Web Workers, How, Why, When

Web Workers (workers for short) are a way to create multi-treaded applications in Javascript and get around the language's single threaded application model.

According to Surma:

> Web Workers, also called "Dedicated Workers", are JavaScript's take on threads. JavaScript engines have been built with the assumption that there is a single thread, and consequently there is no concurrent access JavaScript object memory, which absolves the need for any synchronization mechanism. If regular threads with their shared memory model got added to JavaScript it would be disastrous to say the least. Instead, we have been given Web Workers, which are basically an entire JavaScript scope running on a separate thread, without any shared memory or shared values. To make these completely separated and isolated JavaScript scopes work together you have postMessage(), which allows you to trigger a message event in the other JavaScript scope together with the copy of a value you provide (copied using the structured clone algorithm).

So, the idea is that we get a way to run computationally expensive tasks off the main execution thread, improving the application's performance and responsiveness by making the main thread do less work.

In a script/module hosted in the main page, in this example using a script tag, we create a worker and provide functions for it to communicate with the main thread.

```
<script>
const supportsWorker = 'Worker' in window;

if (!supportsWorker) {
    console.log('Web Workers not supported');
```

```
  'Web Workers not supported'// Implement a fallback strategyew Worker('e
    const result = document.querySelector('#result');

    // post message to worker
    'echoWorker.js'// post message to worker
    worker.postMessage('<h1>Message sent to the worker</h1>');

    worker.onmessage = event => {
        result.innerHTML = event.data;
    }
}
</script>
```

echoWorker will send back whatever message it received from the main thread. It will do this by reacting to messages using the onmessage event and posting the content of the event back too the main thread using postMessage.

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

The combination offers interesting posibilities an example that I found interesting is converting Markdown to HTML using a worker.

The code in the main page is similar to the echo example. The main difference is we're sending a Markdown file in the postMessage to the worker and taking the result of processing that file as the result.

```
<div id='result'></div>
<script></div>
  const supportsWorker = 'Worker' in window;

  if (!supportsWorker) {
    // If the browser doesn't support workers bail
    console.log('Web Workers not supported');
  } else {
```

```
    'Web Workers not supported'// Create the workerker('./markdownWorker.j
    const result = document.querySelector('#result');

    // post message to worker w'./markdownWorker.js'// post message to wor
    worker.postMessage('./content2.md');

    worker.onmessage = event => {
      result.innerHTML = event.data;
    }
  }
</script>
```

I've broken the `markdownWorker.js` file into multiple sections to make the explanation easier.

The worker uses third-party resources via [importScripts](). In this case we load the Remarkable Markdown Parser and the Highlight.js syntax highlighter.

```
importScripts(
  'https://cdn.jsdelivr.net/npm/remarkable@1.7.1/dist/remarkable.js',
  'https://cdn.jsdelivr.net/npm/highlightjs@9.16.2/highlight.pack.min.js'
```

Everything that we want to react to happens inside the `onmessage` event.

We first create and configure a Remarkable instance. The highlight configuration uses Highlight.js to insert the hightlighting classes and attributes that will display the highlighted code.

```
self.onmessage = (event) => {
  const md = new Remarkable('full', {
    html: true,
    linkify: true,
    typographer: true,
    // Set highlight options for highlight.js
    highlight: function(str, lang) {
      if (lang && hljs.getLanguage(lang)) {
        try {
```

```
          return hljs.highlight(lang, str).value;
        } catch (err) {}
      }

      try {
        return hljs.highlightAuto(str).value;
      } catch (err) {}

      return '';
    },
  });
''
```

The final element of the worker is a fetch promise chain. The chain does the following:

1. It fetches the file indicated by `event.data`
2. Converts the downloaded data into text
3. Renders the text file into Markdown using the Remarkable instance we defined earlier
4. Sends the data back to the main script using `postMessage`

If there is an error the catch block triggers and we report it to the console.

```
  fetch(event.data)
  .then((response) => {
    // Convert the response to text
    return response.text();
  })
  .then((content) => {
    let transformedSource = md.render(content);
    postMessage(transformedSource);
  })
  .catch((err) => {
    console.log('There\'s been a problem completing your request: ', err)
  });
};
'There\'s been a problem completing your request: '
```

Some things to consider

While workers have access to most APIs and features available to the main thread they are not fully equivalent.

The biggest issue, for me, is that you can't directtly manipulate the DOM from inside Workers (since the global element is `DedicatedWorkerGlobalScope` and not the window global scope)

See Functions and classes available to workers for a list of functions and APIs that work inside a worker.

Links and resources

- MDN Using Web Workers
- When should you be using Web Workers?
- MDN Using Web Workers