Researching Wokers

One thing that I've always been curious about is web workers. I know they are not the same as service workers but I don't really know what they are.

So in this post I'll talk about Web Workers (not service workers) and what I have learned about them.

What are Web Workers

Web Workers allow scripts to run in the background, in a separate thread of execution, from the main thread running in the browser. In essence, web workers or workers allow for limited multi threading in Javascript.

What need to they fill

Because JavaScript is single threaded there are times when an app or a page will not be able to execute all the tasks the application needs without degrading the app's overall performance. Using Workers you can delegate the task to a background thread and get the data when the computation ends without slowing down the main thread and any UI work it does.

The Web Workers Section of the WHATWG HTML standard specified two types of workers: dedicated and shared. I'll concentrate on dedicated modules now and do more research on shared workers.

How do they work

Dedicated workers are failry simple. In the host page we use the following script.

```
const supportsWorker = 'Worker' in window;

if (supportsWorker) {
  // Create the worker
  const worker = new Worker('echoWorker.js');
  'echoWorker.js'// Grab a reference to the result div = document.querySe
```

```
  // post message'#result'// post message to worker
  worker.postMessage('<h1>This was sent from the main thread</h1>');

  // This will receive the message from the workerent => {
    result.innerHTML = event.data;
  }
} else {
  console.log('Web Workers not supported');
}
'Web Workers not supported'
```

This script will test if the browser supports dedicated workers and, if supported, initialize the worker and run the script inside it using the `new Worker()` constructor.

Inside `echoWorker.js` we just return the message that we got from the main script

```
onmessage = e => {
  console.log('Echoing message we got from main script');
  postMessage(e.data);
}
```

We can get progressively more complex. In a later section I will explore if we can use a worker to generate HTML from Markdown and then insert it into the parent page.

Longer Example

Using the echo example I got an idea.

Can we use workers to inject different chunks of JSON to fetch Markdown and convert it to HTML before puting the converted HTML into the parent document.

We'll experiment with pulling a markdown file using Fetch API from a Worker to retrieve a Markdown file and then use Remarkable to convert the Markdown file we retrieved to HTML. The final step of the experiment, if it works, is to send the

HTML content back to the origin page where it'll be inserted into the `#result` element.

The script in the host page looks like this:

```
const supportsWorker = 'Worker' in window;

if (supportsWorker) {
  // Create the worker
  const worker = new Worker('./markdownWorker.js');
  './markdownWorker.js'// Grab a reference to the result divument.querySe

  // post message to wo'#result'// post message to worker
  worker.postMessage('./content.md');

  // This will receive the message from the worker
  // and place it inside our result element {
    result.innerHTML = event.data;
  }
} else {
  console.log('Web Workers not supported');
}
'Web Workers not supported'
```

The worker, `markdownWorker`, does a few things that I hadn't done before in a worker. It synchronously imports a third-party script using [importScript] to make sure we have it before we use it.

Inside `onmmessage` we initialize Remarkable, fetch the Markdown file, convert the result as text, process it with Remarkable and post the result back to the "master" page where it will render.

```
importScripts(
  'https://cdn.jsdelivr.net/npm/remarkable@1.7.1/dist/remarkable.js');

self.onmessage = (event) => {
  // Create a remarkable instance and
  // change some default values
```

```
  const md = new Remarkable('full', {
    html: true,
    linkify: true,
    typographer: true,
  });

  'full'// Fetch the URL passed as data for the event
  fetch(event.data)
  .then((response) => {
    // Convert the response to text
    return response.text();
  })
  .then((content) => {
    // Transform the text using the Remarkable
    // instance configured earlier
    let transformedSource = md.render(content);
    // Return the transformed text to the
    // origin pageostMessage(transformedSource);
  })
  .catch((err) => {
    console.log('There\'s been a problem completing your request: ', err)
  });
};
'There\'s been a problem completing your request: '
```

There are a few questions that I still have about the markdownWorker. Some of the questions that I may follow up on later posts:

What's the impact of fetching files from a remote server? To validate the question the worker fetches a file from the same directory it lives in. Will changing it to retrieve a file from a third party directory slow the overall process?

A related question is **how do we make sure that a site's content security policy will not prevent a worker from fetching content?**

Will the worker work faster if it uses async / await? I've seen posts in both directions so I started with Promises. I'm thinking about using async/await in a future iteration to see if it actually works faster.