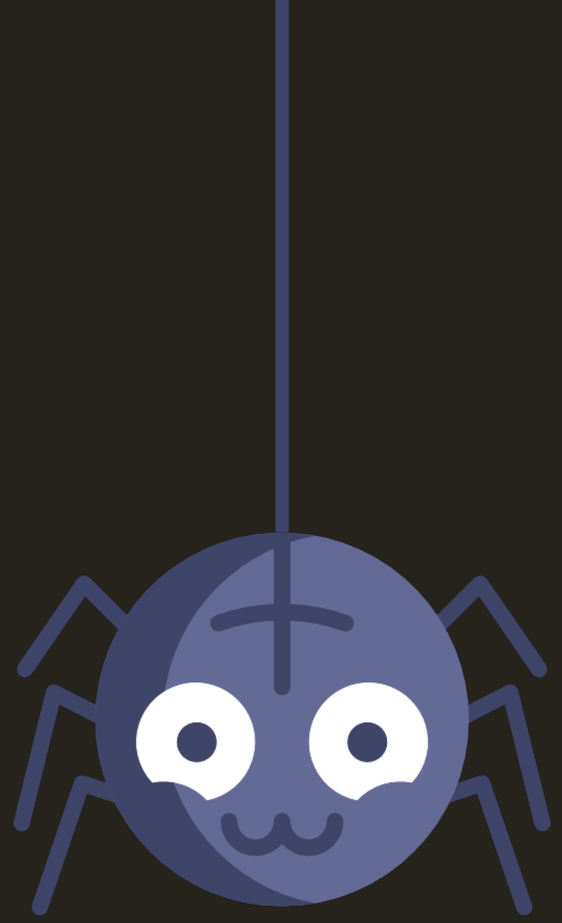
GETTING OUT OF OUR USERS' WAY

LESS JANK WITH WEB WORKERS

#PerfMatters

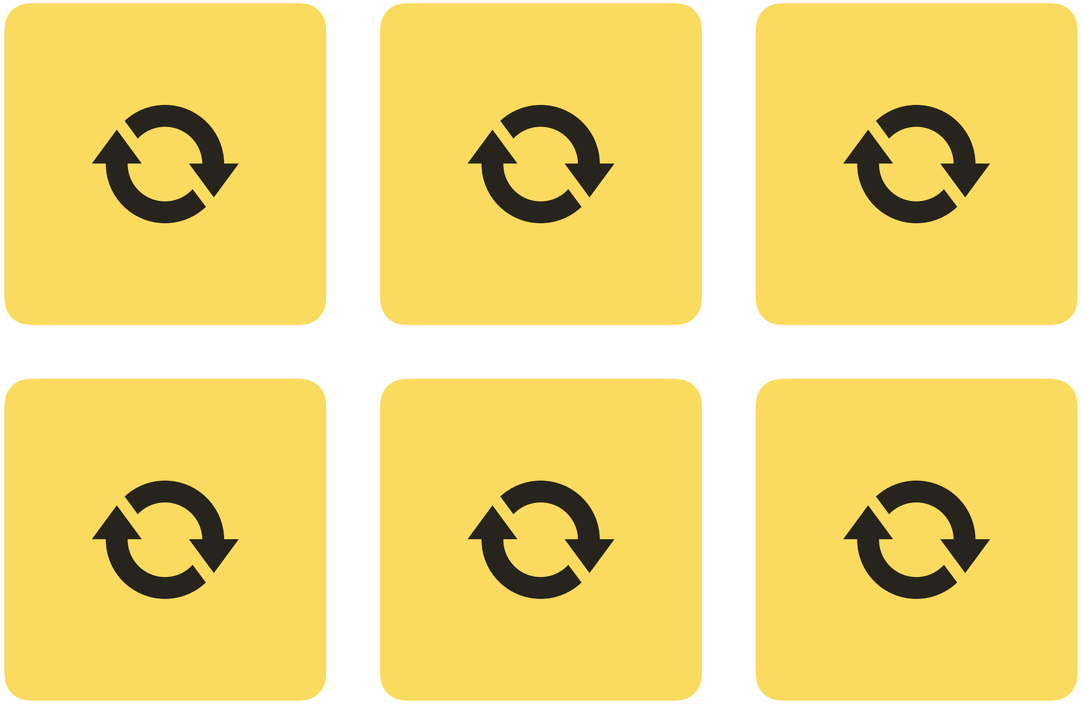
@TRENTMWILLIS



Hello **PerfMatters**!
Think about your web app for a moment...

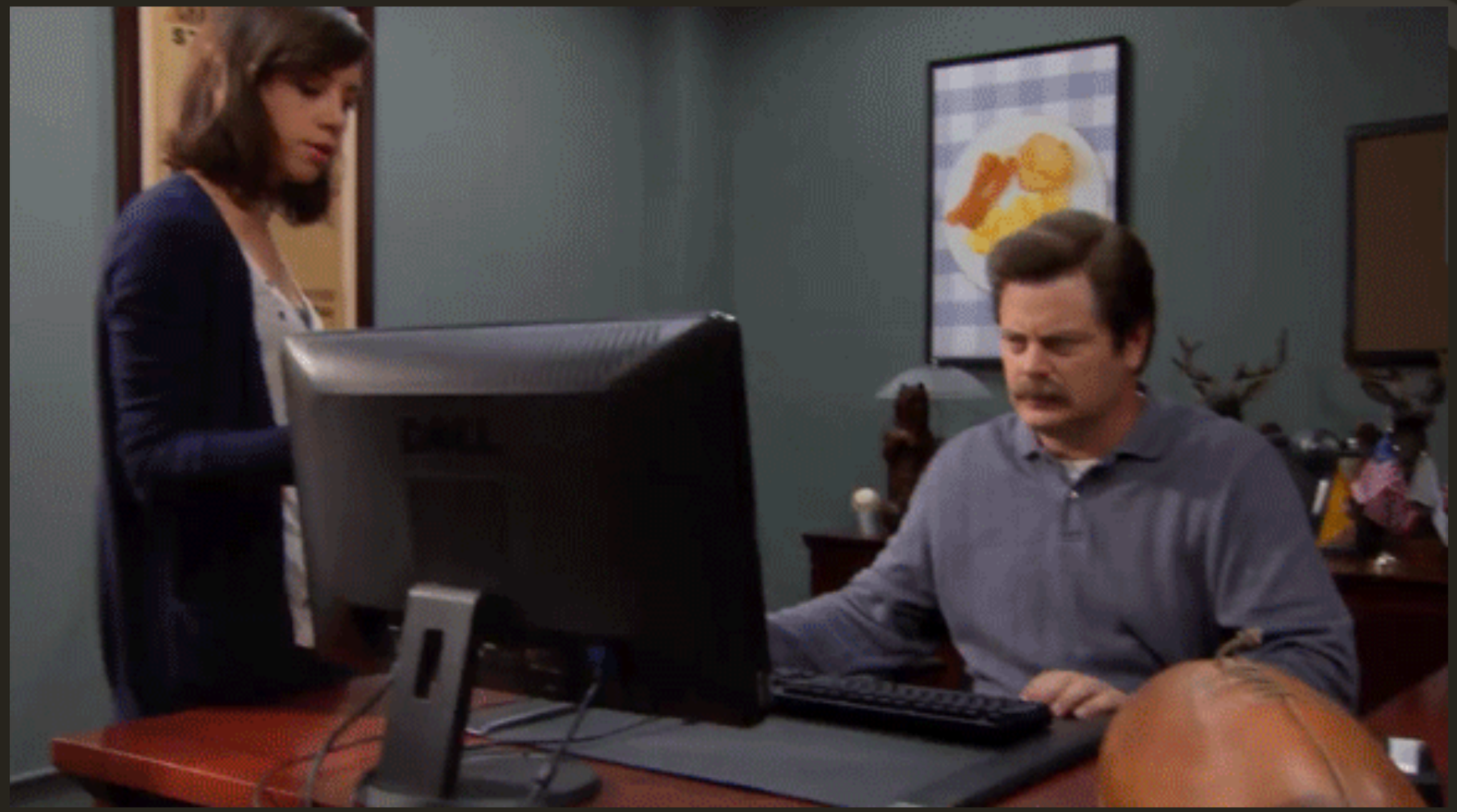
Cool Web App That Pays The Bills

Cool Web App That Pays The Bills



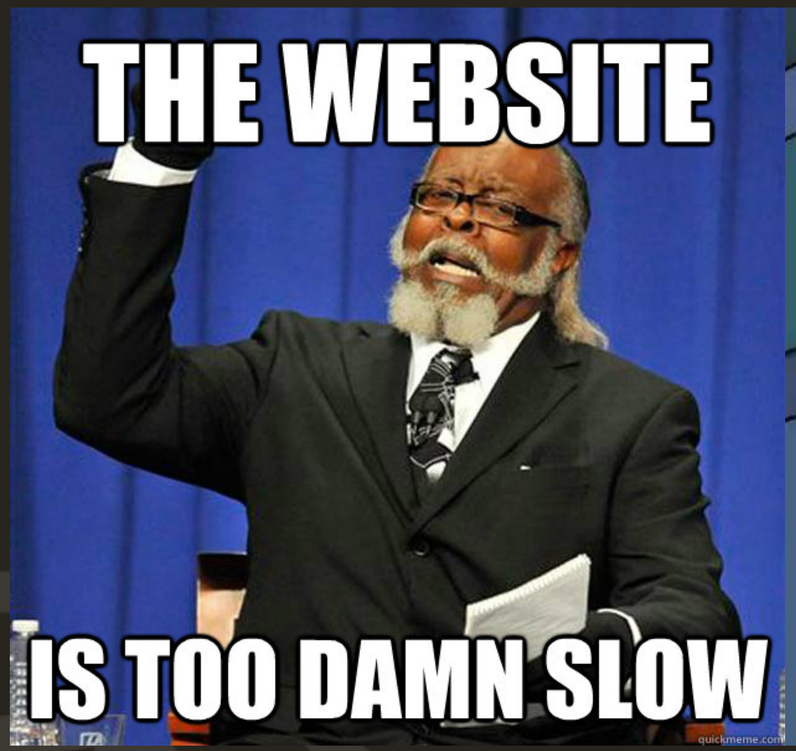
#PerfMatters

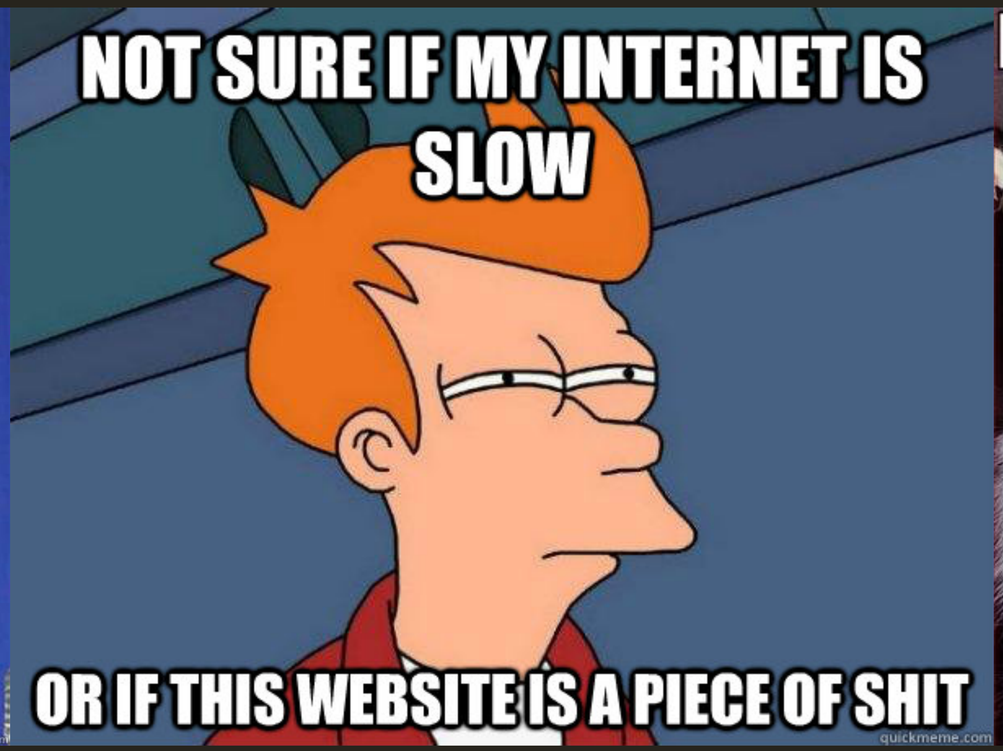
@TRENTMWILLIS

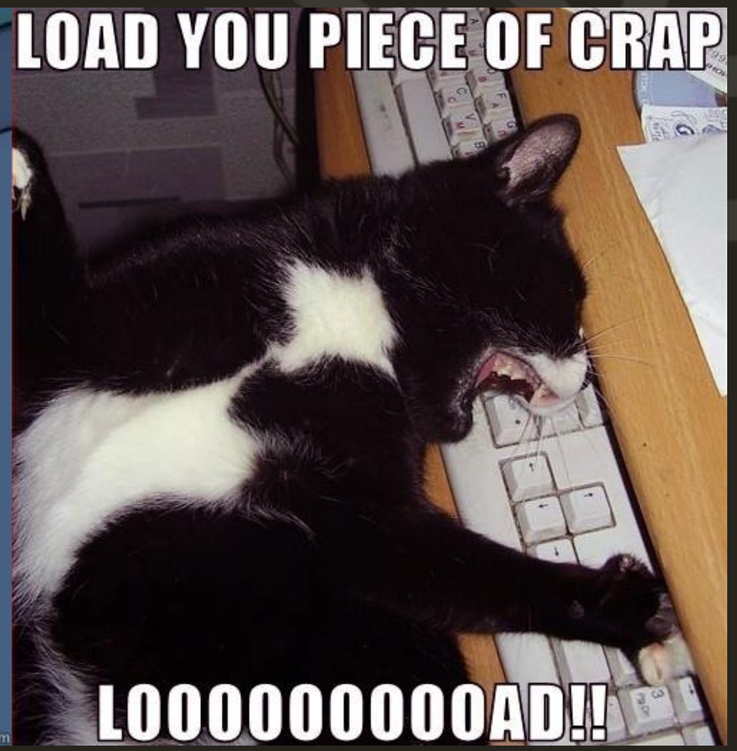


#PerfMatters

@TRENTMWILLIS







#PerfMatters

@TRENTMWILLIS

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

Web Workers

They **help**, but they **complicate**

GETTING OUT OF OUR USERS' WAY

LESS JANK WITH WEB WORKERS

#PerfMatters

@TRENTMWILLIS

@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.postMessage(message);
```

```
// worker thread
self // “window” for a Worker
```

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

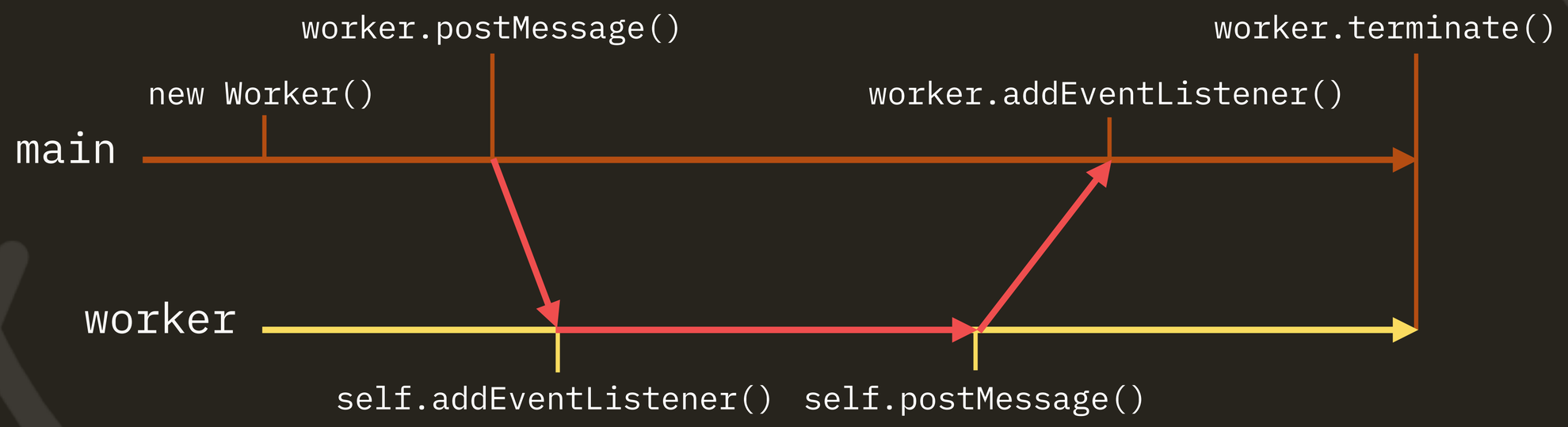
```
// worker thread
self.addEventListener(
  'message',
  event => {
    console.log(event.data);
    self.postMessage(message);
  }
);
```

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

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

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

Web Worker life-cycle



PROBLEMS

Problem: It is hard to know when a worker's task is complete

When am I done?

```
worker.postMessage('doTask');
```

Problem: Workers are difficult to manage and coordinate

```
worker.postMessage('doTask');
otherWorker.postMessage('doOtherTask');
```

How do I manage both results?

```
processResults(
  taskResult,
  otherTaskResult
);
```

Problem: Workers are difficult to test

How do I unit test this?

```
worker.postMessage('doNetworkTask');
```

Problem: Workers are difficult to test

How do I stub the network?

```
worker.postMessage('doNetworkTask');
```

Problem: Workers can not be dynamically defined

If only this was possible...

```
const worker = new Worker(data => {
    // Expensive data operations...
    return processedData;
});
```

SOLUTIONS

Problem: It is hard to know when a worker's task is complete

Solution: Turn messages into Promises

Replace one platform feature with another!

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);
});
```

Solution: Turn messages into Promises

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

Solution: Turn messages into Promises

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

Solution: Turn messages into Promises

promise-worker

github.com/nolanlawson/promise-worker

Problem: Workers are difficult to manage and coordinate

Solution: Use Promises (again)

Problem: Workers are difficult to manage and coordinate

Solution: Expose Worker methods as main thread functions

Solution: Expose Worker methods as main thread functions

```
backendOneWorker
backendTwoWorker
```

Solution: Expose Worker methods as main thread functions

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

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);
```

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);
```

A good worker abstraction looks like any other object!

Solution: Expose Worker methods as main thread functions

Comlink

github.com/GoogleChromeLabs/comlink

Solution: Expose Worker methods as main thread functions

Workerize

github.com/developit/workerize

Solution: Expose Worker methods as main thread functions

importFromWorker

github.com/GoogleChromeLabs/import-from-worker

Problem: Workers can not be dynamically defined

Solution: Create Workers from Blob URLs of functions

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);
};
```

Solution: Create Workers from Blob URLs of functions

greenlet

github.com/developit/greenlet

Web Worker libraries to use

promise-worker -> turn Worker messages into Promises

greenlet -> turn a Function into a Worker

workerize -> turn a Module into Worker

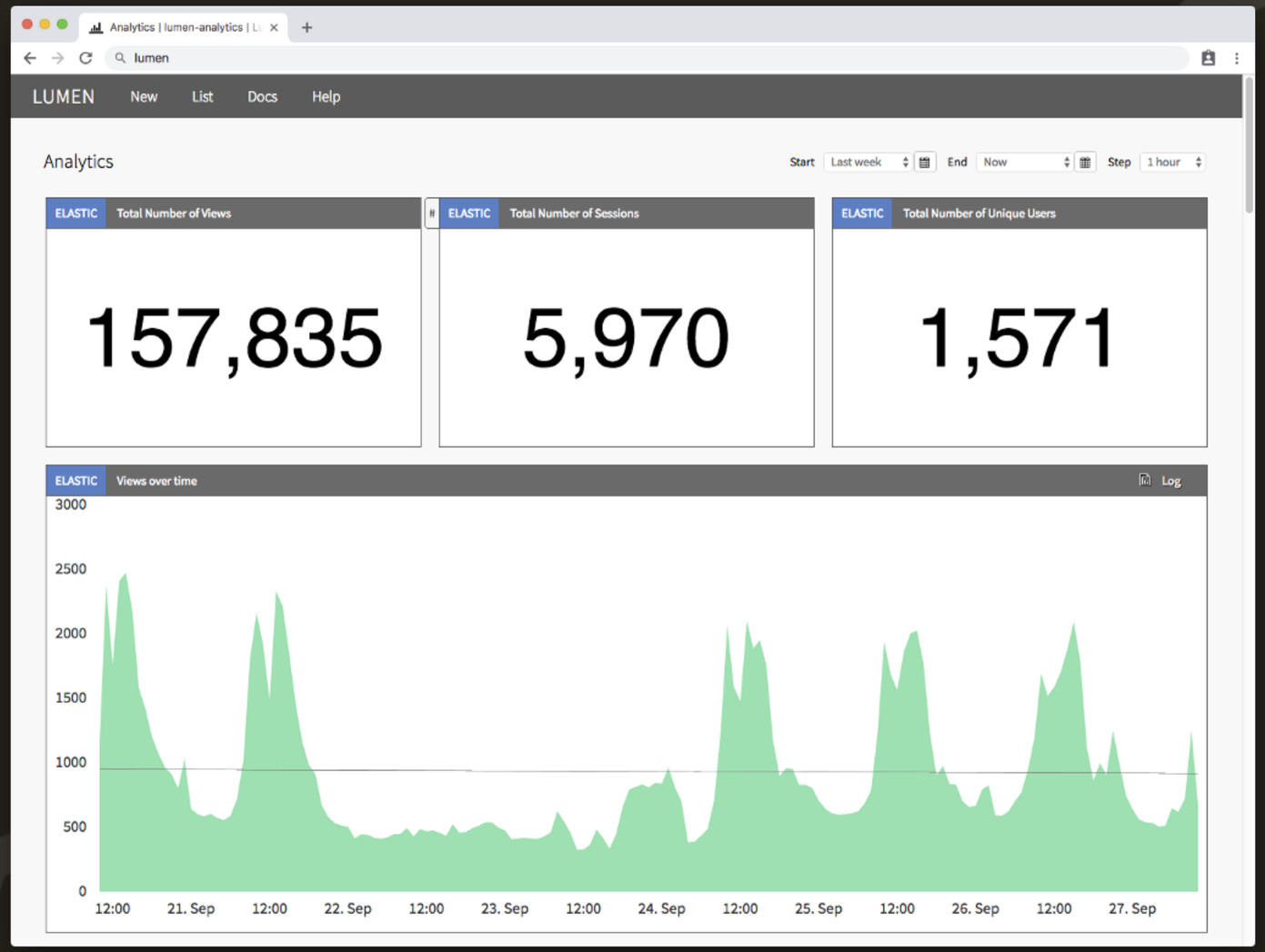
comlink -> give a Worker a nice main thread interface

importFromWorker -> turn a Module import into a Worker

Lumen

bit.ly/netflix-lumen

Lumen



#PerfMatters

@TRENTMWILLIS

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.”

VaporBoy (WASMBoy)

Runs a WASM-based GameBoy emulator
***with Web Workers** for smooth UI*
vaporboy.net

we can "weave" a
web of web workers!

Worker-To-**Worker** Communication

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

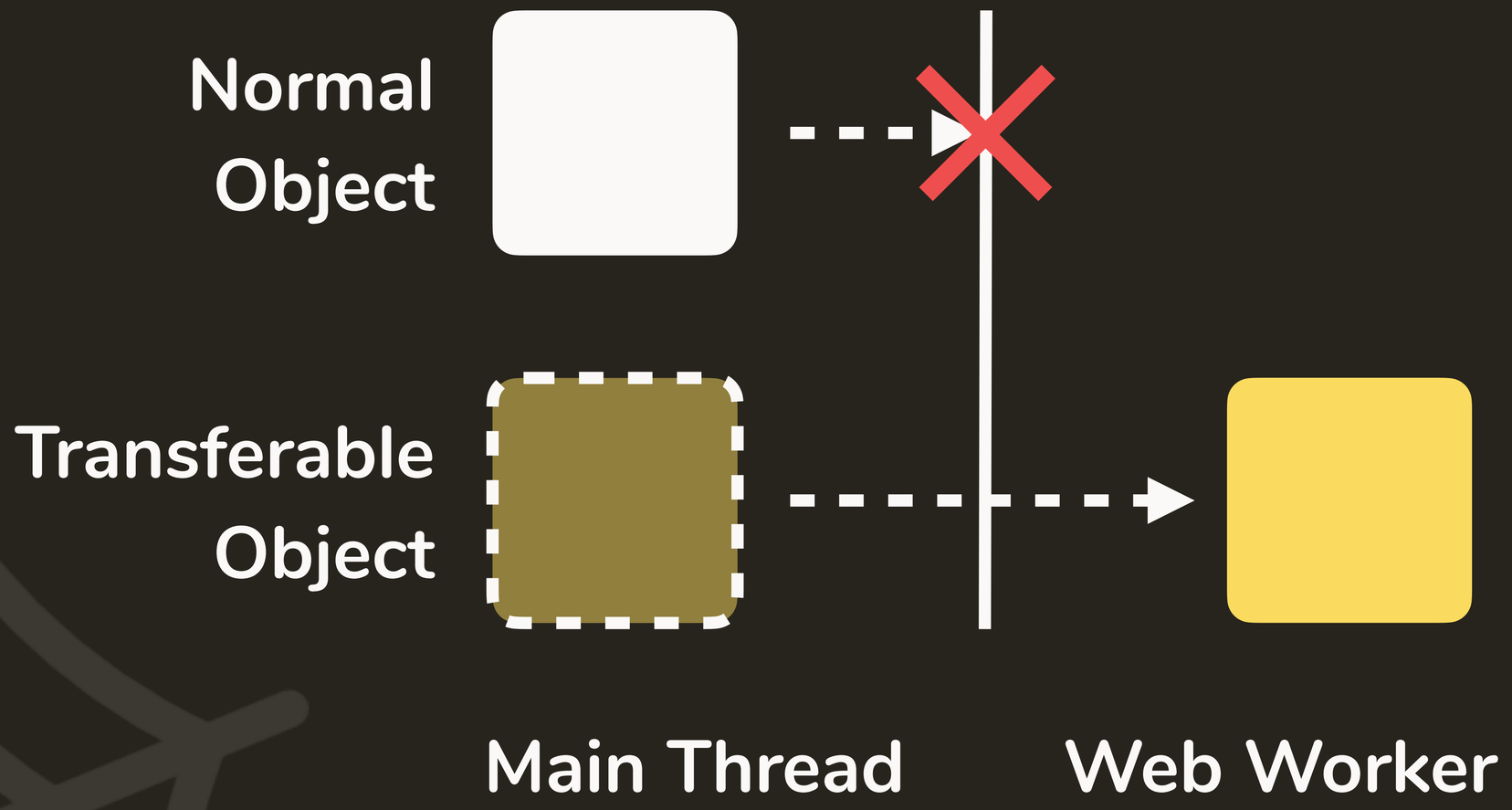
MessageChannel
consists of 2
MessagePorts

```
// 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]);
```

A **Transferable** object can be transferred between execution contexts.



#PerfMatters

@TRENTMWILLIS

```
// 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 API

Allows a <canvas> to be used in a Web Worker

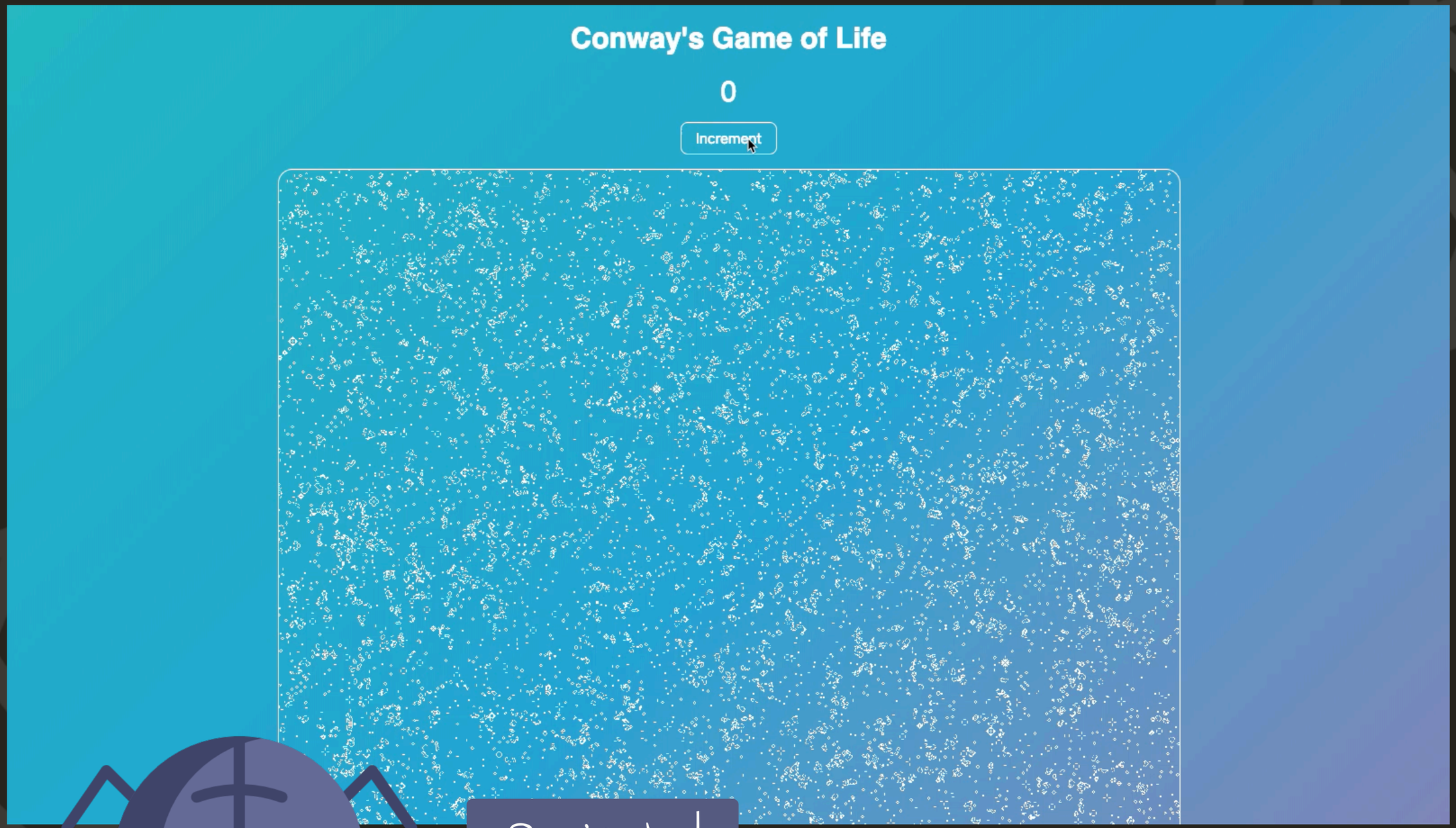
Non-Blocking **DOM** Operations

worker-dom

github.com/ampproject/worker-dom

Conway’s Game of Life

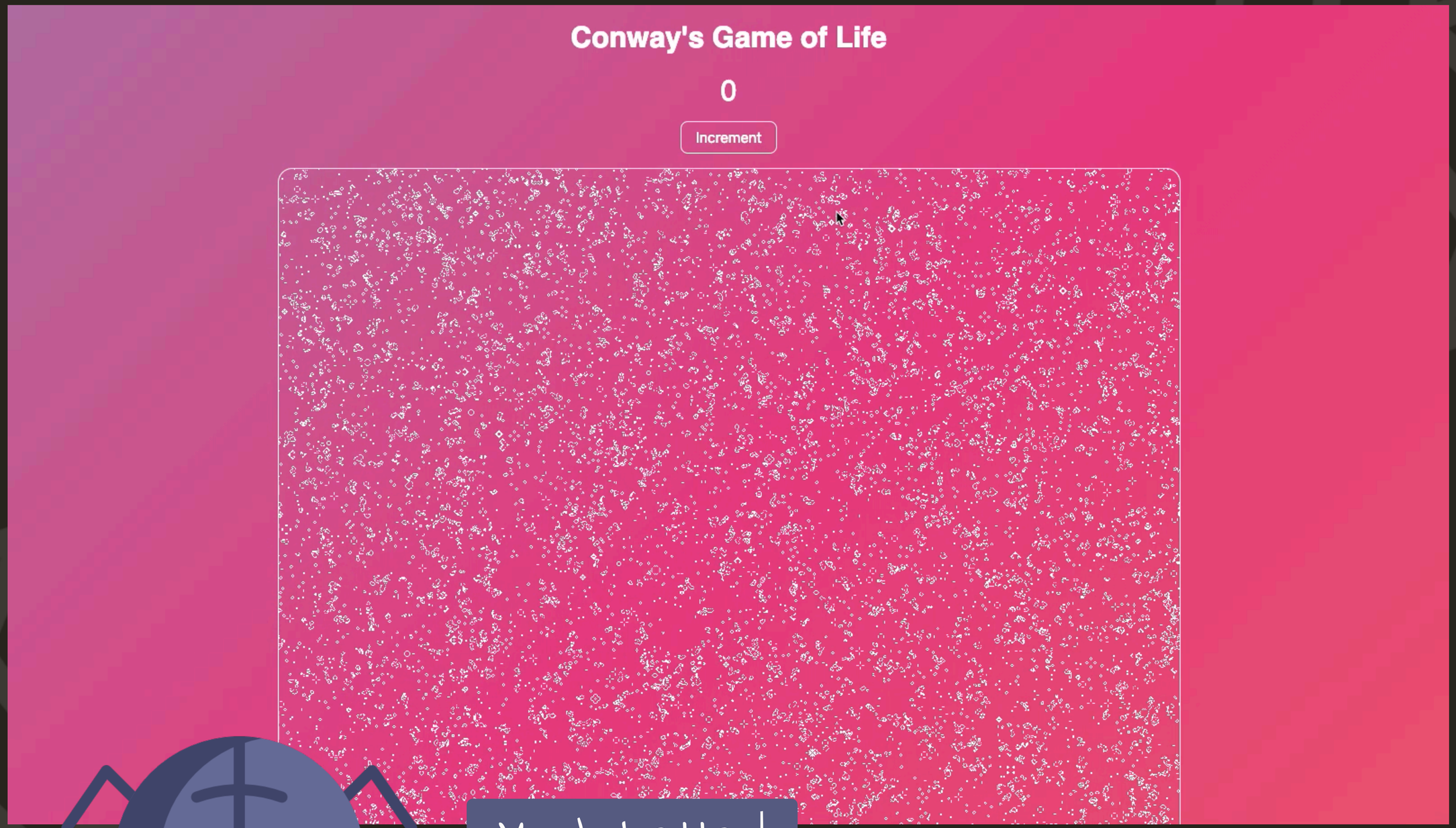
canvas-of-life.glitch.me



So janky!

#PerfMatters

@TRENTMWILLIS



#PerfMatters

@TRENTMWILLIS

You can do a **LOT** with Web Workers, but...

Problem: Workers are difficult to test

How do we **test** them?

Problem: Workers are difficult to test

A Tale of **Two Strategies**

Problem: Workers are difficult to test

Solution #1: Run the testing framework and worker in the same thread

Solution #1: Run the testing framework and worker in the same thread

```
// 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...
```

Solution #1: Run the testing framework and worker in the same thread

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

Problem: Workers are difficult to test

Solution #2: Treat your Worker as a Function

Solution #2: 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!`);
});
```

Solution #2: Treat your Worker as a Function

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

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

worker-box

github.com/trentmwillis/worker-box

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

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

Web Workers are powerful, but avoid using them directly, instead stand on the shoulders of giants. Let's get out of our users' way and give them **better experiences!**

#PerfMatters

@TRENTMWILLIS

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
- **Comlink**: github.com/GoogleChromeLabs/comlink
- **Workerize**: github.com/developit/workerize
- **ImportFromWorker**: github.com/GoogleChromeLabs/import-from-worker
- **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