Understanding and Using Web Workers

Javascript does a lot of things decently enough but one of its biggest drawbacks is that it's single threaded. All scripts that run on a page or application run in the same execution context.

Web Workers provide ones way to work around Javascript single threaded execution model. The idea is that we have two script.

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.

Executing this script, for example, will prevent all other Javascript from executing until the script finished running and loogging all 60000 numbers to the console.

```
i = 0;
while (i < 60000) {
  console.log("The number is " + i);
  i++;
}
```

When working with workers we have two scripts, the main script and the worker.

In the main script we load the worker like this:

```
if (typeof(Worker) !== "undefined") {
    worker = new Worker("worker.js");
}
```

In the worker script we can do the heavy loading script without interrupting the main thread execution.

```
i = 0;
while (i < 200000) {
    postMessage("Web Worker Counter: " + i);
    i++;
}
```

It's not all rosy though. Web Workers have quite a few limitations:

- no access to the DOM: the Window object and the Document object are not available
- they can communicate back with the main JavaScript program using **messaging**
- they need to be loaded from the same origin (**domain, port and protocol**)
- they don't work if you serve the page using the file protocol (**file://**)

An important note is that the global scope of a Web Worker, instead of Window on the main thread, is a WorkerGlobalScope object.

Communication Between Workers and Main Thread

There are two main ways to communicate to a Web Worker:

- the postMessage API offered by the Web Worker object
- the Channel Messaging API

In the main.js script we use the worker's `postMessage` method to pass the message we want to send.

There is a second optional parameter that contains an array of Transferable objects to transfer ownership of. If the ownership of an object is transferred, it becomes unusable (neutered) in the context it was sent from and becomes available only to the worker it was sent to.

Transferable objects are instances of classes like `ArrayBuffer`, `MessagePort` or `ImageBitmap` objects that can be transferred. `null` is not an acceptable value for transfer.

main.js

```
const worker = new Worker('worker.js')
worker.postMessage('hello')
```

The worker script registers success and error messages that will be sent to the main script when the respective event triggers.

worker.js

```
onmessage = (event) => {
  console.log(event.data)
}

onerror = (event) => {
  console.error(event.message)
}
```

We can push messages from the worker back to the main script. You can have multiple event listeners asnwering with different messages to what the main script sends.

worker.js

```
onmessage = (event) => {
  console.log(event.data)
  postMessage('hey')
}
```

```
onerror = (event) => {
  console.error(event.message)
}
```

The main script posts messages to the worker with the data that we want the worker to use.

main.js

```
const worker = new Worker('worker.js')
worker.postMessage('hello')

worker.onmessage = (event) => {
  console.log(event.data)
}
```

Channel API: Another way to communicate with workers

There is another way to have workers and host pages, the Channel Messaging 7API.

The Channel Messaging API is a message bus that allows multiple channels to communicate with each other.

In the example below the main script listens on port1 and posts messages to the worker in port2.

main.js

```
const worker = new Worker('worker.js')
const messageChannel = new MessageChannel()

messageChannel.port1.addEventListener('message', (event) => {
  console.log(event.data)
})

worker.postMessage(data, [messageChannel.port2])
```

The worker script listens to messages and logs the results to console.

worker.js

```
addEventListener('message', (event) => {
  console.log(event.data)
})
```

A Web Worker can send messages back by posting a message to messageChannel.port2, like this:

```
addEventListener('message', (event) => {
  event.ports[0].postMessage(data)
})
```

Loading libraries in a Web Worker

Web Workers load scripts using the `importScripts()` global function defined in their WorkerGlobalScope object. The first line of your worker script should import the scripts (one or more) you need in your worker

```
importScripts('../utils/file.js',
  './something.js')
```

APIs available in Web Workers

Web Workers cannot access the DOM so you cannot interact with the window and document objects.

All is not gloom, you can use many other APIs like:

- XHR or Fetch API
- BroadcastChannel API
- FileReader API
- IndexedDB
- Notifications API
- Promises
- Service Workers
- Channel Messaging API
- Cache API
- Console API (console.log() and friends)
- JavaScript Timers (setTimeout, setInterval…)
- CustomEvents API: addEventListener() and removeEventListener()
- current URL, which you can access through the location property in read mode
- WebSockets
- WebGL
- SVG Animations

So: When Do We Use Web Workers

With all the limitations there are still plenty of things you can do with workers.

The first thing that comes to mind is to take those computationally heavy tasks off the main thread and make your application more responsive.

We are not limited to strings. While not an easy API to use it allows more transfer more complex data structures like array and image buffers.

I'm just starting too experiment with Workers and it's definitely an interesitng area of research.

Links And Resources

- When should you be using web workers
- Web workers vs Service workers vs Worklets — Ire Aderinokun
- Web Workers — Flavio Copes
- JavaScript Web Workers: A Beginner's Guide
- The Basics of Web Workers — HTML5 Rocks
- Using Web Workers — MDN
- Web Workers API — MDN