WEB TECHNOLOGIES 1

Web Workers

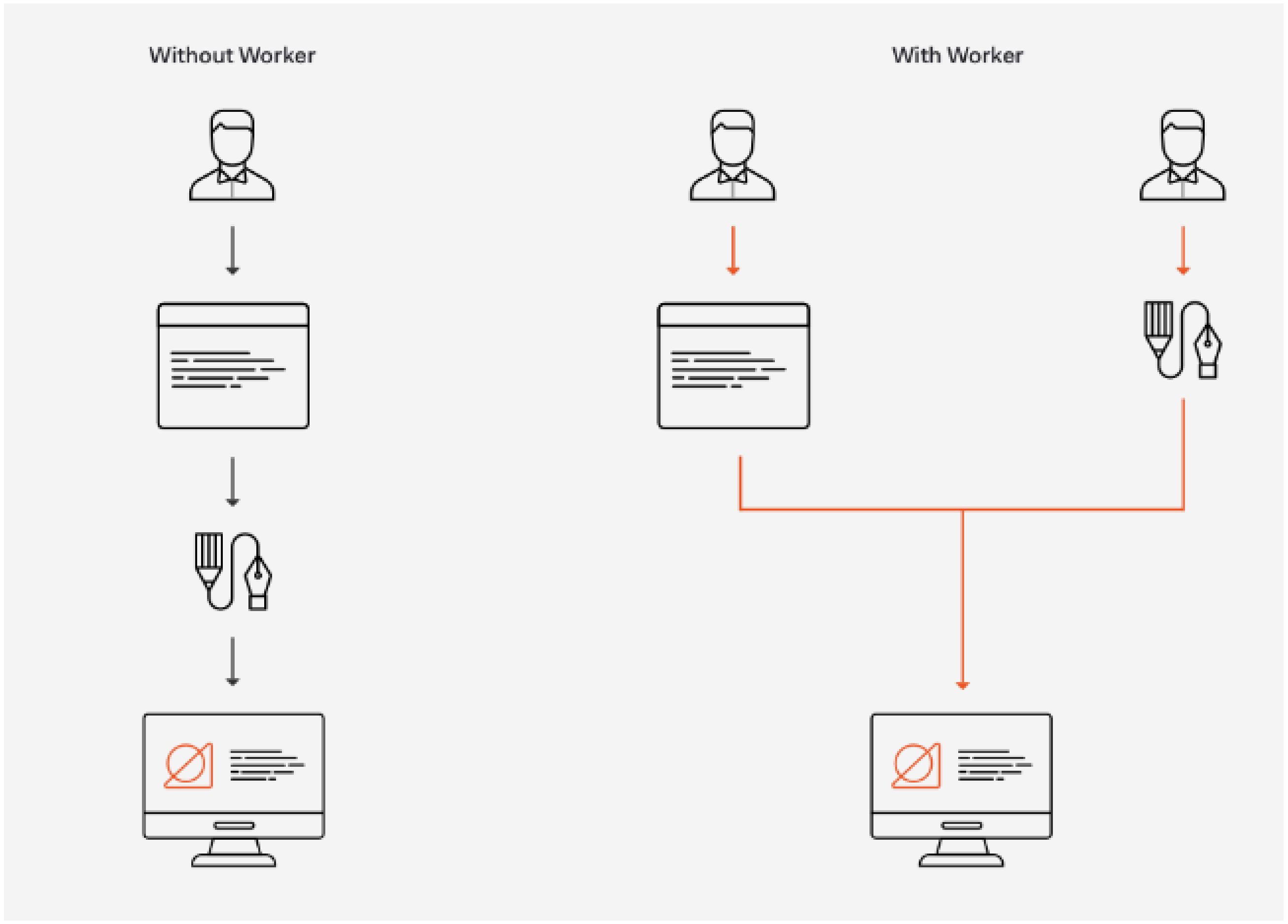
Web Worker

- Web Workers are a simple means for web content to run scripts in background threads.
- The worker thread can perform tasks without interfering with the user interface.
- In addition, they can perform I/O using XMLHttpRequest(although the responseXML and channel attributes are always null).
- Once created, a worker can send messages to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa).

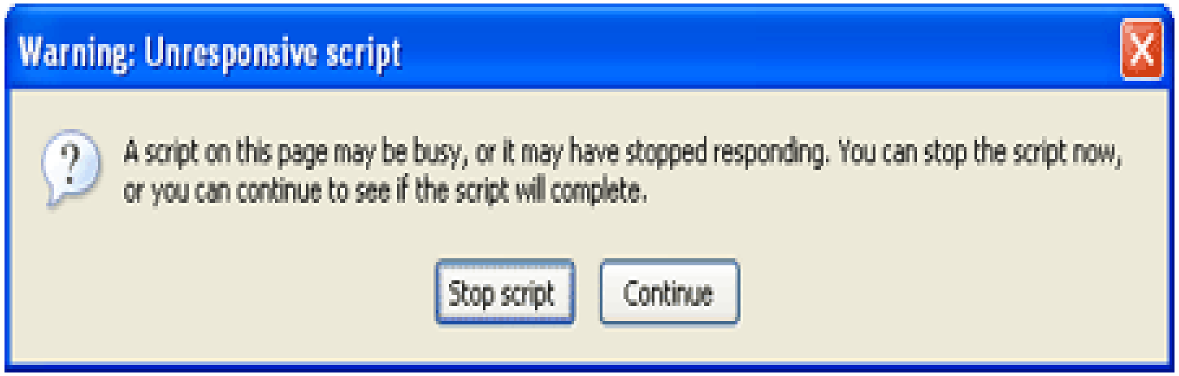
Web Workers

- Using web workers in HTML5 allows you to prevent the execution of bigger tasks from freezing up your web page.
- A web worker performs the job in the **background**, independent of other scripts and thus not affecting their performance.
- The process is also called **threading**, i.e., separating the tasks into multiple parallel threads.
- During the time, the user can browse normally, as the page stays **fully responsive**.

Web Worker



Web Workers



Warning: Unresponsive script

A script on this page may be busy, or it may have stopped responding. You can stop the script now, or you can continue to see if the script will complete.

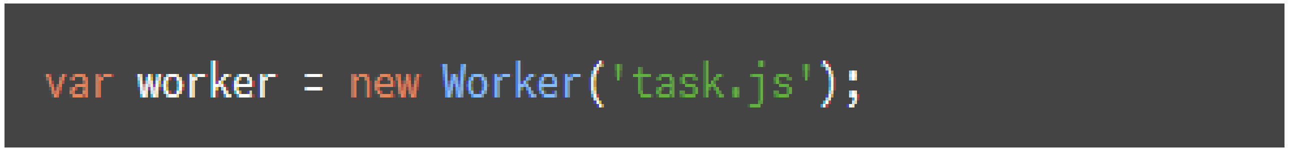
- Web Workers are not part of JavaScript, they’re a browser feature which can be accessed through JavaScript.
- Most browsers have historically been single-threaded , and most JavaScript implementations happen in the browser.
- Web Workers are not implemented in Node.JS

Types of Web Workers

- Dedicated Workers
  - Dedicated Web Workers are instantiated by the main process and can only communicate with it.
  - A dedicated worker is only accessible by the script that called it.
- Shared Workers
  - Shared workers can be reached by all processes running on the same origin (different browser tabs, iframes or other shared workers).
  - A shared worker is accessible by multiple scripts, similar to the basic dedicated worker, except that it has two functions available handled by different script files

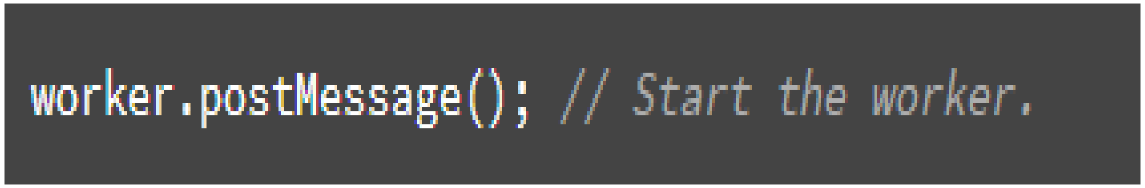
Getting Started

- Web Workers run in an isolated thread.
- The code that execute needs to be contained in a separate file.
- But before that, the first thing to do is create a new Worker object in main page. The constructor takes the name of the worker script:



```
var worker = new Worker('task.js');
```

- If the specified file exists, the browser will spawn a new worker thread, which is downloaded asynchronously.
- The worker will not begin until the file has completely downloaded and executed. If the path to your worker returns an 404, the worker will fail silently.
- After creating the worker, start it by calling the postMessage() method:

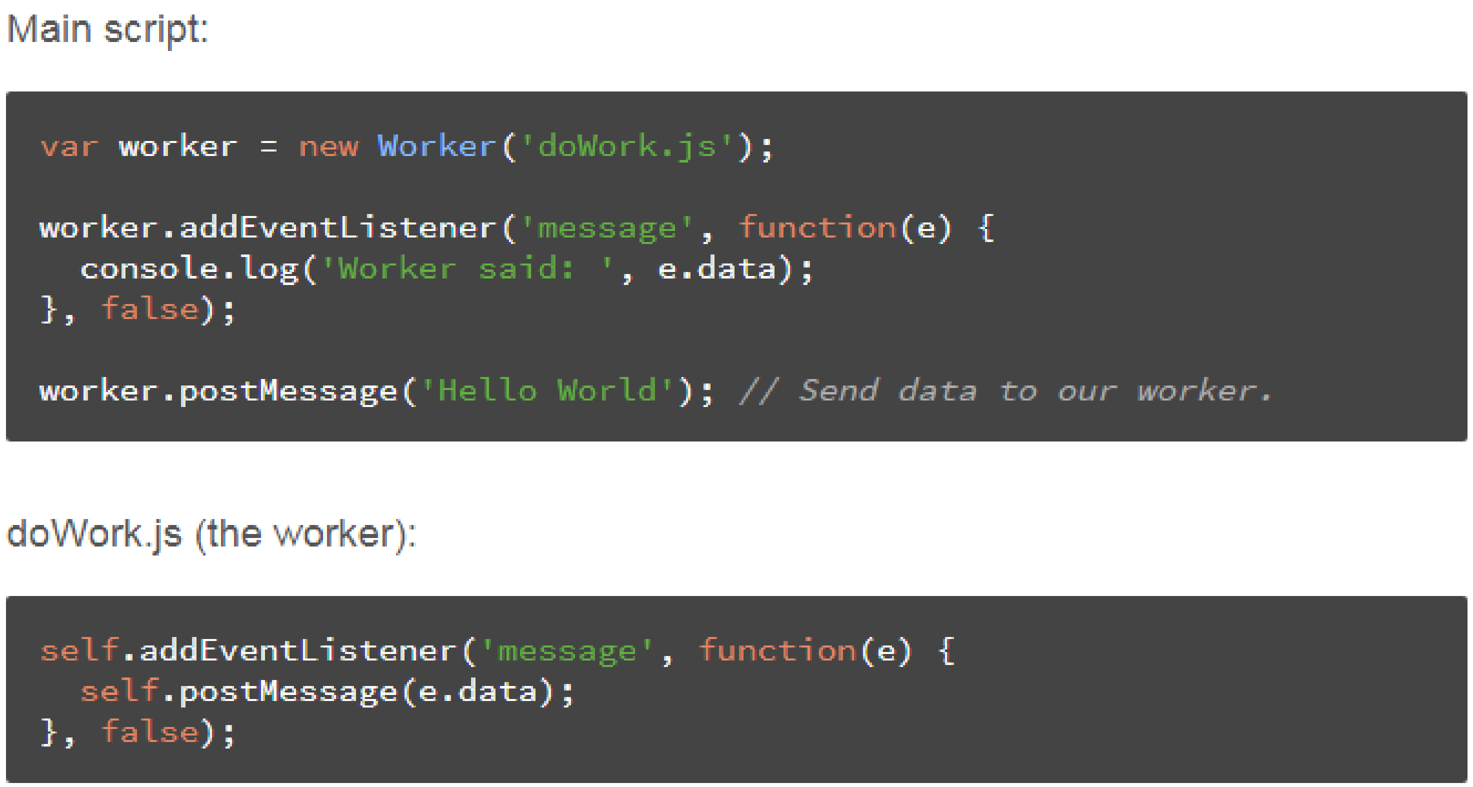


```
worker.postMessage(); // Start the worker.
```

Communicating with a Worker via Message Passing

- Communication between a work and its parent page is done using an event model and the postMessage() method.
- Depending on your browser/version, postMessage() can accept either a string or JSON object as its single argument.
- The latest versions of the modern browsers support passing a JSON object.
- In the example of using a string to pass 'Hello World' to a worker in doWork.js.
- The worker simply returns the message that is passed to it

Communicating with a Worker via Message Passing



Main script:

```
var worker = new Worker('doWork.js');

worker.addEventListener('message', function(e) {
  console.log('Worker said: ', e.data);
}, false);

worker.postMessage('Hello World'); // Send data to our worker.
```

doWork.js (the worker):

```
self.addEventListener('message', function(e) {
  self.postMessage(e.data);
}, false);
```

- When postMessage() is called from the main page, our worker handles that message by defining an onmessage handler for the message event.
- The message is accessible in Event.data

WEB WORKER COMMUNICATION

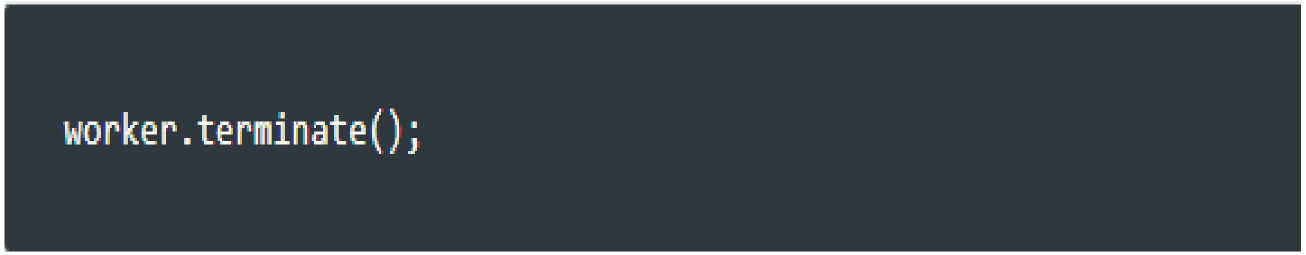
- The postMessage() takes a single parameter representing the data that you want to send. This data may be any value or JavaScript object handled by the structured clone algorithm.
- Messages passed between the main page and workers are copied, not shared nor as a reference.
- There are two ways to stop a worker: by calling worker.terminate() from the main page or by calling self.close() inside of the worker itself.
- In the context of a worker, both self and this reference the global scope for the worker.

TERMINATING A WORKER

- If you need to immediately terminate a running worker from the main thread, you can do so by calling the worker's terminate method:

  // Worker.terminate();

- Web worker is destroyed immediately without any chance of completing any ongoing or pending operations.
- The web worker is also given no time to clean up. Thus, terminating a web worker abruptly may lead to memory leaks.



```
worker.terminate();
```

HANDLING ERRORS

- As with any JavaScript code, you’ll want to handle any errors that are thrown in your Web Workers.
- If an error occurs while a worker is executing, the ErrorEvent is fired. It receives an event named *error* which implements the *ErrorEvent* interface.
- The event doesn't bubble and is cancelable; to prevent the default action from taking place, the worker can call the error event's preventDefault().
- The interface contains three useful properties for figuring out what went wrong:
  - **filename** - the name of the worker script that caused the error
  - **lineno** - the line number where the error occurred
  - **message** - a description of the error

Features available to Web Workers

- Web Workers have access **only to a subset** of JavaScript features due to their multi-threaded nature. Here’s the list of features:
  - The navigator object
  - The location object (read-only)
  - XMLHttpRequest
  - setTimeout()/clearTimeout()
  - setInterval()/clearInterval()
  - The Application Cache
  - Importing external scripts using importScripts()
  - Creating other web workers

Web Worker Limitations

- All web worker scripts must be served from the same domain.
- You cannot have direct access to the DOM and the global document, window object, parent object.
- The window object exposes limited API. For instance, location and navigator and XMLHttpRequest objects.
- Restricted local access. Web workers do not work on static files. For instance file://my/file/on/my/computer.

USE CASES FOR WEB WORKERS

- **Encryption:**
- End-to-End encryption is getting more and more popular due to the increasing rigorousness of regulations on personal and sensitive data.
- Encryption can be a something quite time-consuming, especially if there’s a lot of data that has to be frequently encrypted (before sending it to the server, for example).
- This is a very good scenario in which a Web Worker can be used since it doesn’t require any access to the DOM or anything fancy — it’s pure algorithms doing their job.
- Once in the worker, it is seamless to the end user and doesn’t impact their experience.

USE CASES FOR WEB WORKERS

- **Prefetching data:**
- In order to optimize your website or web application and improve data loading time.
- We can leverage Web Workers to load and store some data in advance so that you can use it later when needed.
- Web Workers won’t impact your app’s UI, unlike when this is done without workers.

THE SIZE OF MESSAGES

- There are 2 ways to send messages to Web Workers:
- **Copying the message:**
  - The message is serialized, copied, sent over, and then de-serialized at the other end.
  - The page and worker do not share the same instance, so the end result is that a duplicate is created on each pass.
  - Most browsers implement this feature by automatically JSON encoding/decoding the value at either end.
  - As expected, these data operations add significant overhead to the message transmission. The bigger the message, the longer it takes to be sent.
- **Transferring the message:**
  - This means that the original sender can no longer use it once sent.
  - Transferring data is almost instantaneous.
  - The limitation is that only ArrayBuffer is transferable.

POSTMESSAGE

- When the button is clicked, *postMessage* will be called from the main page. The *worker.postMessage* line passes the JSON object to the worker, adding *cmd* and *data* keys with their respective values. The worker will handle that message through the defined message handler.
- When the message arrives, the actual computing is being performed in the worker, without blocking the event loop.
- The worker is checking the passed event *e* and executes just like a standard JavaScript function. When it's done, the result is passed back to the main page.
- In the context of a worker, both the *self* and *this* reference the global scope for the worker.
- There are two ways to stop a worker:
  - by calling worker.terminate() from the main page
  - by calling self.close() inside of the worker itself.