Web workers vs Service workers vs Worklets

Nov 20, 2018 *javascript*

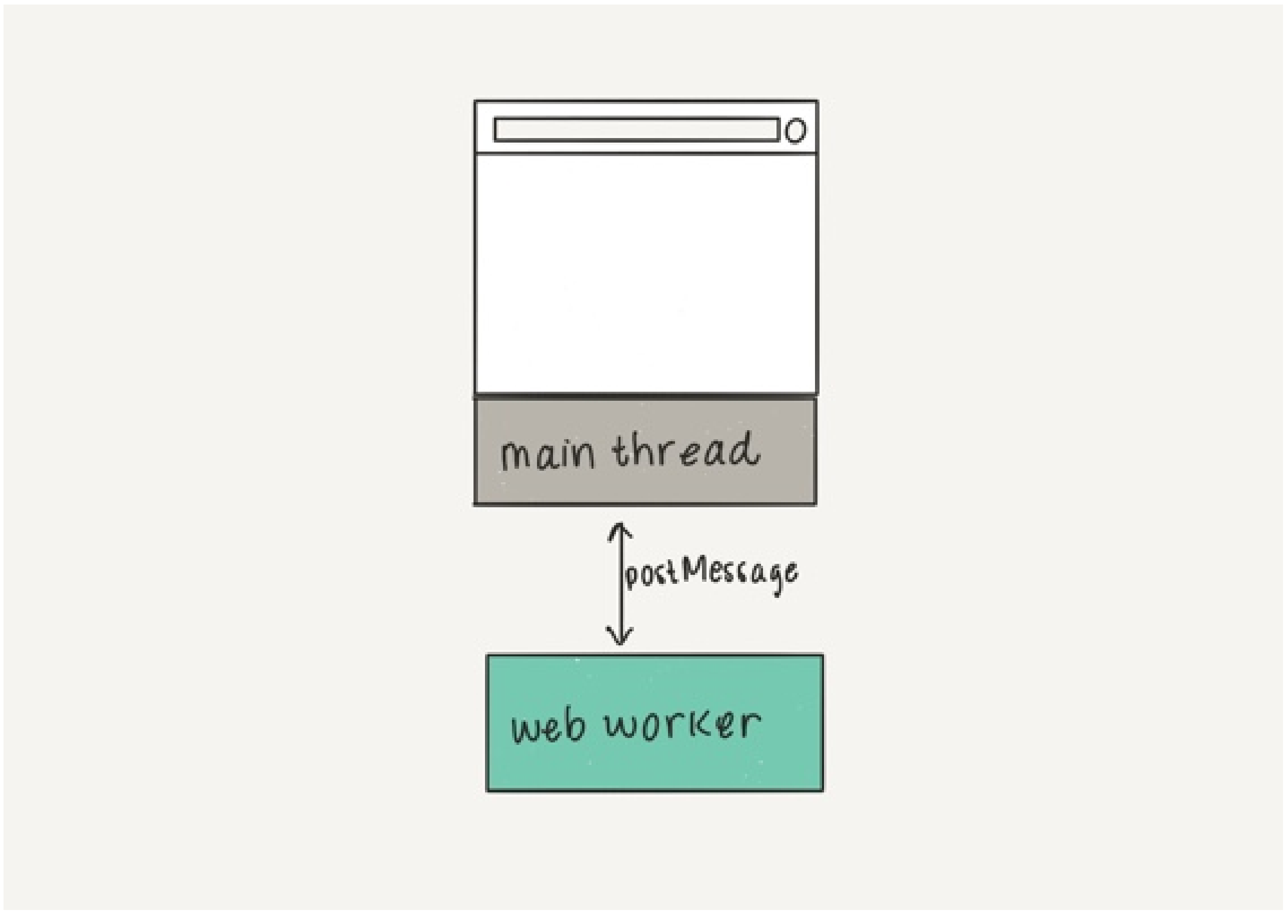
Web workers, service workers, and worklets. All of these are what I would call "Javascript Workers", and although they do have some similarities in how they work, they have very little overlap in what they are used for.

Broadly speaking, a worker is a script that runs on a thread separate to the browser's main thread. If you think about your typical Javascript file that is included in your HTML document via a `<script>` tag, that runs on the main thread. If there is too much activity on the main thread, it can slow down the site, making interactions jittery and unresponsive.

Web workers, service workers, and worklets are all scripts that run on a separate thread. So what are the differences between these three types of workers?

Web workers

Web workers are the most general purpose type of worker. Unlike service workers and worklets as we will see below, they do not have a specific use case, other than the feature of being run separately to the main thread. As a result, web workers can be used to offload pretty much any heavy processing from the main thread.



Web workers are created using the **`Web Workers API`**. After creating a dedicated Javascript file for our worker, we can add it as a new **`Worker`**.

```
/* main.js */

const myWorker = new Worker('worker.js');
```

This will start to run whatever code we have in the **`worker.js`** file. As I mentioned, this can be almost anything, but web workers are most useful for offloading processes that might take a long time or are run in parallel to other processes. A great example is the image processing web application, Squoosh, which uses web workers to handle image manipulation tasks, leaving the main thread available for the user to interact with the application without interruption.

Like all workers, web workers do not have access to the DOM, which means that any information needed will have to be passed between the worker and the main script using **`window.postMessage()`**.

```
/* main.js */

// Create worker
const myWorker = new Worker('worker.js');

// Send message to worker
myWorker.postMessage('Hello!');

// Receive message from worker
myWorker.onmessage = function(e) {
  console.log(e.data);
}
```

In our worker script, we can listen for messages from the main script, and return a response.

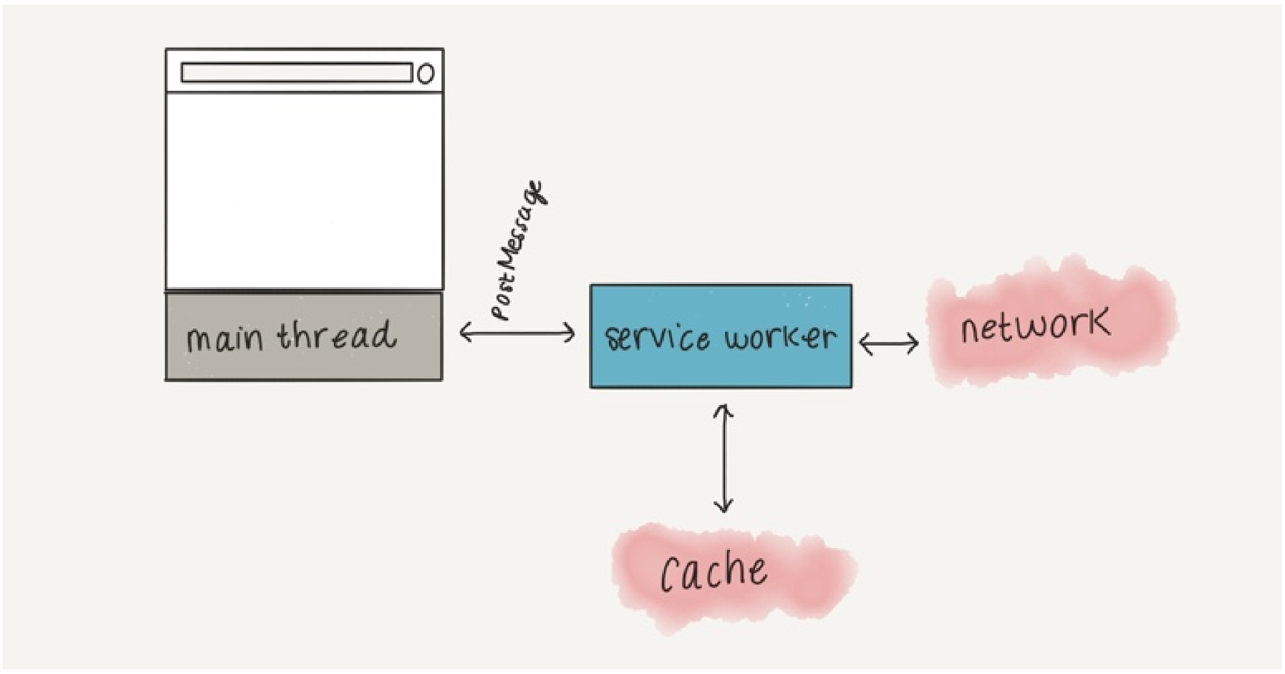
```
/* worker.js */

// Receive message from main file
self.onmessage = function(e) {
  console.log(e.data);

  // Send message to main file
  self.postMessage(workerResult);
}
```

Service workers

Service workers are a type of worker that serve the explicit purpose of being a proxy between the browser and the network and/or cache.



Like web workers, service workers are registered in the main javascript file, referencing a dedicated service worker file.

```
/* main.js */

navigator.serviceWorker.register('/service-worker.js');
```

Unlike regular web workers, service workers have some extra features that allow them to fulfil their proxy purpose. Once they are installed and activated, service workers are able to intercept any network requests made from the main document.

```
/* service-worker.js */

// Install
self.addEventListener('install', function(event) {
    // ...
});

// Activate
self.addEventListener('activate', function(event) {
    // ...
});

// Listen for network requests from the main document
self.addEventListener('fetch', function(event) {
    // ...
});
```

Once intercepted, a service worker can, for example, respond by returning a document from the cache instead of going to the network, thereby allowing web applications to function offline!

```
/* service-worker.js */

self.addEventListener('fetch', function(event) {
    // Return data from cache
    event.respondWith(
        caches.match(event.request);
    );
});
```

Worklets

Worklets are a very lightweight, highly specific, worker. They enable us as developers to hook into various parts of the browser's rendering process.

When a web page is being rendered, the browser goes through a number of steps. I cover this in more detail in my article on Understanding the Critical Rendering Path, but there are four steps we need to worry about here - Style, Layout, Paint, & Composite.

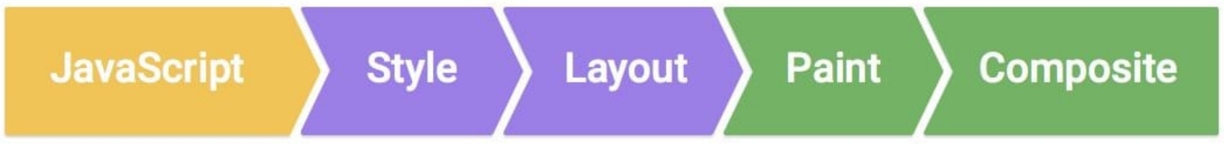


Image credit: Rendering Performance, by Paul Lewis

Let's take the Paint stage. This is where the browser applies the styles to each element. The worklet that hooks into this stage of rendering is the Paint Worklet.

The Paint Worklet allows us to create custom images that can be applied anywhere CSS expects an image, for example as a value to the **`background-image`** property.

To create a worklet, as with all workers, we register it in our main javascript file, referencing the dedicated worklet file.

```
/* main.js */

CSS.paintWorklet.addModule('myWorklet.js');
```

In our worklet file, we can create the custom image. The **`paint`** method works very similarly to the Canvas API. Here's an example of a simple black-to-white gradient.

```
/* myWorklet.js */

registerPaint('myGradient', class {
  paint(ctx, size, properties) {
    var gradient = ctx.createLinearGradient(0, 0, 0, size.height / 3);

    gradient.addColorStop(0, "black");
    gradient.addColorStop(0.7, "rgb(210, 210, 210)");
    gradient.addColorStop(0.8, "rgb(230, 230, 230)");
    gradient.addColorStop(1, "white");

    ctx.fillStyle = gradient;
    ctx.fillRect(0, 0, size.width, size.height / 3);
  }
});
```

Finally, we can use this new worklet in our CSS, and the custom image we created will be applied like any other background image.

```
div {
    background-image: paint(myGradient);
}
```

In addition to the Paint Worklet, there are other worklets that hook into other stages of the rendering process. The Animation Worklet hooks into the Composite stage, and the Layout Worklet hooks in to the Layout stage.

Recap

To recap, web workers, service workers, and worklets are all scripts that run on a separate thread to the browser's main thread. Where they differ is in where they are used and what features they have to enable these use cases.

Worklets are hooks into the browser's rendering pipeline, enabling us to have low-level access to the browser's rendering processes such as styling and layout.

Service workers are a proxy between the browser and the network. By intercepting requests made by the document, service workers can redirect requests to a cache, enabling offline access.

Web workers are general-purpose scripts that enable us to offload processor-intensive work from the main thread.

Subscribe to the Newsletter

Receive quality articles written by Ire Aderinokun, frontend developer ce designer.

Email Address*

First Name

Subscribe

←

You might not need a loop

javascript

→

Why and how to use WebP images today

html, *performance*
