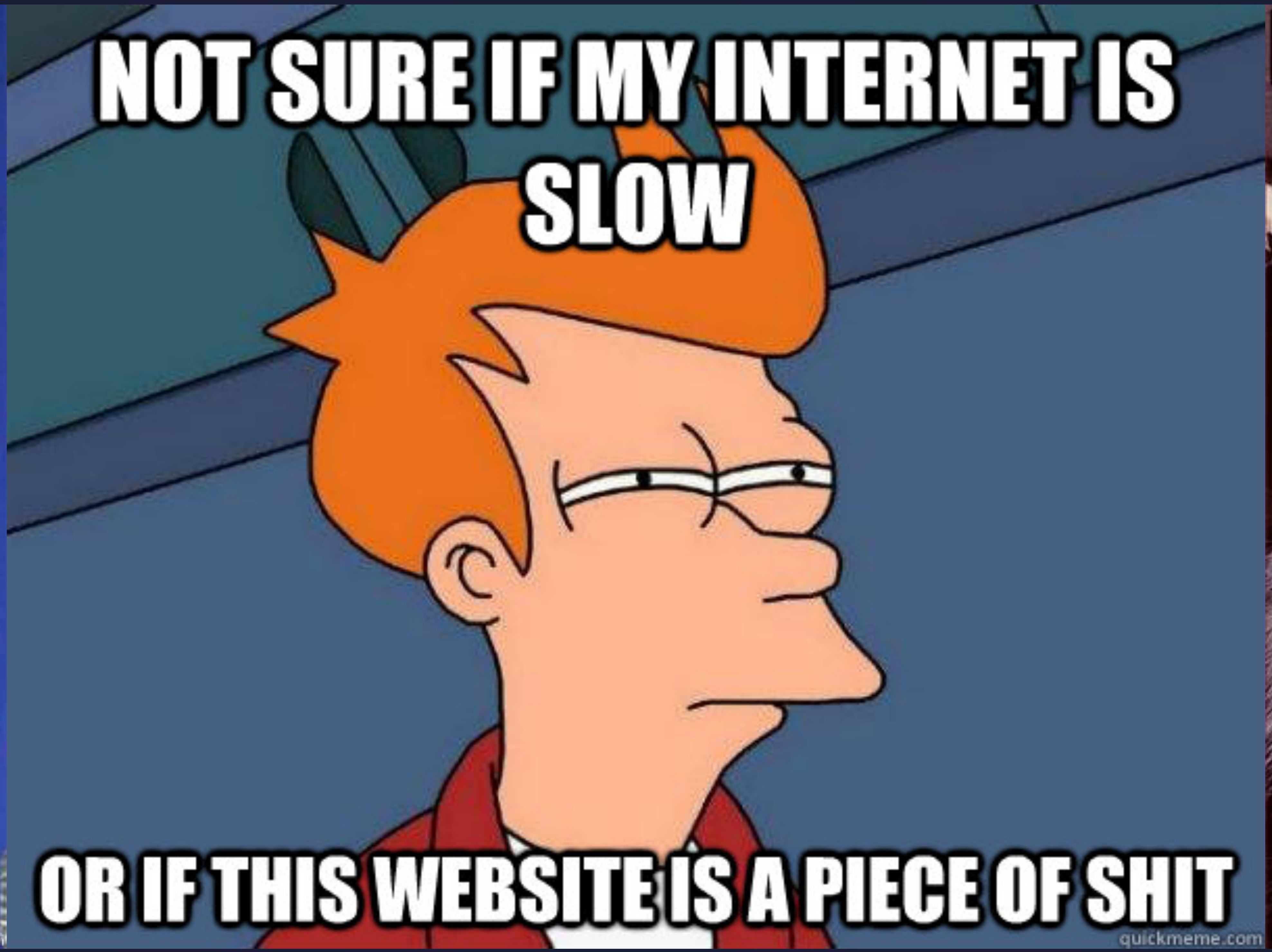
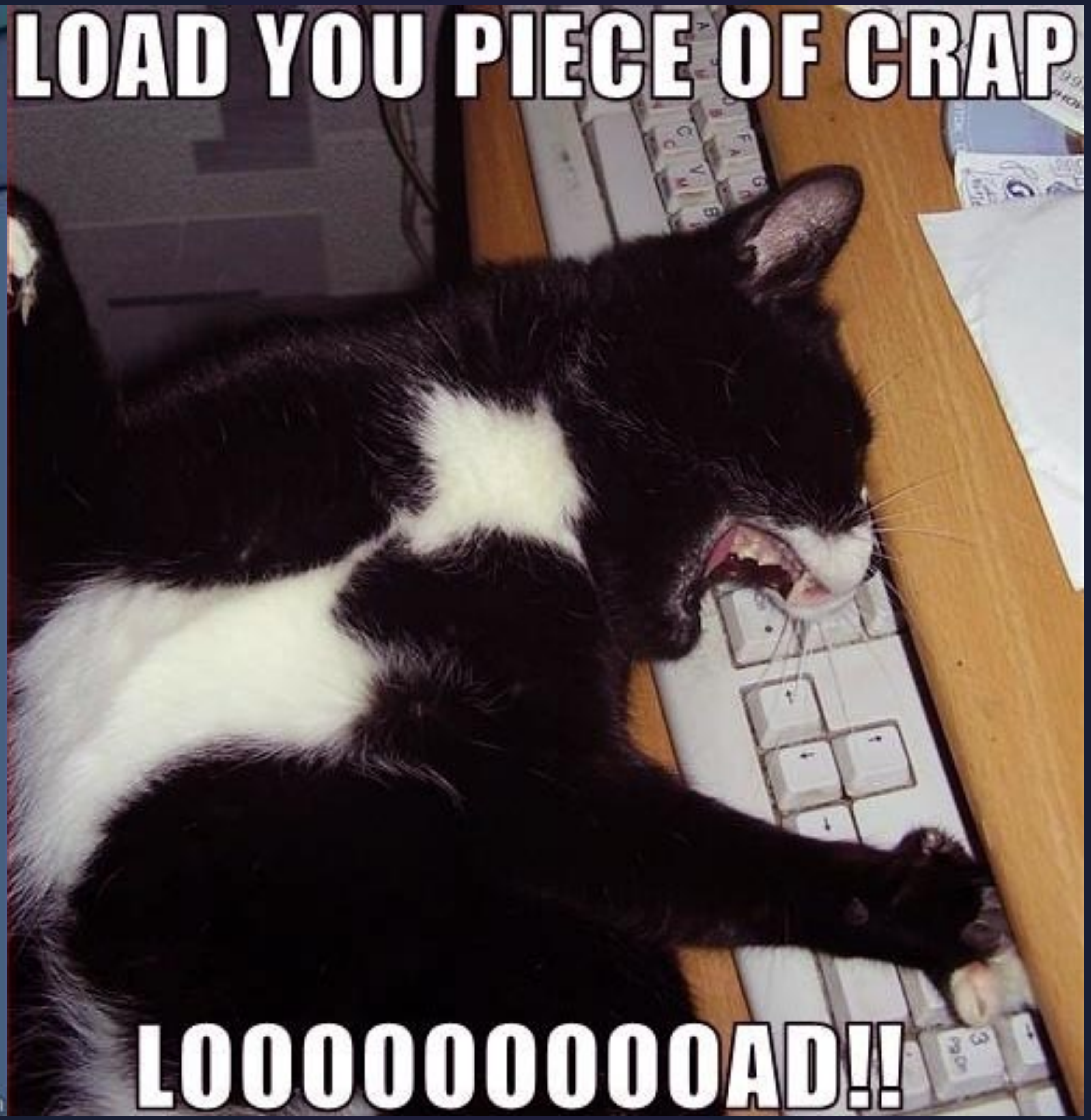

Cool Web App That Pays The Bills


Cool Web App That Pays The Bills

@TRENTMWILLIS

#JSKongress


DELL


THE WEBSITE
IS TOO DAMN SLOW
quickmeme.com

NOT SURE IF MY INTERNET IS SLOW
OR IF THIS WEBSITE IS A PIECE OF SHIT
quickmeme.com

LOAD YOU PIECE OF CRAP
LOOOOOOOOOAD!!

How do we prevent **numerous**, **large**, and/or **slow** data requests from impacting our users?

Web Workers.

They **help**, but also **complicate**.

WEAVING WEBS 🕸F WORKERS

WEAVING WEBS OF WORKERS

@TRENTMWILLIS

Senior UI Engineer at **Netflix**

The Web Workers API

“allows Web application authors to spawn background workers running scripts in parallel to their main page.”

```
new Worker('worker.js');
new Worker('worker.js');
new SharedWorker('worker.js');
new Worker('worker.js');
```

It's like a <script> but loads in a different thread!

Web Workers allow

"for thread-like operation with message-passing as the coordination mechanism."

```
// main thread
worker;
```

```
// main thread
worker.postMessage(message);
```

```
// worker thread
self // WorkerGlobalScope
```

```javascript
// worker thread
self.addEventListener(
  'message',
  event => {
    console.log(event.data);
  }
);
```

```
// worker thread
self.addEventListener(
  'message',
  event => {
    console.log(event.data);
    self.postMessage(message);
  }
);
```

Messaging is the bulk of the web workers API you need!

```
// main thread
worker.addEventListener(
  ‘message',
  event => console.log(event.data)
);
```

```js
// main thread
worker.terminate();
```

PROBLEMS

PROBLEM

Knowing when a task is completed

PROBLEM

Management and coordination of multiple workers

PROBLEM

Difficult to test

PROBLEM

No dynamic definition of workers

SOLUTIONS

PROBLEM

Knowing when a task is completed

PROBLEM Knowing when a task is completed

Turn messages into **Promises**

Replace one platform feature with another!

PROBLEM Knowing when a task is completed
SOLUTION Turn messages into **Promises**

```
const postMessage = (worker, message) => new Promise(resolve => {
    const resolution = (event) => {
        worker.removeEventListener('message', resolution);
        resolve(event.data);
    };
    worker.addEventListener('message', resolution);
    worker.postMessage(message);
});
```

PROBLEM Knowing when a task is completed

SOLUTION Turn messages into **Promises**

```
postMessage(worker, data).then(response => console.log(response));
```

PROBLEM Knowing when a task is completed
SOLUTION Turn messages into **Promises**

```
const response = await postMessage(worker, data);
console.log(response);
```

PROBLEM Knowing when a task is completed

SOLUTION Turn messages into **Promises**

promise-worker

github.com/nolanlawson/promise-worker

PROBLEM

Management and coordination of multiple workers

PROBLEM Management and coordination of multiple workers

Use **Promises** (again)

PROBLEM Management and coordination of multiple workers

Expose Worker methods as

main thread functions

PROBLEM Management and coordination of multiple workers

SOLUTION Expose Worker methods as **main thread functions**

```
backendOneWorker
backendTwoWorker
```

PROBLEM Management and coordination of multiple workers
SOLUTION Expose Worker methods as **main thread functions**

```
const data = await Promise.all([
    backendOneWorker.fetch('first'),
    backendTwoWorker.fetch('second')
]);
```

PROBLEM Management and coordination of multiple workers
SOLUTION Expose Worker methods as **main thread functions**

```
const data = await Promise.all([
    backendOneWorker.fetch('first'),
    backendTwoWorker.fetch('second')
]);
const result = await processingWorker.process(data);
console.log(result);
```

PROBLEM Management and coordination of multiple workers
SOLUTION Expose Worker methods as **main thread functions**

```
const data = await Promise.all([
    backendOne.fetch('first'),
    backendTwo.fetch('second')
]);
const result = await processing.process(data);
console.log(result);
```

PROBLEM Management and coordination of multiple workers

SOLUTION Expose Worker methods as **main thread functions**

Workerize

github.com/developit/workerize

PROBLEM

No dynamic definition of workers

PROBLEM No dynamic definition of workers

Create Workers from **Blob URLs** of functions

PROBLEM No dynamic definition of workers

SOLUTION Create Workers from **Blob URLs** of functions

```
const workerFromFunction = (fn) => {
    const src = `(${fn})();`;
    const blob = new Blob([src], {type: 'application/javascript'});
    const url = URL.createObjectURL(blob);
    return new Worker(url);
};
```

PROBLEM No dynamic definition of workers

SOLUTION Create Workers from **Blob URLs** of functions

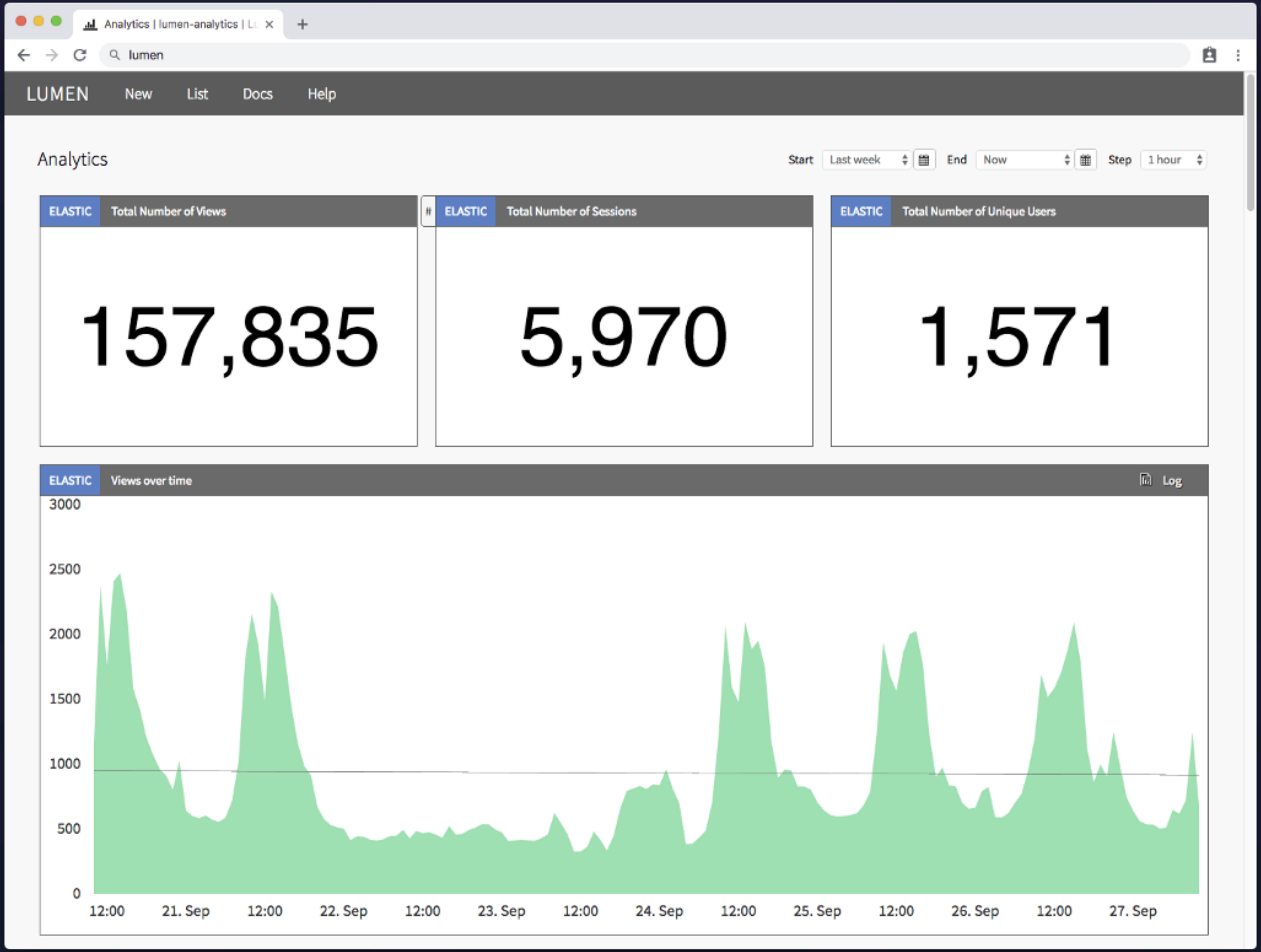
greenlet

github.com/developit/greenlet

Lumen

bit.ly/netflix-lumen

Lumen



Analytics | lumen-analytics | L

lumen

LUMEN New List Docs Help

Analytics

Start Last week End Now Step 1 hour

ELASTIC Total Number of Views

157,835

ELASTIC Total Number of Sessions

5,970

ELASTIC Total Number of Unique Users

1,571

ELASTIC Views over time

Lumen

“The majority of data operations in Lumen are done in Web Workers. This allows Lumen to keep the main thread free for user interactions, such as scrolling and interacting with individual charts, as the dashboard loads all of its data.”

This is how we "weave" a web of web workers!

Worker-To-**Worker** Communication

```
// worker thread
const workerInWorker = new Worker('worker.js');
```

MessageChannel

MessageChannel
consists of 2
MessagePorts

```
// main thread
const worker1 = new Worker('worker-1.js');
const worker2 = new Worker('worker-2.js');
```

```
// main thread
const worker1 = new Worker('worker-1.js');
const worker2 = new Worker('worker-2.js');

const channel = new MessageChannel();
```

```
// main thread
const worker1 = new Worker('worker-1.js');
const worker2 = new Worker('worker-2.js');

const channel = new MessageChannel();

worker1.postMessage('MessagePort', [channel.port1]);
worker2.postMessage('MessagePort', [channel.port2]);
```

Transferable

“[A Transferable] represents an object that can be transferred between different execution contexts, like the main thread and Web Workers.”

```
// worker thread
self.addEventListener('message', (event) => {
    if (event.ports.length) {
    }
});
```

```
// worker thread
self.addEventListener('message', (event) => {
    if (event.ports.length) {
        event.ports[0].onmessage = event => console.log(event.data);
        event.ports[0].postMessage('hello from worker 2’);
    }
});
```

```
const data = await Promise.all([
    backendOneWorker.fetch('first'),
    backendTwoWorker.fetch('second')
]);
const result = await processingWorker.process(data);
console.log(result);
```

You can do this entirely off the main thread!

Non-Blocking **Canvas** Graphics

OffscreenCanvas

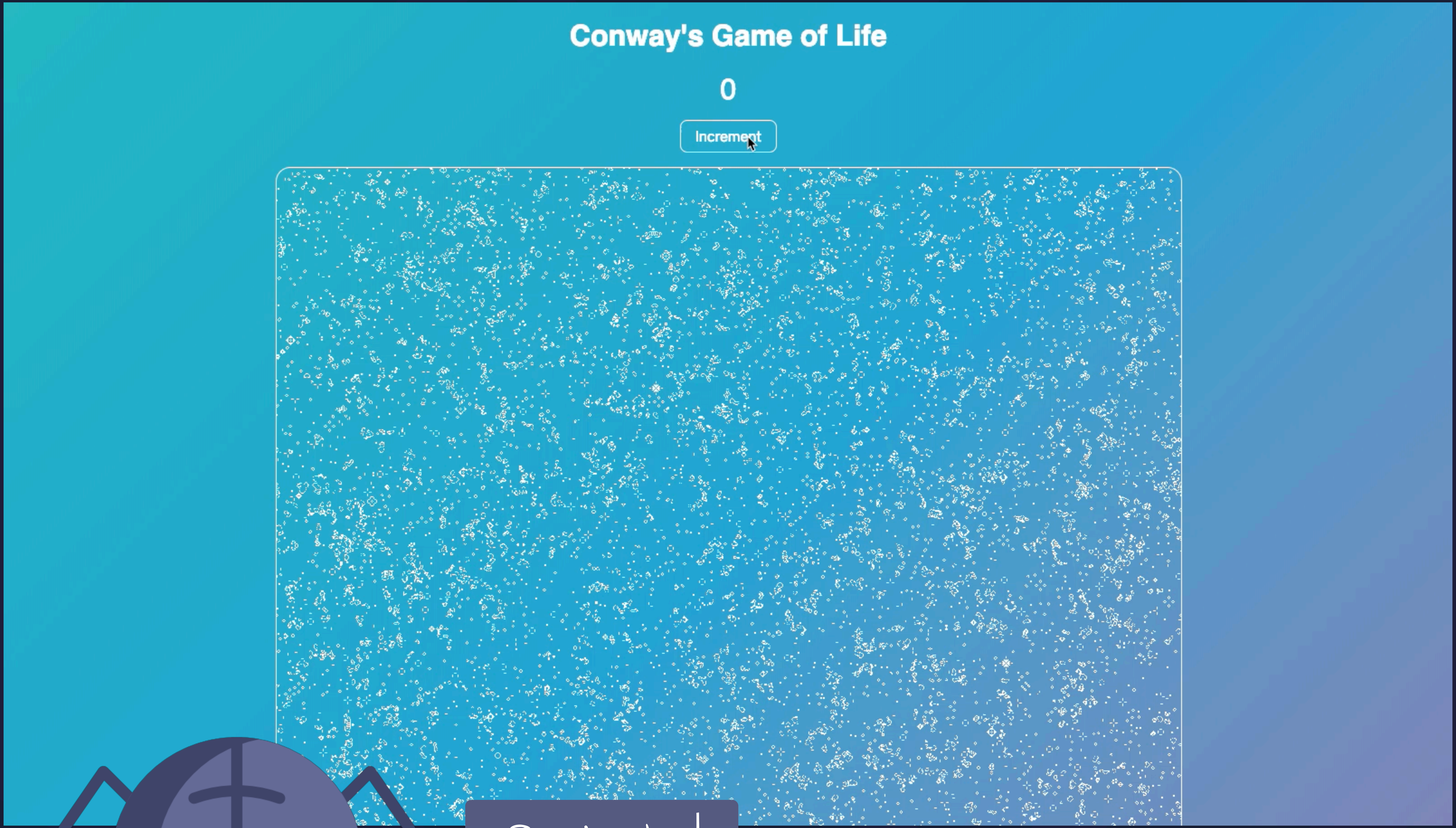
Non-Blocking **DOM** Manipulation

worker-dom

github.com/ampproject/worker-dom

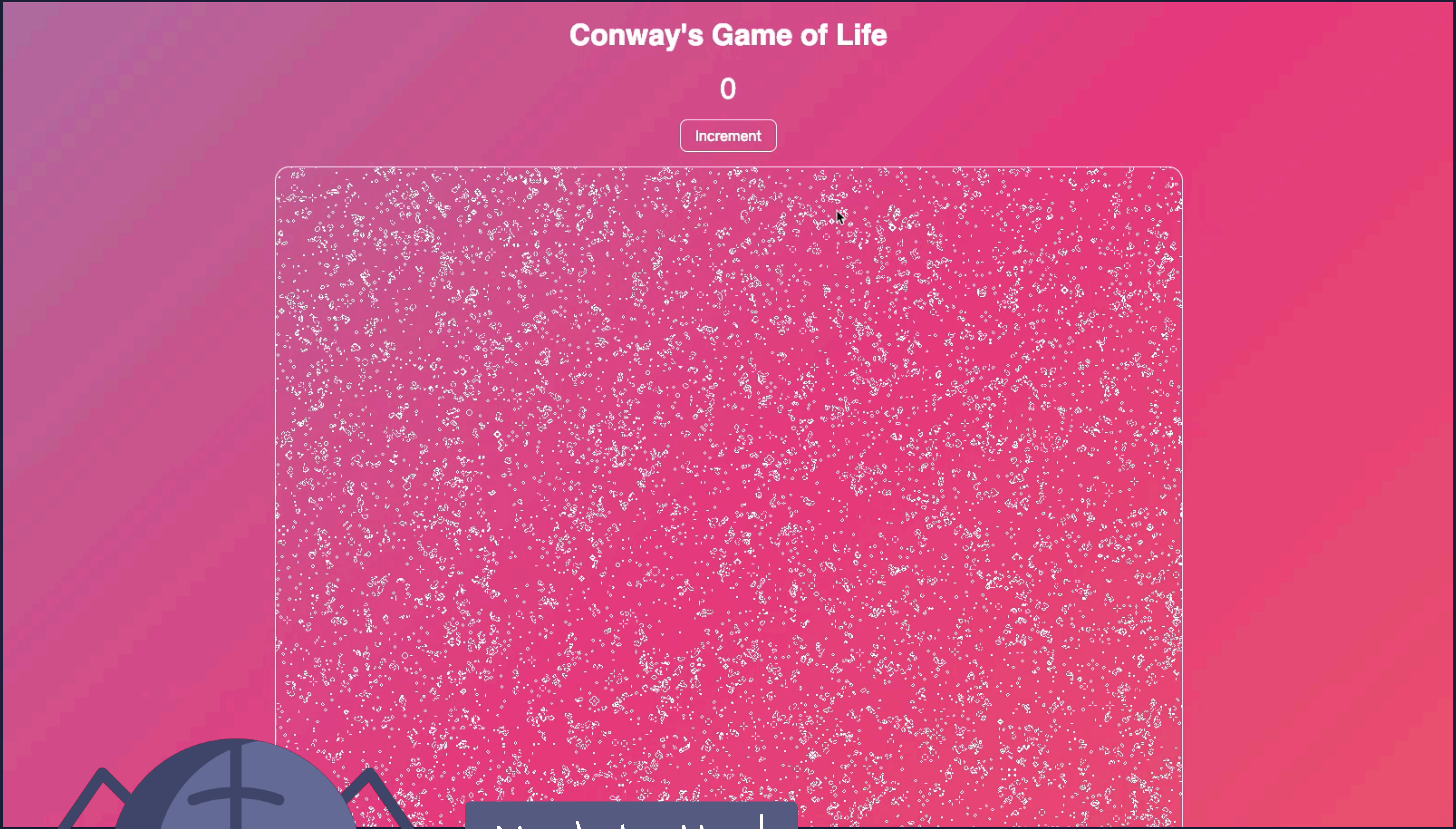
Conway's Game of Life

canvas-of-life.glitch.me


Conway's Game of Life

0

Increment


Conway's Game of Life

0

Increment

@TRENTM

#JSKongress

You Can Do A **LOT** With Web Workers...

PROBLEM Difficult to test

How do we **test** them?

A Tale of **Two Strategies**

PROBLEM Difficult to test

Run your testing framework and worker in **the same thread**

PROBLEM Difficult to test

```html
// Main thread
<script src="test-framework.js"></script>
<script src="worker.js"></script>
<script src="tests.js"></script>
```

PROBLEM Difficult to test

```
// Main thread
<script src="test-framework.js"></script>
<script src="worker.js"></script>
<script src="tests.js"></script>



// Or, worker thread
importScripts('test-framework.js', 'worker.js');
// Your tests here...
```

PROBLEM Difficult to test

That is **NOT** how Workers are used.

PROBLEM Difficult to test

Treat your Worker as a

Function

PROBLEM Difficult to test
SOLUTION Treat your Worker as a **Function**

```
test('transforms data', async (assert) => {
    const worker = new Worker ('transform.js');
    const data = [1, 2, 3];
    const result = postMessage(worker, data);
    assert.equal(result, `I'm transformed!`);
});
```

PROBLEM Difficult to test

SOLUTION Treat your Worker as a **Function**

Sub-Problem: How do we mock/stub calls a Worker?

PROBLEM Difficult to test

SOLUTION Treat your Worker as a **Function**

worker-box

github.com/trentmwillis/worker-box

canvas-of-life.glitch.me/**tests**

Web Workers are powerful

Web Workers are powerful, but avoid using them directly

Web Workers are powerful, but avoid using them directly, instead stand on the shoulders of giants.

Web Workers are powerful, but avoid using them directly, instead stand on the shoulders of giants. There is no better time to start than **right now**.

Resources

- **Spider icon** made by Freepik from www.flaticon.com
- **Web Workers spec**: www.w3.org/TR/workers/
- **Promise Worker**: github.com/nolanlawson/promise-worker
- **Workerize**: github.com/developit/workerize
- **Greenlet**: github.com/developit/greenlet
- **Lumen**: bit.ly/netflix-lumen
- **Worker DOM**: github.com/ampproject/worker-dom
- **Game of Life Demo**: canvas-of-life.glitch.me
- **Worker Box**: github.com/trentmwillis/worker-box