Using a web worker to publish Markdown

Inspired by Surma's article When should you be using Web Workers?, I've been looking at ways to use web workers on my projects and I think I've found a good candidate.

I write in Markdown and, until now, have had to rely on the build process to generate HTML files from the Markdown. It works but it requires rebuilding all the files whenever I add new content.

The idea is as follows:

1. The page will create a worker
2. The worker will convert the Markdown to HTML and process syntax hightlight commands
3. The worker will return the processed content to the main page
4. Upon receiving the content from the worker, the main page will insert the returned content inside the div with ID of result

Before we jump into the code, let's review what are workers?

Web Workers or dedicated web workers, are Javascript's way of doing multithreading. They allow developers to run scripts on the background without interrupting the main thread.

You can run whatever code you like inside the worker thread, with some exceptions. For example, you can't directly manipulate the DOM from inside a worker, or use some default methods and properties of the window object. But you can use a large number of items available under window, including WebSockets, and data storage mechanisms like IndexedDB. See Functions and classes available to workers on MDN for more details.

Data is sent between workers and the main thread via a system of messages — both sides send their messages using the postMessage() method, and respond to messages via the onmessage event handler (the message is contained within the Message event's **data** property). The data is copied rather than shared.

Workers may in turn spawn new workers, as long as those workers are hosted

within the same origin as the parent page.

In addition, workers may use XMLHttpRequest for network I/O, with the exception that the responseXML and channel attributes on XMLHttpRequest always return null. They can also use the Fetch API as described in Running fetch in a web worker (*Medium may require login or subscription*)

The host page

The script in the page's body will do the following:

1. Define a constant for deciding if workers are supported
2. Run an if statement for worker support
    1. If it's not supported we log to console
    2. If it is supported, continue
3. Create a new worker using
4. Grab a reference to the result div
5. Pass the name of the file that we want to process to the worker using `postMessage()`
6. Place the result we get from the worker inside the result div using `innerHTML`

```
<div id='result'></div>
<script></div>
  const supportsWorker = 'Worker' in window; // 1

  if (!supportsWorker) {
    console.log('Web Workers not supported'); 'Web Workers not supported'
  } else { // 2b
    // Create the workerker('./markdownWorker.js'); // 3
    // Grab a referenc'./markdownWorker.js'// 3
    // Grab a reference to the result div
    const result = document.querySelector('#result'); // 4

    // post message to worker the file name
    worker.postMessage('./content2.md'); // 5

    // This will receive the message from the
    // worker and place it inside our result
```

```
    // element
    worker.onmessage = event => {
      result.innerHTML = event.data;
    } // 6
  }
</script>
```

The Worker Script

The idea for the worker script is that it will convert the markdown to HTML and will highlight the code inside fenced blocks.

> The specific tasks are as follows:

1. Import Remarkable and HighlightJS from CDN
2. When we receive a message from the main page (using `onmessage`) we create a new instance of Remarkable
3. Inside we also configure Highlight.js to highlight based on the language in the fenced block
4. If that doesn't work then we let Highlight.js autodetect the language
5. If neither named or automatic highlighting work, we return an empty string
6. Fetch the page (the payload is in `event.data`)
7. Transform the content using the Remarkable instance configured earlier
8. Send the converted content back to the main page using `postMessage()`
9. If there is an error log it to console. We're done

```
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'
); 'https://cdn.jsdelivr.net/npm/highlightjs@9.16.2/highlight.pack.min.js

self.onmessage = (event) => {
  const md = new Remarkable('full', { // 2
    html: true,
    linkify: true,
    typographer: true,
    highlight: function(str, lang) { // 3
```

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

      try { // 4      return ''; // 5
    },
  });

  fetch(event.data) // 6
  .then((response) => {
   ''// 5
    },
  });

  fetch(event.data) // 6
  .then((response) => {
    return response.text();
  })
  .then((content) => { // 7
    let transformedSource = md.render(content);
    postMessage(transformedSource); // 8
  })
  .catch((err) => { // 9
    console.log('There\'s been a problem \n
    completing your request: ', err);
  });
};
```

Yes, converting Markdown to HTML is a trivial example but we could make it more complex by running multiple workers to break the content down or run multiple tasks and have them return when they return content.