

# Web Workers

A web worker is a generic interface for threading inside the browser. They are useful for heavy tasks, usually involving data or calculations. It's useful for parsing data, sorting, concatenating, like creating an HTML based on a big set of data. Also, it's useful for binary manipulation, such as, for example, if you want to take the raw data from the camera and do a QR code reader, or if you want to recognize characters, or if you want to recognize users' faces.

All that image manipulation can be done in a worker. It can also be used for a WebSocket centralized manager, so if you have a WebSocket connection to our server, we can centralize all the communication in a web worker, and it can also be used for gaming, typically for a game loop because it's a thread, and then we are not attaching the original thread, the UI thread. Being a generic interface for thread in the operating system means that there are several ways to create the worker.

So we have a generic interface in JavaScript, known as the web worker, but then we have a specific kind of workers that we will see in a minute. Usually, their creation is expensive, so we don't want to create thousands of workers. Also, we don't want to create workers in the loop. And we don't want to create workers, terminate those workers and then create a new one, terminate, create, terminate. We want to reuse workers. So just create one and reuse that one.

Because it's a thread separate from the main thread, it has no access to the user interface, so no DOM, no document get to them by ID, no alert, nothing that has to do with the user interface. And because there is no access to the user interface, we need a way for the main thread to update the UI based on our calculations. So we have a messaging API between the main thread and the worker. There are four kinds of workers today.

1. Dedicated Worker(Owned by the document)
2. Shared Worker(Shared by different documents)
3. Service Worker(Owns a scope, network proxy)
4. Audio Worker(For web audio processing)

## 1. Dedicated Worker (Owned by the document):

We have a dedicated worker, that's the most typical example of a web worker. In this case, it's a thread that is owned by the document who created it. So we have an HTML, we have a window, a tab, or perhaps a web app. That HTML with JavaScript

created a worker. So that's a dedicated worker for that particular instance of our web app. If you open a new tab, that new tab on the same URL will have another dedicated worker for it.

## 2. Shared Worker (Shared by Different documents)

Another kind of worker is known as the shared worker. In this case, the shared worker is shared by different documents. So for example, if you have Google Drive or your email opened, and you open several tabs from the same domain, for example, from Google Drive, all of them can share a worker. So that's a shared worker. It's one thread that can be used and reused by several tabs on the same domain.

## 3. Service Worker (Owns a scope and network)

In this case, it owns a scope, so the whole domain or folder in your domain, and it will act as a network proxy.

## 4. Audio Worker (For web audio processing)

We also have AudioWorklets, and that's a thread only for web audio processing and can be used with the web audio API to process audio coming from the microphone or going to the headphones.

The web worker API, the original spec, only includes the dedicated worker and shared worker, but not every browser is supporting shared workers. Dedicated workers are available everywhere. And for shared workers, service workers, and AudioWorklet, we need to check before using them if they are available on that particular device on that particular browser. So each web worker will be a thread in the operating system. Each web worker can contain their own tasks, and the main thread will also contain their own tasks.

But remember, tasks, concurrent tasks, are not being executed in parallel within the same thread. If you have ever worked with threads on a different platform, such as Java, .NET, native apps, this is much simpler. Web worker has a simplified threading operation version, so there are no locks, no semaphores, or similar patterns to use because there is no access to the DOM or any synchronous API. There are no risks of trying to access the same data at the same time.

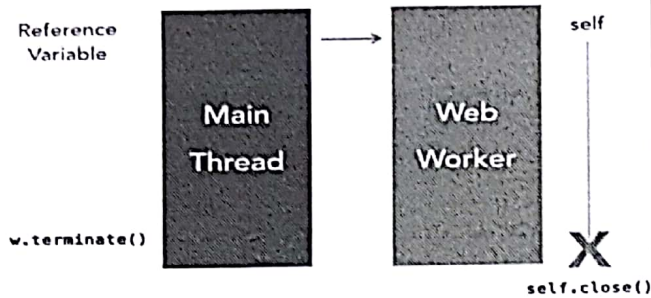
A web worker can create subtasks, so using `setTimeout`, `setInterval`, so we can have several tasks that they will run concurrently, but not in parallel within the same thread. A service worker can use XML, HTTP request, or `Hx`, `Fetch`, `WebSockets`, `JSON`, and `IndexedDB`. We don't have access to other APIs, we don't have access to user interface, geolocation, sensors, we don't have access, for example, for local storage because that's a synchronous API, and that will require a lock.



## Create a Dedicated Worker:

```
const worker = new Worker("worker.js",{  
  name:"debug-name", //new version browsers only support  
  type:"classic/module",  
  credentials:"omit|same-origin|include"  
});
```

Life cycle of Dedicated worker:



### Termination

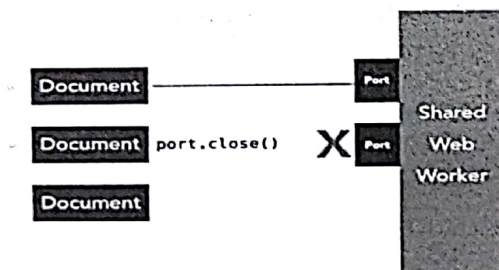
The main thread or the worker can terminate it; also, closing the window owning the thread will cause a termination

Dedicated workers should be reside on same origin.

## Create a shared worker:

```
const w = new SharedWorker("shworker.js",{  
  name:"debug-name"  
});
```

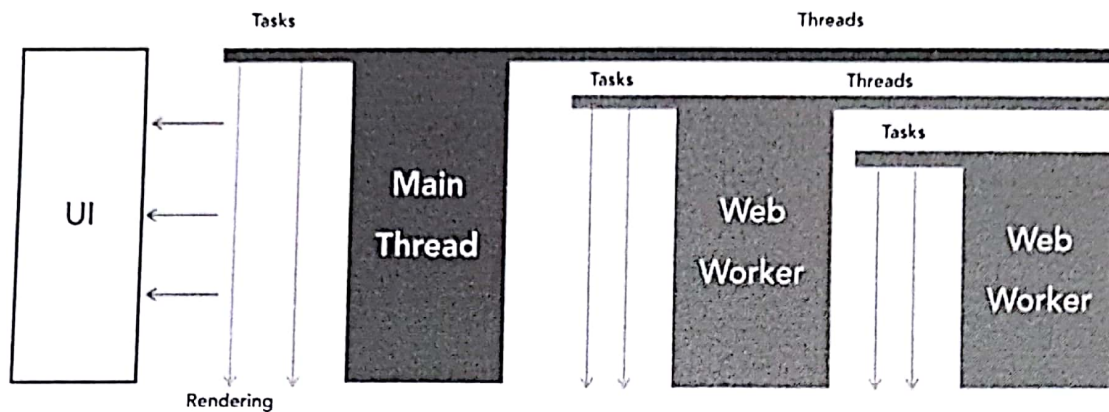
Life cycle of shared worker:



### Life Cycle of a Shared Web Worker

Each document can call `close()` on the port, but there is no terminate method

## Creating sub workers (Workers with in workers)



## Subworkers

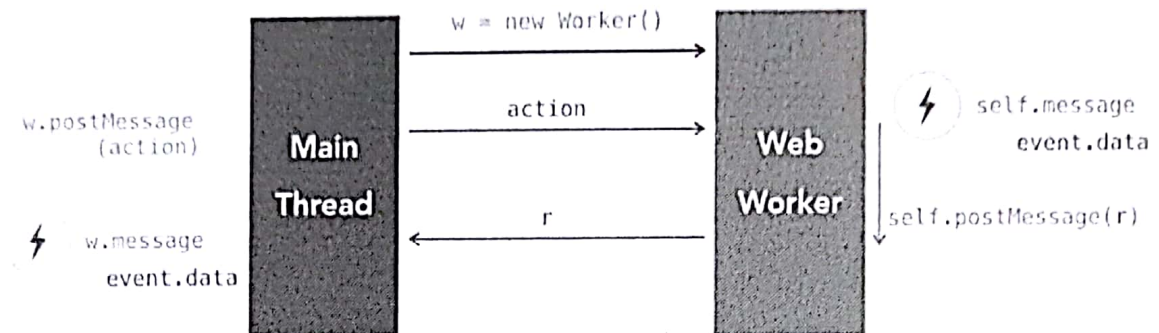
### Import external scripts:

```
//In aworkers scope  
  
importScripts("script.js");//use ofr single file import  
  
importScripts("script1.js","script2.js","script3.js");//we can import more than  
one also
```

- It's a sync API
- You can call it once with several files or several times at different parts of the code.
- If you want to use es6 modules, move from classic to module type when registering.
- If you are loading the worker with an inline technique (blob), be careful with importScripts
- Always use absolute url's as scripts will not be relative to your pages url ,but to the workers url that is blob based.



Send messages to dedicated workers:



## Reusable worker

Send and receive messages via dedicated works:

```
//server.js
var express = require("express");
var path = require("path");
var bodyParser = require("body-parser");
var app = express();

console.log(__dirname);
app.use(express.static(path.join(__dirname, "../views")));
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.listen(4200);
console.log("server is listening on port 4200");
```

```
//worker.js
self.addEventListener("message", event=>{
  if(event.data.action === "start"){
    console.log("Hi I am in worker");
    start();
  }
})
function start(){
  self.postMessage({
    action: "output",
```



```

    data:"worker code"
  })
}

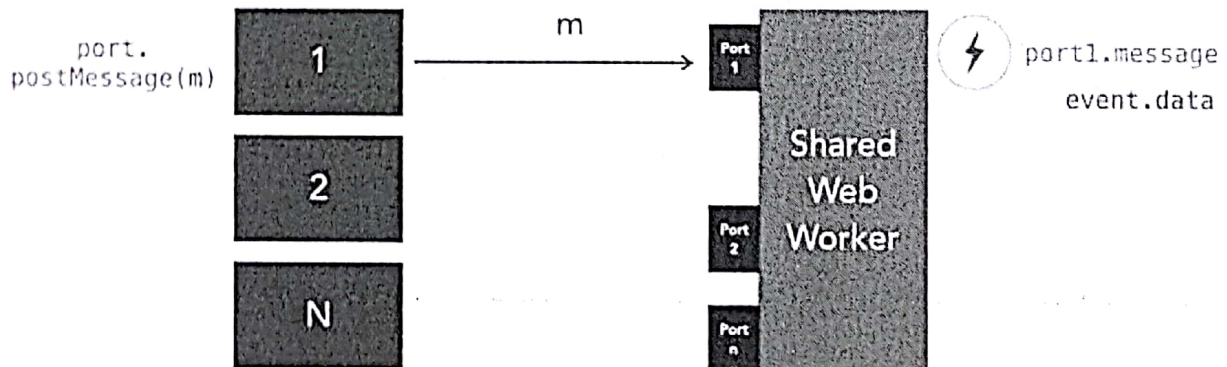
```

```

//index.html
<script>
  const worker = new Worker("worker.js", {
    name: "chandra-worker"
  });
  worker.addEventListener("message", event => {
    if (event.data.action == "output") {
      document.querySelector("#output").innerHTML = event.data.data;
    }
  })
  function start() {
    worker.postMessage({
      action: "start"
    })
  }
</script>
<button onclick="start()"> Click me</button>
<div id="output"></div>

```

Send Messages to shared workers:

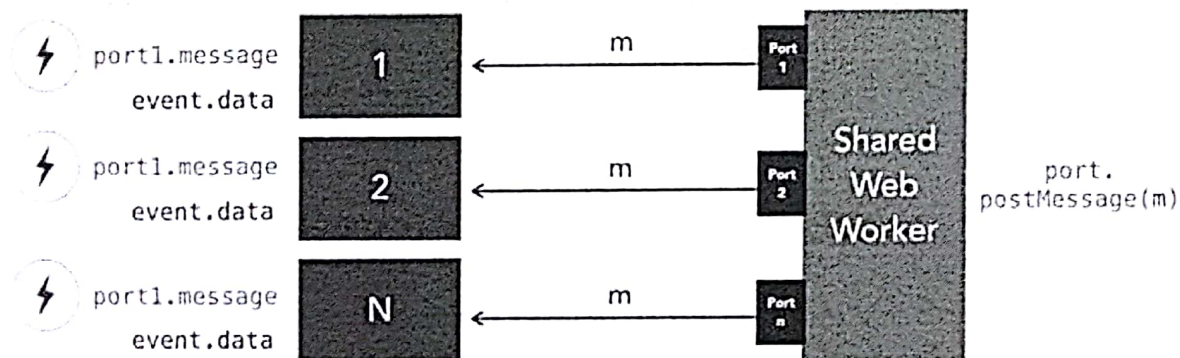


Messaging with a shared web worker

It's time to talk about the ownership of objects. So when we are using the `postMessage` API, we need to understand what's going on with the objects or the data that we are sending from one thread to the other thread. So for example, we are in the main thread. We have an object. In this case, we call that object `m`. It's the green object that we see on the screen. And you're posting that object to the web-worker. The web-worker will receive that object in the message, even handler.

But it's important to understand that the object that we receive on the other side has the same data, but it's a copy. So it's a clone of the original object. So now in memory, we have two objects with the same data. That means that we need to be very careful when we are sending a lot of data like megabytes of data or an array of images or something like that because we are duplicating that data and that has a performance impact. There is an additional API that is not available in every browser, but it's available in all the latest versions of all the browsers that will let us transfer objects instead of the default copy of the object.

To do that, when we are posting a message, we need to send a second argument with an array of objects that we want to transfer. So in this case, and typically, we are sending one object and then an array with only one element that is the same object. So we are sending an object and we are also asking for transferring the ownership of that object from our thread to the other thread. So in that case, if we do this, our `m` object will be transferred the same object to the web-worker, to the other thread which means there is no real transferring memory we are just pointing to the same space in memory.



Broadcasting from a shared worker



## Error Handling:

What happens if some kind of error happens within the scope of the thread? So in the web worker. There is an error event that will be fired if there are unhandled exceptions in the thread. When combined to that error event within the worker itself or in the main thread. So the main thread can know if there is any error happening in one of its dedicated workers. Also, sometimes we can send wrong data on `postMessage`.

In that case, another event will be fired. In this case the name means `messageerror`. So it's `messageerror` in that case on both sides. You can know if there is a problem with the `postMessage` argument that you are sending. Why is that a problem? Because we have already seen that both message will accept all primitive types for Copy. So that's the default `postMessage`. Also you can send `Date`, `RegularExpressions`, `Blob`, `File`, `Object`, `FileList`, `ArrayBuffers`, `ImageData`, normal Arrays, normal Objects, Maps, and Sets.

Anything else might not be possible to be sent in `postMessage`. For example, the window object, thumb element, and many other object within JavaScript. Also for remember that if you want to Transfer, you can transfer `ArrayBuffers`, `Blobs`, or `ImageData`. And trying to transfer other things might also result in a `messageerror`. What you can do is sometimes for example if you want to send other Object, you can try to move that Object into an `ArrayBuffer`. Such as serialize you need in JSON or something else and then get the bytes of that as an `ArrayBuffer` and then Transfer the `ArrayBuffer`.

Libraries for web workers:

1. Catiline.js:

### Catiline.js

```
const worker = cw( (data, callback) => {  
  // we receive data  
  // we process and execute  
  // callback with the answer  
});  
worker.data(message)  
  .then(reply => {  
  
  })
```

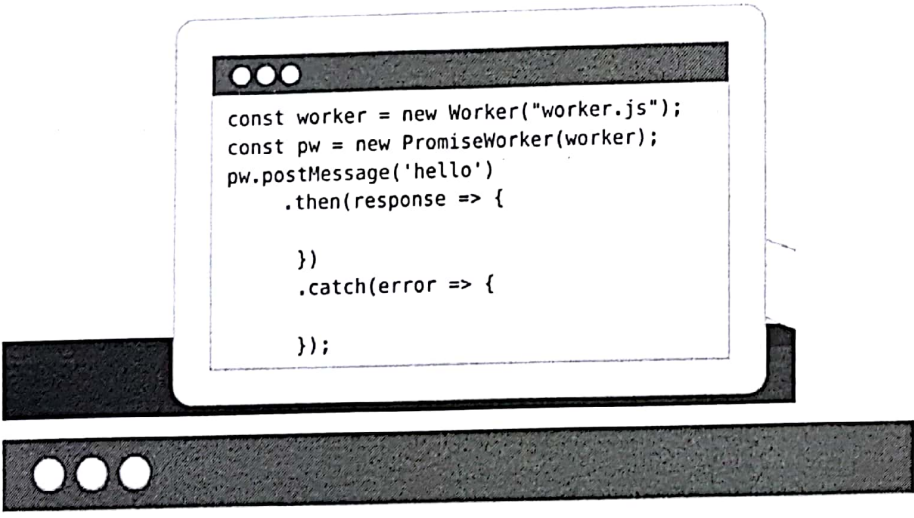


Catiline.js. In this case, we don't need to create a worker as an external file. It creates embedded web workers. And for that we need to create a worker with level cw function. That means Catiline worker. And we need to send a function as an argument. That function will receive data as a first argument and a callback function as a second argument. And then we need to send messages to that worker with the data function.

So worker.data. We send the message. That message will be received as data as the first argument in the function. And there is a then there similar to a promise. This is not really a promise. And where we're going to receive the reply. That's basically the framework. Of course we need to download the framework and import the framework in the scope of our HTML.

## 2. Promise worker:

### PromiseWorker



```
const worker = new Worker("worker.js");
const pw = new PromiseWorker(worker);
pw.postMessage('hello')
  .then(response => {

  })
  .catch(error => {

  });
```

```
const worker = new Worker("worker.js");
const pw = new PromiseWorker(worker);
const response =
    await
pw.postMessage('hello');
```

In this case it's based on the normal Web Worker API.

So we still need to create another external file for the worker. And we need to import one child script in the main thread and one child script in the worker. So this library is split in two pieces and it's basically promisifying the Web Worker API. So if you like to work with promises, and for example async/await, in the case that you're using ES8 or if you're transpiling or using TypeScript and you want to use that thing well, you can use PromiseWorker.

In this case you need to create the normal Worker and then you create the PromiseWorker. So it's going to promisify your worker. So then when you post a message you will receive a promise. So you can use then or catch or you can use await and try-catch.

### 3. Thread.js

## Thread.js

```
const thread = spawn( (data, done) => {  
  // receive data, process and call done  
  done("bye");  
});  
  
thread.send("hello")  
  .on("message", response => { })  
  .on("error", error => { })  
  .on("exit", () => { });
```

This library is getting rid out of the name of worker. It's talking about threads.

And a thread it's also a function that we send to a global variable and in that case that function will receive data and done. Of course you can change those names. Data will be incoming data. And done is a callback function that you need to call when you want to send an answer. And then you use that thread sending messages and then listening for events. There are three events that we listen with the on chain the function.

So this is like jQuery. So we are receiving the message. That's the response that we get from that thread. We can listen for errors or exit which means that the worker has been terminated. So this are some libraries that you can use to create web workers in a different way