HTML

Living Standard — Last Updated 18 July 2019

10 Web workers §

10.1 Introduction §

10.1.1 Scope §

This section is non-normative.

This specification defines an API for running scripts in the background independently of any user interface scripts.

This allows for long-running scripts that are not interrupted by scripts that respond to clicks or other user interactions, and allows long tasks to be executed without yielding to keep the page responsive.

Workers (as these background scripts are called herein) are relatively heavy-weight, and are not intended to be used in large numbers. For example, it would be inappropriate to launch one worker for each pixel of a four megapixel image. The examples below show some appropriate uses of workers.

Generally, workers are expected to be long-lived, have a high start-up performance cost, and a high per-instance memory cost.

10.1.2 Examples §

This section is non-normative.

There are a variety of uses that workers can be put to. The following subsections show various examples of this use.

10.1.2.1 A background number-crunching worker §

This section is non-normative.

The simplest use of workers is for performing a computationally expensive task without interrupting the user interface.

In this example, the main document spawns a worker to (naïvely) compute prime numbers, and progressively displays the most recently found prime number.

The main page is as follows:

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Worker example: One-core computation</title>
 </head>
 <body>
  <p>The highest prime number discovered so far is: <output id="result"></output></p>
  <script>
   var worker = new Worker('worker.js');
   worker.onmessage = function (event) {
     document.getElementById('result').textContent = event.data;
   };
  </script>
 </body>
</html>
```

The `Worker()` constructor call creates a worker and returns a `Worker` object representing that worker, which is used to communicate with the worker. That object's `onmessage` event handler allows the code to receive messages from the worker.

The worker itself is as follows:

```
var n = 1;
search: while (true) {
  n += 1;
  for (var i = 2; i <= Math.sqrt(n); i += 1)
    if (n % i == 0)
      continue search;
  // found a prime!
  postMessage(n);
}
```

The bulk of this code is simply an unoptimized search for a prime number. The `postMessage()` method is used to send a message back to the page when a prime is found.

View this example online.

10.1.2.2 Using a JavaScript module as a worker §

This section is non-normative.

All of our examples so far show workers that run classic scripts. Workers can instead be instantiated using module scripts, which have the usual benefits: the ability to use the JavaScript `import` statement to import other modules; strict mode by default; and top-level declarations not polluting the worker's global scope.

Note that such module-based workers follow different restrictions regarding cross-origin content, compared to classic workers. Unlike classic workers, module workers can be instantiated using a cross-origin script, as long as that script is exposed using the CORS protocol. Additionally, the `importScripts()` method will automatically fail inside module workers; the JavaScript `import` statement is generally a better choice.

In this example, the main document uses a worker to do off-main-thread image manipulation. It imports the filters used from another module.

The main page is as follows:

```
<!DOCTYPE html>
<meta charset="utf-8">
<title>Worker example: image decoding</title>

<p>
  <label>
    Type an image URL to decode
    <input type="url" id="image-url" list="image-list">
    <datalist id="image-list">
      <option value="https://html.spec.whatwg.org/images/drawImage.png">
      <option value="https://html.spec.whatwg.org/images/robots.jpeg">
      <option value="https://html.spec.whatwg.org/images/arcTo2.png">
    </datalist>
  </label>
</p>

<p>
  <label>
    Choose a filter to apply
    <select id="filter">
      <option value="none">none</option>
      <option value="grayscale">grayscale</option>
      <option value="brighten">brighten by 20%</option>
    </select>
  </label>
</p>

<canvas id="output"></canvas>

<script type="module">
  const worker = new Worker("worker.js", { type: "module" });
  worker.onmessage = receiveFromWorker;

  const url = document.querySelector("#image-url");
  const filter = document.querySelector("#filter");
  const output = document.querySelector("#output");

  url.oninput = updateImage;
  filter.oninput = sendToWorker;

  let imageData, context;

  function updateImage() {
    const img = new Image();
    img.src = url.value;

    img.onload = () => {
      output.innerHTML = "";

      const canvas = document.createElement("canvas");
      canvas.width = img.width;
      canvas.height = img.height;

      context = canvas.getContext("2d");
      context.drawImage(img, 0, 0);
      imageData = context.getImageData(0, 0, canvas.width, canvas.height);

      sendToWorker();
      output.appendChild(canvas);
```

```
  }

  function sendToWorker() {
    worker.postMessage({ imageData, filter: filter.value });
  }

  function receiveFromWorker(e) {
    context.putImageData(e.data, 0, 0);
  }
</script>
```

The worker file is then:

```
import * as filters from "./filters.js";

self.onmessage = e => {
  const { imageData, filter } = e.data;
  filters[filter](imageData);
  self.postMessage(imageData, [imageData.data.buffer]);
};
```

Which imports the file `filters.js`:

```
export function none() {}

export function grayscale({ data: d }) {
  for (let i = 0; i < d.length; i += 4) {
    const [r, g, b] = [d[i], d[i + 1], d[i + 2]];

    // CIE luminance for the RGB
    // The human eye is bad at seeing red and blue, so we de-emphasize them.
    d[i] = d[i + 1] = d[i + 2] = 0.2126 * r + 0.7152 * g + 0.0722 * b;
  }
};

export function brighten({ data: d }) {
  for (let i = 0; i < d.length; ++i) {
    d[i] *= 1.2;
  }
};
```

View this example online.

10.1.2.3 Shared workers introduction §

This section is non-normative.

This section introduces shared workers using a Hello World example. Shared workers use slightly different APIs, since each worker can have multiple connections.

This first example shows how you connect to a worker and how a worker can send a message back to the page when it connects to it. Received messages are displayed in a log.

Here is the HTML page:

```
<!DOCTYPE HTML>
<meta charset="utf-8">
<title>Shared workers: demo 1</title>
<pre id="log">Log:</pre>
<script>
  var worker = new SharedWorker('test.js');
  var log = document.getElementById('log');
  worker.port.onmessage = function(e) { // note: not worker.onmessage!
    log.textContent += '\n' + e.data;
  }
</script>
```

Here is the JavaScript worker:

```
onconnect = function(e) {
  var port = e.ports[0];
  port.postMessage('Hello World!');
}
```

View this example online.

This second example extends the first one by changing two things: first, messages are received using `addEventListener()` instead of an event handler IDL attribute, and second, a message is sent *to* the worker, causing the worker to send another message in return. Received messages are again displayed in a log.

Here is the HTML page:

```
<!DOCTYPE HTML>
<meta charset="utf-8">
<title>Shared workers: demo 2</title>
<pre id="log">Log:</pre>
<script>
  var worker = new SharedWorker('test.js');
  var log = document.getElementById('log');
  worker.port.addEventListener('message', function(e) {
    log.textContent += '\n' + e.data;
  }, false);
  worker.port.start(); // note: need this when using addEventListener
  worker.port.postMessage('ping');
</script>
```

Here is the JavaScript worker:

```
onconnect = function(e) {
  var port = e.ports[0];
  port.postMessage('Hello World!');
  port.onmessage = function(e) {
    port.postMessage('pong'); // not e.ports[0].postMessage!
    // e.target.postMessage('pong'); would work also
  }
}
```

View this example online.

Finally, the example is extended to show how two pages can connect to the same worker; in this case, the second page is merely in an `iframe` on the first page, but the same principle would apply to an entirely separate page in a separate top-level browsing context.

Here is the outer HTML page:

```
<!DOCTYPE HTML>
<meta charset="utf-8">
<title>Shared workers: demo 3</title>
<pre id="log">Log:</pre>
<script>
  var worker = new SharedWorker('test.js');
  var log = document.getElementById('log');
  worker.port.addEventListener('message', function(e) {
    log.textContent += '\n' + e.data;
  }, false);
  worker.port.start();
  worker.port.postMessage('ping');
</script>
<iframe src="inner.html"></iframe>
```

Here is the inner HTML page:

```
<!DOCTYPE HTML>
<meta charset="utf-8">
<title>Shared workers: demo 3 inner frame</title>
<pre id=log>Inner log:</pre>
<script>
  var worker = new SharedWorker('test.js');
  var log = document.getElementById('log');
  worker.port.onmessage = function(e) {
   log.textContent += '\n' + e.data;
  }
</script>
```

Here is the JavaScript worker:

```
var count = 0;
onconnect = function(e) {
  count += 1;
  var port = e.ports[0];
  port.postMessage('Hello World! You are connection #' + count);
```

```
  port.onmessage = function(e) {
    port.postMessage('pong');
  }
}
```

View this example online.

10.1.2.4 Shared state using a shared worker §

This section is non-normative.

In this example, multiple windows (viewers) can be opened that are all viewing the same map. All the windows share the same map information, with a single worker coordinating all the viewers. Each viewer can move around independently, but if they set any data on the map, all the viewers are updated.

The main page isn't interesting, it merely provides a way to open the viewers:

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Workers example: Multiviewer</title>
  <script>
   function openViewer() {
     window.open('viewer.html');
   }
  </script>
 </head>
 <body>
  <p><button type=button onclick="openViewer()">Open a new
  viewer</button></p>
  <p>Each viewer opens in a new window. You can have as many viewers
  as you like, they all view the same data.</p>
 </body>
</html>
```

The viewer is more involved:

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Workers example: Multiviewer viewer</title>
  <script>
   var worker = new SharedWorker('worker.js', 'core');

   // CONFIGURATION
   function configure(event) {
     if (event.data.substr(0, 4) != 'cfg ') return;
     var name = event.data.substr(4).split(' ', 1)[0];
     // update display to mention our name is name
     document.getElementsByTagName('h1')[0].textContent += ' ' + name;
     // no longer need this listener
     worker.port.removeEventListener('message', configure, false);
   }
   worker.port.addEventListener('message', configure, false);

   // MAP
   function paintMap(event) {
     if (event.data.substr(0, 4) != 'map ') return;
     var data = event.data.substr(4).split(',');
     // display tiles data[0] .. data[8]
     var canvas = document.getElementById('map');
     var context = canvas.getContext('2d');
     for (var y = 0; y < 3; y += 1) {
       for (var x = 0; x < 3; x += 1) {
         var tile = data[y * 3 + x];
         if (tile == '0')
           context.fillStyle = 'green';
         else
           context.fillStyle = 'maroon';
         context.fillRect(x * 50, y * 50, 50, 50);
       }
     }
   }
   worker.port.addEventListener('message', paintMap, false);
```

```
  // PUBLIC CHAT
  function updatePublicChat(event) {
    if (event.data.substr(0, 4) != 'txt ') return;
    var name = event.data.substr(4).split(' ', 1)[0];
    var message = event.data.substr(4 + name.length + 1);
    // display "<name> message" in public chat
    var public = document.getElementById('public');
    var p = document.createElement('p');
    var n = document.createElement('button');
    n.textContent = '<' + name + '> ';
    n.onclick = function () { worker.port.postMessage('msg ' + name); };
    p.appendChild(n);
    var m = document.createElement('span');
    m.textContent = message;
    p.appendChild(m);
    public.appendChild(p);
  }
  worker.port.addEventListener('message', updatePublicChat, false);

  // PRIVATE CHAT
  function startPrivateChat(event) {
    if (event.data.substr(0, 4) != 'msg ') return;
    var name = event.data.substr(4).split(' ', 1)[0];
    var port = event.ports[0];
    // display a private chat UI
    var ul = document.getElementById('private');
    var li = document.createElement('li');
    var h3 = document.createElement('h3');
    h3.textContent = 'Private chat with ' + name;
    li.appendChild(h3);
    var div = document.createElement('div');
    var addMessage = function(name, message) {
      var p = document.createElement('p');
      var n = document.createElement('strong');
      n.textContent = '<' + name + '> ';
      p.appendChild(n);
      var t = document.createElement('span');
      t.textContent = message;
      p.appendChild(t);
      div.appendChild(p);
    };
    port.onmessage = function (event) {
      addMessage(name, event.data);
    };
    li.appendChild(div);
    var form = document.createElement('form');
    var p = document.createElement('p');
    var input = document.createElement('input');
    input.size = 50;
    p.appendChild(input);
    p.appendChild(document.createTextNode(' '));
    var button = document.createElement('button');
    button.textContent = 'Post';
    p.appendChild(button);
    form.onsubmit = function () {
      port.postMessage(input.value);
      addMessage('me', input.value);
      input.value = '';
      return false;
    };
    form.appendChild(p);
    li.appendChild(form);
    ul.appendChild(li);
  }
  worker.port.addEventListener('message', startPrivateChat, false);

  worker.port.start();
 </script>
</head>
<body>
 <h1>Viewer</h1>
 <h2>Map</h2>
 <p><canvas id="map" height=150 width=150></canvas></p>
 <p>
  <button type=button onclick="worker.port.postMessage('mov left')">Left</button>
  <button type=button onclick="worker.port.postMessage('mov up')">Up</button>
  <button type=button onclick="worker.port.postMessage('mov down')">Down</button>
  <button type=button onclick="worker.port.postMessage('mov right')">Right</button>
  <button type=button onclick="worker.port.postMessage('set 0')">Set 0</button>
```

```
   <button type=button onclick="worker.port.postMessage('set 1')">Set 1</button>
  </p>
  <h2>Public Chat</h2>
  <div id="public"></div>
  <form onsubmit="worker.port.postMessage('txt ' + message.value); message.value = ''; return false;">
   <p>
    <input type="text" name="message" size="50">
    <button>Post</button>
   </p>
  </form>
  <h2>Private Chat</h2>
  <ul id="private"></ul>
 </body>
</html>
```

There are several key things worth noting about the way the viewer is written.

Multiple listeners. Instead of a single message processing function, the code here attaches multiple event listeners, each one performing a quick check to see if it is relevant for the message. In this example it doesn't make much difference, but if multiple authors wanted to collaborate using a single port to communicate with a worker, it would allow for independent code instead of changes having to all be made to a single event handling function.

Registering event listeners in this way also allows you to unregister specific listeners when you are done with them, as is done with the `configure()` method in this example.

Finally, the worker:

```
var nextName = 0;
function getNextName() {
  // this could use more friendly names
  // but for now just return a number
  return nextName++;
}

var map = [
 [0, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 1, 0, 1, 1],
 [0, 1, 0, 1, 0, 0, 0],
 [0, 1, 0, 1, 0, 1, 1],
 [0, 0, 0, 1, 0, 0, 0],
 [1, 0, 0, 1, 1, 1, 1],
 [1, 1, 0, 1, 1, 0, 1],
];

function wrapX(x) {
  if (x < 0) return wrapX(x + map[0].length);
  if (x >= map[0].length) return wrapX(x - map[0].length);
  return x;
}

function wrapY(y) {
  if (y < 0) return wrapY(y + map.length);
  if (y >= map[0].length) return wrapY(y - map.length);
  return y;
}

function wrap(val, min, max) {
  if (val < min)
    return val + (max-min)+1;
  if (val > max)
    return val - (max-min)-1;
  return val;
}

function sendMapData(viewer) {
  var data = '';
  for (var y = viewer.y-1; y <= viewer.y+1; y += 1) {
    for (var x = viewer.x-1; x <= viewer.x+1; x += 1) {
      if (data != '')
        data += ',';
      data += map[wrap(y, 0, map[0].length-1)][wrap(x, 0, map.length-1)];
    }
  }
  viewer.port.postMessage('map ' + data);
}

var viewers = {};
onconnect = function (event) {
  var name = getNextName();
```

```
  event.ports[0]._data = { port: event.ports[0], name: name, x: 0, y: 0, };
  viewers[name] = event.ports[0]._data;
  event.ports[0].postMessage('cfg ' + name);
  event.ports[0].onmessage = getMessage;
  sendMapData(event.ports[0]._data);
};

function getMessage(event) {
  switch (event.data.substr(0, 4)) {
    case 'mov ':
      var direction = event.data.substr(4);
      var dx = 0;
      var dy = 0;
      switch (direction) {
        case 'up': dy = -1; break;
        case 'down': dy = 1; break;
        case 'left': dx = -1; break;
        case 'right': dx = 1; break;
      }
      event.target._data.x = wrapX(event.target._data.x + dx);
      event.target._data.y = wrapY(event.target._data.y + dy);
      sendMapData(event.target._data);
      break;
    case 'set ':
      var value = event.data.substr(4);
      map[event.target._data.y][event.target._data.x] = value;
      for (var viewer in viewers)
        sendMapData(viewers[viewer]);
      break;
    case 'txt ':
      var name = event.target._data.name;
      var message = event.data.substr(4);
      for (var viewer in viewers)
        viewers[viewer].port.postMessage('txt ' + name + ' ' + message);
      break;
    case 'msg ':
      var party1 = event.target._data;
      var party2 = viewers[event.data.substr(4).split(' ', 1)[0]];
      if (party2) {
        var channel = new MessageChannel();
        party1.port.postMessage('msg ' + party2.name, [channel.port1]);
        party2.port.postMessage('msg ' + party1.name, [channel.port2]);
      }
      break;
  }
}
```

Connecting to multiple pages. The script uses the `onconnect` event listener to listen for multiple connections.

Direct channels. When the worker receives a "msg" message from one viewer naming another viewer, it sets up a direct connection between the two, so that the two viewers can communicate directly without the worker having to proxy all the messages.

View this example online.

10.1.2.5 Delegation §

This section is non-normative.

With multicore CPUs becoming prevalent, one way to obtain better performance is to split computationally expensive tasks amongst multiple workers. In this example, a computationally expensive task that is to be performed for every number from 1 to 10,000,000 is farmed out to ten subworkers.

The main page is as follows, it just reports the result:

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Worker example: Multicore computation</title>
 </head>
 <body>
  <p>Result: <output id="result"></output></p>
  <script>
   var worker = new Worker('worker.js');
   worker.onmessage = function (event) {
     document.getElementById('result').textContent = event.data;
   };
```

```
 </body>
</html>
```

The worker itself is as follows:

```
// settings
var num_workers = 10;
var items_per_worker = 1000000;

// start the workers
var result = 0;
var pending_workers = num_workers;
for (var i = 0; i < num_workers; i += 1) {
  var worker = new Worker('core.js');
  worker.postMessage(i * items_per_worker);
  worker.postMessage((i+1) * items_per_worker);
  worker.onmessage = storeResult;
}

// handle the results
function storeResult(event) {
  result += 1*event.data;
  pending_workers -= 1;
  if (pending_workers <= 0)
    postMessage(result); // finished!
}
```

It consists of a loop to start the subworkers, and then a handler that waits for all the subworkers to respond.

The subworkers are implemented as follows:

```
var start;
onmessage = getStart;
function getStart(event) {
  start = 1*event.data;
  onmessage = getEnd;
}

var end;
function getEnd(event) {
  end = 1*event.data;
  onmessage = null;
  work();
}

function work() {
  var result = 0;
  for (var i = start; i < end; i += 1) {
    // perform some complex calculation here
    result += 1;
  }
  postMessage(result);
  close();
}
```

They receive two numbers in two events, perform the computation for the range of numbers thus specified, and then report the result back to the parent.

View this example online.

10.1.2.6 Providing libraries §

This section is non-normative.

Suppose that a cryptography library is made available that provides three tasks:

Generate a public/private key pair

Takes a port, on which it will send two messages, first the public key and then the private key.

Given a plaintext and a public key, return the corresponding ciphertext

Takes a port, to which any number of messages can be sent, the first giving the public key, and the remainder giving the plaintext, each of which is encrypted and then sent on that same channel as the ciphertext. The user can close the port when it is done encrypting content.

Given a ciphertext and a private key, return the corresponding plaintext

Takes a port, to which any number of messages can be sent, the first giving the private key, and the remainder giving the ciphertext, each of which is decrypted and then sent on that same channel as the plaintext. The user can close the port when it is done decrypting content.

The library itself is as follows:

```
function handleMessage(e) {
  if (e.data == "genkeys")
    genkeys(e.ports[0]);
  else if (e.data == "encrypt")
    encrypt(e.ports[0]);
  else if (e.data == "decrypt")
    decrypt(e.ports[0]);
}

function genkeys(p) {
  var keys = _generateKeyPair();
  p.postMessage(keys[0]);
  p.postMessage(keys[1]);
}

function encrypt(p) {
  var key, state = 0;
  p.onmessage = function (e) {
    if (state == 0) {
      key = e.data;
      state = 1;
    } else {
      p.postMessage(_encrypt(key, e.data));
    }
  };
}

function decrypt(p) {
  var key, state = 0;
  p.onmessage = function (e) {
    if (state == 0) {
      key = e.data;
      state = 1;
    } else {
      p.postMessage(_decrypt(key, e.data));
    }
  };
}

// support being used as a shared worker as well as a dedicated worker
if ('onmessage' in this) // dedicated worker
  onmessage = handleMessage;
else // shared worker
  onconnect = function (e) { e.port.onmessage = handleMessage; }


// the "crypto" functions:

function _generateKeyPair() {
  return [Math.random(), Math.random()];
}

function _encrypt(k, s) {
  return 'encrypted-' + k + ' ' + s;
}

function _decrypt(k, s) {
  return s.substr(s.indexOf(' ')+1);
}
```

Note that the crypto functions here are just stubs and don't do real cryptography.

This library could be used as follows:

```
<!DOCTYPE HTML>
<html>
 <head>
  <meta charset="utf-8">
  <title>Worker example: Crypto library</title>
  <script>
   const cryptoLib = new Worker('libcrypto-v1.js'); // or could use 'libcrypto-v2.js'
   function startConversation(source, message) {
     const messageChannel = new MessageChannel();
     source.postMessage(message, [messageChannel.port2]);
     return messageChannel.port1;
```

```
    function getKeys() {
      let state = 0;
      startConversation(cryptoLib, "genkeys").onmessage = function (e) {
        if (state === 0)
          document.getElementById('public').value = e.data;
        else if (state === 1)
          document.getElementById('private').value = e.data;
        state += 1;
      };
    }
    function enc() {
      const port = startConversation(cryptoLib, "encrypt");
      port.postMessage(document.getElementById('public').value);
      port.postMessage(document.getElementById('input').value);
      port.onmessage = function (e) {
        document.getElementById('input').value = e.data;
        port.close();
      };
    }
    function dec() {
      const port = startConversation(cryptoLib, "decrypt");
      port.postMessage(document.getElementById('private').value);
      port.postMessage(document.getElementById('input').value);
      port.onmessage = function (e) {
        document.getElementById('input').value = e.data;
        port.close();
      };
    }
  </script>
  <style>
   textarea { display: block; }
  </style>
 </head>
 <body onload="getKeys()">
  <fieldset>
   <legend>Keys</legend>
   <p><label>Public Key: <textarea id="public"></textarea></label></p>
   <p><label>Private Key: <textarea id="private"></textarea></label></p>
  </fieldset>
  <p><label>Input: <textarea id="input"></textarea></label></p>
  <p><button onclick="enc()">Encrypt</button> <button onclick="dec()">Decrypt</button></p>
 </body>
</html>
```

A later version of the API, though, might want to offload all the crypto work onto subworkers. This could be done as follows:

```
function handleMessage(e) {
  if (e.data == "genkeys")
    genkeys(e.ports[0]);
  else if (e.data == "encrypt")
    encrypt(e.ports[0]);
  else if (e.data == "decrypt")
    decrypt(e.ports[0]);
}

function genkeys(p) {
  var generator = new Worker('libcrypto-v2-generator.js');
  generator.postMessage('', [p]);
}

function encrypt(p) {
  p.onmessage = function (e) {
    var key = e.data;
    var encryptor = new Worker('libcrypto-v2-encryptor.js');
    encryptor.postMessage(key, [p]);
  };
}

function encrypt(p) {
  p.onmessage = function (e) {
    var key = e.data;
    var decryptor = new Worker('libcrypto-v2-decryptor.js');
    decryptor.postMessage(key, [p]);
  };
}

// support being used as a shared worker as well as a dedicated worker
if (this.onconnect === this) // dedicated worker
```

```
  onmessage = handleMessage;
else // shared worker
  onconnect = function (e) { e.ports[0].onmessage = handleMessage };
```

The little subworkers would then be as follows.

For generating key pairs:

```
onmessage = function (e) {
  var k = _generateKeyPair();
  e.ports[0].postMessage(k[0]);
  e.ports[0].postMessage(k[1]);
  close();
}

function _generateKeyPair() {
  return [Math.random(), Math.random()];
}
```

For encrypting:

```
onmessage = function (e) {
  var key = e.data;
  e.ports[0].onmessage = function (e) {
    var s = e.data;
    postMessage(_encrypt(key, s));
  }
}

function _encrypt(k, s) {
  return 'encrypted-' + k + ' ' + s;
}
```

For decrypting:

```
onmessage = function (e) {
  var key = e.data;
  e.ports[0].onmessage = function (e) {
    var s = e.data;
    postMessage(_decrypt(key, s));
  }
}

function _decrypt(k, s) {
  return s.substr(s.indexOf(' ')+1);
}
```

Notice how the users of the API don't have to even know that this is happening — the API hasn't changed; the library can delegate to subworkers without changing its API, even though it is accepting data using message channels.

View this example online.

10.1.3 Tutorials §

10.1.3.1 Creating a dedicated worker §

This section is non-normative.

Creating a worker requires a URL to a JavaScript file. The `Worker()` constructor is invoked with the URL to that file as its only argument; a worker is then created and returned:

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

If you want your worker script to be interpreted as a module script instead of the default classic script, you need to use a slightly different signature:

```
var worker = new Worker('helper.mjs', { type: "module" });
```

10.1.3.2 Communicating with a dedicated worker §

This section is non-normative.

Dedicated workers use `MessagePort` objects behind the scenes, and thus support all the same features, such as sending structured data, transferring binary data, and transferring other ports.

To receive messages from a dedicated worker, use the `onmessage` event handler IDL attribute on the `Worker` object:

```
worker.onmessage = function (event) { ... };
```

You can also use the `addEventListener()` method.

Note

The implicit `MessagePort` used by dedicated workers has its port message queue implicitly enabled when it is created, so there is no equivalent to the `MessagePort` interface's `start()` method on the `Worker` interface.

To *send* data to a worker, use the `postMessage()` method. Structured data can be sent over this communication channel. To send `ArrayBuffer` objects efficiently (by transferring them rather than cloning them), list them in an array in the second argument.

```
worker.postMessage({
  operation: 'find-edges',
  input: buffer, // an ArrayBuffer object
  threshold: 0.6,
}, [buffer]);
```

To receive a message inside the worker, the `onmessage` event handler IDL attribute is used.

```
onmessage = function (event) { ... };
```

You can again also use the `addEventListener()` method.

In either case, the data is provided in the event object's `data` attribute.

To send messages back, you again use `postMessage()`. It supports the structured data in the same manner.

```
postMessage(event.data.input, [event.data.input]); // transfer the buffer back
```

10.1.3.3 Shared workers §

This section is non-normative.

Shared workers are identified by the URL of the script used to create it, optionally with an explicit name. The name allows multiple instances of a particular shared worker to be started.

Shared workers are scoped by origin. Two different sites using the same names will not collide. However, if a page tries to use the same shared worker name as another page on the same site, but with a different script URL, it will fail.

Creating shared workers is done using the `SharedWorker()` constructor. This constructor takes the URL to the script to use for its first argument, and the name of the worker, if any, as the second argument.

```
var worker = new SharedWorker('service.js');
```

Communicating with shared workers is done with explicit `MessagePort` objects. The object returned by the `SharedWorker()` constructor holds a reference to the port on its `port` attribute.

```
worker.port.onmessage = function (event) { ... };
worker.port.postMessage('some message');
worker.port.postMessage({ foo: 'structured', bar: ['data', 'also', 'possible']});
```

Inside the shared worker, new clients of the worker are announced using the `connect` event. The port for the new client is given by the event object's `source` attribute.

```
onconnect = function (event) {
  var newPort = event.source;
  // set up a listener
  newPort.onmessage = function (event) { ... };
  // send a message back to the port
  newPort.postMessage('ready!'); // can also send structured data, of course
};
```

10.2 Infrastructure §

There are two kinds of workers; dedicated workers, and shared workers. Dedicated workers, once created, are linked to their creator; but message ports can be used to

orker to multiple other browsing contexts or workers. Shared workers, on the other hand, are named, and once created any script

running in the same origin can obtain a reference to that worker and communicate with it.

10.2.1 The global scope §

The global scope is the "inside" of a worker.

10.2.1.1 The `WorkerGlobalScope` common interface §

IDL

```
[Exposed=Worker]
interface WorkerGlobalScope : EventTarget {
  readonly attribute WorkerGlobalScope self;
  readonly attribute WorkerLocation location;
  readonly attribute WorkerNavigator navigator;
  void importScripts(USVString... urls);

  attribute OnErrorEventHandler onerror;
  attribute EventHandler onlanguagechange;
  attribute EventHandler onoffline;
  attribute EventHandler ononline;
  attribute EventHandler onrejectionhandled;
  attribute EventHandler onunhandledrejection;
};
```

MDN ▶WorkerGlobalScope

`WorkerGlobalScope` serves as the base class for specific types of worker global scope objects, including `DedicatedWorkerGlobalScope`, `SharedWorkerGlobalScope`, and `ServiceWorkerGlobalScope`.

A `WorkerGlobalScope` object has an associated **owner set** (a set of `Document` and `WorkerGlobalScope` objects). It is initially empty and populated when the worker is created or obtained.

Note

It is a set, instead of a single owner, to accomodate `SharedWorkerGlobalScope` objects.

A `WorkerGlobalScope` object has an associated **worker set** (a set of `WorkerGlobalScope` objects). It is initially empty and populated when the worker creates or obtains further workers.

A `WorkerGlobalScope` object has an associated **type** ("`classic`" or "`module`"). It is set during creation.

A `WorkerGlobalScope` object has an associated **url** (null or a URL). It is initially null.

A `WorkerGlobalScope` object has an associated **name** (a string). It is set during creation.

Note

The name can have different semantics for each subclass of `WorkerGlobalScope`. For `DedicatedWorkerGlobalScope` instances, it is simply a developer-supplied name, useful mostly for debugging purposes. For `SharedWorkerGlobalScope` instances, it allows obtaining a reference to a common shared worker via the `SharedWorker()` constructor. For `ServiceWorkerGlobalScope` objects, it doesn't make sense (and as such isn't exposed through the JavaScript API at all).

A `WorkerGlobalScope` object has an associated **HTTPS state** (an HTTPS state value). It is initially "`none`".

A `WorkerGlobalScope` object has an associated **referrer policy** (a referrer policy). It is initially the empty string.

A `WorkerGlobalScope` object has an associated **CSP list**, which is a CSP list containing all of the Content Security Policy objects active for the worker. It is initially an empty list.

A `WorkerGlobalScope` object has an associated **module map**. It is a module map, initially empty.

For web developers (non-normative)

***workerGlobal* . `self`**
Returns *workerGlobal*.

***workerGlobal* . `location`**
Returns *workerGlobal*'s `WorkerLocation` object.

***workerGlobal* . `navigator`**
Returns *workerGlobal*'s `WorkerNavigator` object.

***workerGlobal* . `importScripts`(*urls*...)**
Fetches each URL in *urls*, executes them one-by-one in the order they are passed, and then returns (or throws if something went amiss).

The **`self`** attribute must return the `WorkerGlobalScope` object itself.

MDN ▶WorkerGlobalScope/self

The **`location`** attribute must return the `WorkerLocation` object whose associated `WorkerGlobalScope` object is the `Wo`

MDN ▶WorkerGlobalScope/location

Note

While the `WorkerLocation` object is created after the `WorkerGlobalScope` object, this is not problematic as it cannot be observed from script.

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by objects implementing the `WorkerGlobalScope` interface:

Event handler	Event handler event type
onerror	error
onlanguagechange	languagechange
onoffline	offline
ononline	online
onrejectionhandled	rejectionhandled
onunhandledrejection	unhandledrejection

MDN ▶WorkerGlobalScope/onerror
MDN ▶WindowEventHandlers/onlanguagechange
MDN ▶WorkerGlobalScope/onlanguagechange
MDN ▶WorkerGlobalScope/onoffline
MDN ▶WorkerGlobalScope/ononline

10.2.1.2 Dedicated workers and the `DedicatedWorkerGlobalScope` interface §

IDL

```
[Global=(Worker,DedicatedWorker),Exposed=DedicatedWorker]
interface DedicatedWorkerGlobalScope : WorkerGlobalScope {
  [Replaceable] readonly attribute DOMString name;

  void postMessage(any message, sequence<object> transfer);
  void postMessage(any message, optional PostMessageOptions options);

  void close();

  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};
```

MDN ▶DedicatedWorkerGlobalScope

`DedicatedWorkerGlobalScope` objects act as if they had an implicit `MessagePort` associated with them. This port is part of a channel that is set up when the worker is created, but it is not exposed. This object must never be garbage collected before the `DedicatedWorkerGlobalScope` object.

All messages received by that port must immediately be retargeted at the `DedicatedWorkerGlobalScope` object.

For web developers (non-normative)

***dedicatedWorkerGlobal* . `name`**

Returns *dedicatedWorkerGlobal*'s name, i.e. the value given to the `Worker` constructor. Primarily useful for debugging.

***dedicatedWorkerGlobal* . `postMessage`(*message* [, *transfer*])**

***dedicatedWorkerGlobal* . `postMessage`(*message* [, { `transfer` }])**

Clones *message* and transmits it to the `Worker` object associated with *dedicatedWorkerGlobal*. *transfer* can be passed as a list of objects that are to be transferred rather than cloned.

***dedicatedWorkerGlobal* . `close`()**

Aborts *dedicatedWorkerGlobal*.

The **`name`** attribute must return the `DedicatedWorkerGlobalScope` object's name. Its value represents the name used primarily for debugging purposes.

MDN ▶DedicatedWorkerGlobalScope/name

The **`postMessage(message, transfer)`** and **`postMessage(message, options)`** methods on `Dedi` invoked, it immediately invoked the respective `postMessage(message, transfer)` and `postMessage` arguments, and returned the same return value.

MDN ▶DedicatedWorkerGlobalScope/postMessage

To **close a worker**, given a *workerGlobal*, run these steps:

1. Discard any tasks that have been added to *workerGlobal*'s event loop's task queues.
2. Set *workerGlobal*'s closing flag to true. (This prevents any further tasks from being queued.)

The **`close()`** method, when invoked, must close a worker with this `DedicatedWorkerGlobalScope` object.

MDN ▶DedicatedWorkerGlobalScope/close

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by objects implementing the `DedicatedWorkerGlobalScope` interface:

Event handler	Event handler event type
onmessage	message
onmessageerror	messageerror

MDN ▶DedicatedWorkerGlobalScope/onmessage
MDN ▶DedicatedWorkerGlobalScope/onmessageerror

…on cache networking model, a dedicated worker is an extension of the cache host from which it was created.

10.2.1.3 Shared workers and the `SharedWorkerGlobalScope` interface §

IDL

```
[Global=(Worker,SharedWorker),Exposed=SharedWorker]
interface SharedWorkerGlobalScope : WorkerGlobalScope {
  [Replaceable] readonly attribute DOMString name;

  void close();

  attribute EventHandler onconnect;
};
```

MDN ▶SharedWorkerGlobalScope

A `SharedWorkerGlobalScope` object has an associated **constructor origin**, and **constructor url**. They are initialized when the `SharedWorkerGlobalScope` object is created, in the run a worker algorithm.

Shared workers receive message ports through `connect` events on their `SharedWorkerGlobalScope` object for each connection.

For web developers (non-normative)

***sharedWorkerGlobal* . `name`**

Returns *sharedWorkerGlobal*'s name, i.e. the value given to the `SharedWorker` constructor. Multiple `SharedWorker` objects can correspond to the same shared worker (and `SharedWorkerGlobalScope`), by reusing the same name.

***sharedWorkerGlobal* . `close`()**

Aborts *sharedWorkerGlobal*.

The **`name`** attribute must return the `SharedWorkerGlobalScope` object's name. Its value represents the name that can be used to obtain a reference to the worker using the `SharedWorker` constructor.

MDN ▶SharedWorkerGlobalScope/name

The **`close()`** method, when invoked, must close a worker with this `SharedWorkerGlobalScope` object.

MDN ▶SharedWorkerGlobalScope/close

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by objects implementing the `SharedWorkerGlobalScope` interface:

Event handler	Event handler event type
`onconnect`	`connect`

MDN ▶SharedWorkerGlobalScope/onconnect

10.2.2 The event loop §

A worker event loop's task queues only have events, callbacks, and networking activity as tasks. These worker event loops are created by the run a worker algorithm.

Each `WorkerGlobalScope` object has a **closing** flag, which must be initially false, but which can get set to true by the algorithms in the processing model section below.

Once the `WorkerGlobalScope`'s closing flag is set to true, the event loop's task queues must discard any further tasks that would be added to them (tasks already on the queue are unaffected except where otherwise specified). Effectively, once the closing flag is true, timers stop firing, notifications for all pending background operations are dropped, etc.

10.2.3 The worker's lifetime §

Workers communicate with other workers and with browsing contexts through message channels and their `MessagePort` objects.

Each `WorkerGlobalScope` object *worker global scope* has a list of **the worker's ports**, which consists of all the `MessagePort` objects that are entangled with another port and that have one (but only one) port owned by *worker global scope*. This list includes the implicit `MessagePort` in the case of dedicated workers.

Given an environment settings object *o* when creating or obtaining a worker, the **relevant owner to add** depends on the type of global object specified by *o*. If *o* specifies a global object that is a `WorkerGlobalScope` object (i.e., if we are creating a nested worker), then the relevant owner is that global object. Otherwise, *o* specifies a global object that is a `Window` object, and the relevant owner is the responsible document specified by *o*.

A worker is said to be a **permissible worker** if its `WorkerGlobalScope`'s owner set is not empty or:

- its owner set has been empty for no more than a short user-agent-defined timeout value,
- its `WorkerGlobalScope` object is a `SharedWorkerGlobalScope` object (i.e., the worker is a shared worker), and
- the user agent has a browsing context whose `Document` object is not completely loaded.

Note

The second part of this definition allows a shared worker to survive for a short time while a page is loading, in case that page is going to contact the shared worker again. This can be used by user agents as a way to avoid the cost of restarting a shared worker used by a site when the user is navigating from page to page within that site.

A worker is said to be an **active needed worker** if any its owners are either `Document` objects that are fully active or active needed workers.

A worker is said to be a **protected worker** if it is an active needed worker and either it has outstanding timers, database transactions, or network connections, or its list of the worker's ports is not empty, or its `WorkerGlobalScope` is actually a `SharedWorkerGlobalScope` object (i.e. the worker is a shared worker).

A worker is said to be a **suspendable worker** if it is not an active needed worker but it is a permissible worker.

10.2.4 Processing model §

When a user agent is to **run a worker** for a script with `Worker` or `SharedWorker` object *worker*, URL *url*, environment settings object *outside settings*, `MessagePort` *outside port*, and a `WorkerOptions` dictionary *options*, it must run the following steps.

1. Create a separate parallel execution environment (i.e. a separate thread or process or equivalent construct), and run the rest of these steps in that context.

   For the purposes of timing APIs, this is the **official moment of creation** of the worker.

2. Let *is shared* be true if *worker* is a `SharedWorker` object, and false otherwise.

3. Let *owner* be the relevant owner to add given *outside settings*.

4. Let *parent worker global scope* be null.

5. If *owner* is a `WorkerGlobalScope` object (i.e., we are creating a nested worker), then set *parent worker global scope* to *owner*.

6. Let *realm execution context* be the result of creating a new JavaScript realm with the following customizations:

   - For the global object, if *is shared* is true, create a new `SharedWorkerGlobalScope` object. Otherwise, create a new `DedicatedWorkerGlobalScope` object.

7. Let *worker global scope* be the global object of *realm execution context*'s Realm component.

   Note

   This is the `DedicatedWorkerGlobalScope` or `SharedWorkerGlobalScope` object created in the previous step.

8. Set up a worker environment settings object with *realm execution context* and *outside settings*, and let *inside settings* be the result.

9. Set *worker global scope*'s name to the value of *options*'s `name` member.

10. If *is shared* is true, then:

    1. Set *worker global scope*'s constructor origin to *outside settings*'s origin.

    2. Set *worker global scope*'s constructor url to *url*.

11. Let *destination* be "`sharedworker`" if *is shared* is true, and "`worker`" otherwise.

12. Obtain *script* by switching on the value of *options*'s `type` member:

    ↪ **"classic"**

    Fetch a classic worker script given *url*, *outside settings*, *destination*, and *inside settings*.

    ↪ **"module"**

    Fetch a module worker script graph given *url*, *outside settings*, *destination*, the value of the `credentials` member of *options*, and *inside settings*.

    In both cases, to perform the fetch given *request*, perform the following steps if the *is top-level* flag is set:

    1. Set *request*'s reserved client to *inside settings*.

    2. Fetch *request*, and asynchronously wait to run the remaining steps as part of fetch's process response for the response *response*.

    3. Set *worker global scope*'s url to *response*'s url.

    4. Set *worker global scope*'s HTTPS state to *response*'s HTTPS state.

    5. Set *worker global scope*'s referrer policy to the result of parsing the `Referrer-Policy` header of *response*.

    6. Execute the Initialize a `global object`'s CSP list algorithm on *worker global scope* and *response*. [CSP]

    7. Asynchronously complete the perform the fetch steps with *response*.

    If the algorithm asynchronously completes with null, then:

    1. Queue a task to fire an event named `error` at *worker*.

    2. Run the environment discarding steps for *inside settings*.

    3. Return.

    Otherwise, continue the rest of these steps after the algorithm's asynchronous completion, with *script* being the asynchronous completion value.

13. Associate *worker* with *worker global scope*.

14. Create a new `MessagePort` object whose owner is *inside settings*. Let *inside port* be this new object.

15. Associate *inside port* with *worker global scope*.

16. Entangle *outside port* and *inside port*.
17. Append *owner* to *worker global scope*'s owner set.
18. If *parent worker global scope* is not null, then append *worker global scope* to *parent worker global scope*'s worker set.
19. Set *worker global scope*'s type to the value of the `type` member of *options*.
20. Create a new `WorkerLocation` object and associate it with *worker global scope*.
21. **Closing orphan workers**: Start monitoring the worker such that no sooner than it stops being a protected worker, and no later than it stops being a permissible worker, *worker global scope*'s closing flag is set to true.
22. **Suspending workers**: Start monitoring the worker, such that whenever *worker global scope*'s closing flag is false and the worker is a suspendable worker, the user agent suspends execution of script in that worker until such time as either the closing flag switches to true or the worker stops being a suspendable worker.
23. Set *inside settings*'s execution ready flag.
24. If *script* is a classic script, then run the classic script *script*. Otherwise, it is a module script; run the module script *script*.

    Note

    In addition to the usual possibilities of returning a value or failing due to an exception, this could be prematurely aborted by the terminate a worker algorithm defined below.
25. Enable *outside port*'s port message queue.
26. If *is shared* is false, enable the port message queue of the worker's implicit port.
27. If *is shared* is true, then queue a task, using the DOM manipulation task source, to fire an event named `connect` at *worker global scope*, using `MessageEvent`, with the `data` attribute initialized to the empty string, the `ports` attribute initialized to a new frozen array containing *inside port*, and the `source` attribute initialized to *inside port*.
28. Enable the client message queue of the `ServiceWorkerContainer` object whose associated service worker client is *worker global scope*'s relevant settings object.
29. **Event loop**: Run the responsible event loop specified by *inside settings* until it is destroyed.

    Note

    The handling of events or the execution of callbacks by tasks run by the event loop might get prematurely aborted by the terminate a worker algorithm defined below.

    Note

    The worker processing model remains on this step until the event loop is destroyed, which happens after the closing flag is set to true, as described in the event loop processing model.
30. Empty the *worker global scope*'s list of active timers.
31. Disentangle all the ports in the list of the worker's ports.
32. Empty *worker global scope*'s owner set.

When a user agent is to **terminate a worker** it must run the following steps in parallel with the worker's main loop (the "run a worker" processing model defined above):

1. Set the worker's `WorkerGlobalScope` object's closing flag to true.
2. If there are any tasks queued in the `WorkerGlobalScope` object's event loop's task queues, discard them without processing them.
3. Abort the script currently running in the worker.
4. If the worker's `WorkerGlobalScope` object is actually a `DedicatedWorkerGlobalScope` object (i.e. the worker is a dedicated worker), then empty the port message queue of the port that the worker's implicit port is entangled with.

User agents may invoke the terminate a worker algorithm when a worker stops being an active needed worker and the worker continues executing even after its closing flag was set to true.

The task source for the tasks mentioned above is the DOM manipulation task source.

10.2.5 Runtime script errors §

Whenever an uncaught runtime script error occurs in one of the worker's scripts, if the error did not occur while handling a previous script error, the user agent must report the error for that script, with the position (line number and column number) where the error occurred, using the `WorkerGlobalScope` object as the target.

If the error is still *not handled* afterwards, the error may be reported to a developer console.

For dedicated workers, if the error is still *not handled* afterwards, the user agent must queue a task to run these steps:

1. Let *notHandled* be the result of firing an event named `error` at the `Worker` object associated with the worker, using `ErrorEvent`, with the `cancelable` attribute initialized to true, the `message`, `filename`, `lineno`, and `colno` attributes initialized appropriately, and the `error` attribute initialized to null.
2. If *notHandled* is true, then the user agent must act as if the uncaught runtime script error had occurred in the global scope that the `Worker` object is in, thus repeating the entire runtime script error reporting process one level up.

If the implicit port connecting the worker to its `Worker` object has been disentangled (i.e. if the parent worker has been terminated), then the user agent must act as if the `Worker` object had no `error` event handler and as if that worker's `onerror` attribute was null, but must otherwise act as described above.

Note

Thus, error reports propagate up to the chain of dedicated workers up to the original `Document`, even if some of the workers along this chain have been terminated and garbage collected.

The task source for the task mentioned above is the DOM manipulation task source.

10.2.6 Creating workers §

10.2.6.1 The `AbstractWorker` mixin §

IDL

```
interface mixin AbstractWorker {
  attribute EventHandler onerror;
};
```

MDN ▶AbstractWorker

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by objects implementing the `AbstractWorker` interface:

Event handler	Event handler event type
`onerror`	`error`

MDN ▶AbstractWorker/onerror

10.2.6.2 Script settings for workers §

When the user agent is required to **set up a worker environment settings object**, given a JavaScript execution context *execution context* and environment settings object *outside settings*, it must run the following steps:

1. Let *inherited responsible browsing context* be *outside settings*'s responsible browsing context.
2. Let *inherited origin* be *outside settings*'s origin.
3. Let *worker event loop* be a newly created worker event loop.
4. Let *realm* be the value of *execution context*'s Realm component.
5. Let *worker global scope* be *realm*'s global object.
6. Let *settings object* be a new environment settings object whose algorithms are defined as follows:

   The realm execution context

   Return *execution context*.

   The module map

   Return *worker global scope*'s module map.

   The responsible browsing context

   Return *inherited responsible browsing context*.

   The responsible event loop

   Return *worker event loop*.

   The responsible document

   Not applicable (the responsible event loop is not a window event loop).

   The API URL character encoding

   Return UTF-8.

   The API base URL

   Return *worker global scope*'s url.

   The origin

   Return a unique opaque origin if *worker global scope*'s url's scheme is "`data`", and *inherited origin* otherwise.

   The HTTPS state

   Return *worker global scope*'s HTTPS state.

The referrer policy

Return *worker global scope*'s referrer policy.

7. Set *settings object*'s id to a new unique opaque string, *settings object*'s creation URL to *worker global scope*'s url, *settings object*'s target browsing context to null, and *settings object*'s active service worker to null.
8. Set *realm*'s [[HostDefined]] field to *settings object*.
9. Return *settings object*.

10.2.6.3 Dedicated workers and the `Worker` interface §

IDL

```
[Constructor(USVString scriptURL, optional WorkerOptions options), Exposed=(Window,Worker)]
interface Worker : EventTarget {
  void terminate();

  void postMessage(any message, sequence<object> transfer);
  void postMessage(any message, optional PostMessageOptions options);
  attribute EventHandler onmessage;
  attribute EventHandler onmessageerror;
};

dictionary WorkerOptions {
  WorkerType type = "classic";
  RequestCredentials credentials = "same-origin"; // credentials is only used if type is "module"
  DOMString name = "";
};

enum WorkerType { "classic", "module" };

Worker includes AbstractWorker;
```

MDN ▶Worker

For web developers (non-normative)

***worker* = new `Worker`(*scriptURL* [, *options*])**

Returns a new `Worker` object. *scriptURL* will be fetched and executed in the background, creating a new global environment for which *worker* represents the communication channel. *options* can be used to define the name of that global environment via the `name` option, primarily for debugging purposes. It can also ensure this new global environment supports JavaScript modules (specify `type: "module"`), and if that is specified, can also be used to specify how *scriptURL* is fetched through the `credentials` option.

***worker* . `terminate`()**

Aborts *worker*'s associated global environment.

***worker* . `postMessage`(*message* [, *transfer*])**

***worker* . `postMessage`(*message* [, { `transfer` }])**

Clones *message* and transmits it to *worker*'s global environment. *transfer* can be passed as a list of objects that are to be transferred rather than cloned.

The `terminate()` method, when invoked, must cause the terminate a worker algorithm to be run on the worker with which the object is

MDN ▶Worker/terminate

`Worker` objects act as if they had an implicit `MessagePort` associated with them. This port is part of a channel that is set up when the worker is created, but it is not exposed. This object must never be garbage collected before the `Worker` object.

All messages received by that port must immediately be retargeted at the `Worker` object.

The `postMessage(message, transfer)` and `postMessage(message, options)` methods on `Worker` objects act as if, when the respective `postMessage(message, transfer)` and `postMessage(message, options)` on the port, with the same argum value.

MDN ▶Worker/postMessage

Example

The `postMessage()` method's first argument can be structured data:

```
worker.postMessage({opcode: 'activate', device: 1938, parameters: [23, 102]});
```

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by objects implementing the `Worker` interface:

Event handler	Event handler event type
`onmessage`	`message`
`onmessageerror`	`messageerror`

MDN ▶Worker/onmessage
MDN ▶Worker/onmessageerror

When the `Worker(scriptURL, options)` constructor is invoked, the user agent must run the following steps:

MDN ▶Worker/Worker

1. The user agent may throw a "SecurityError" DOMException if the request violates a policy decision (e.g. if the user agent is configured to not allow the page to start dedicated workers).
2. Let *outside settings* be the current settings object.
3. Parse the *scriptURL* argument relative to *outside settings*.
4. If this fails, throw a "SyntaxError" DOMException.
5. Let *worker URL* be the resulting URL record.

   Note

   Any same-origin URL (including blob: URLs) can be used. data: URLs can also be used, but they create a worker with an opaque origin.
6. Let *worker* be a new Worker object.
7. Create a new MessagePort object whose owner is *outside settings*. Let this be the *outside port*.
8. Associate the *outside port* with *worker*.
9. Run this step in parallel:
   1. Run a worker given *worker*, *worker URL*, *outside settings*, *outside port*, and *options*.
10. Return *worker*.

10.2.6.4 Shared workers and the SharedWorker interface §

IDL

```
[Constructor(USVString scriptURL, optional (DOMString or WorkerOptions) options),
 Exposed=(Window,Worker)]
interface SharedWorker : EventTarget {
  readonly attribute MessagePort port;
};
SharedWorker includes AbstractWorker;
```

MDN ▶SharedWorker

For web developers (non-normative)

***sharedWorker* = new SharedWorker(*scriptURL* [, *name*])**

Returns a new SharedWorker object. *scriptURL* will be fetched and executed in the background, creating a new global environment for which *sharedWorker* represents the communication channel. *name* can be used to define the name of that global environment.

***sharedWorker* = new SharedWorker(*scriptURL* [, *options*])**

Returns a new SharedWorker object. *scriptURL* will be fetched and executed in the background, creating a new global environment for which *sharedWorker* represents the communication channel. *options* can be used to define the name of that global environment via the `name` option. It can also ensure this new global environment supports JavaScript modules (specify `type: "module"`), and if that is specified, can also be used to specify how *scriptURL* is fetched through the `credentials` option.

***sharedWorker* . port**

Returns *sharedWorker*'s MessagePort object which can be used to communicate with the global environment.

The **port** attribute must return the value it was assigned by the object's constructor. It represents the MessagePort for communicating

MDN ▶SharedWorker/port

A user agent has an associated **shared worker manager** which is the result of starting a new parallel queue.

Note

Each user agent has a single shared worker manager for simplicity. Implementations could use one per origin; that would not be observably different and enables more concurrency.

When the **SharedWorker(*scriptURL*, *options*)** constructor is invoked:

MDN ▶SharedWorker/SharedWorker

1. Optionally, throw a "SecurityError" DOMException if the request violates a policy decision (e.g. if the user agent is configured to not allow the page to start shared workers).
2. If *options* is a DOMString, set *options* to a new WorkerOptions dictionary whose `name` member is set to the value of *options* and whose other members are set to their default values.
3. Let *outside settings* be the current settings object.
4. Parse *scriptURL* relative to *outside settings*.
5. If this fails, throw a "SyntaxError" DOMException.
6. Otherwise, let *urlRecord* be the resulting URL record.

   Note

   Any same-origin URL (including blob: URLs) can be used. data: URLs can also be used, but they create a worker with an opaque origin.
7. Let *worker* be a new SharedWorker object.

File an issue about the selected text ort object whose owner is *outside settings*. Let this be the *outside port*.

9. Assign *outside port* to the `port` attribute of *worker*.
10. Let *callerIsSecureContext* be the result of executing Is environment settings object a secure context? on *outside settings*.
11. Enqueue the following steps to the shared worker manager:
    1. Let *worker global scope* be null.
    2. If there exists a `SharedWorkerGlobalScope` object whose closing flag is false, constructor origin is same origin with *outside settings*'s origin, constructor url equals *urlRecord*, and name equals the value of *options*'s `name` member, then set *worker global scope* to that `SharedWorkerGlobalScope` object.

       Note

       `data:` URLs create a worker with an opaque origin. Both the constructor origin and constructor url are compared so the same `data:` URL can be used within an origin to get to the same `SharedWorkerGlobalScope` object, but cannot be used to bypass the same origin restriction.
    3. If *worker global scope* is not null, but the user agent has been configured to disallow communication between the worker represented by the *worker global scope* and the scripts whose settings object is *outside settings*, then set *worker global scope* to null.

       Note

       For example, a user agent could have a development mode that isolates a particular top-level browsing context from all other pages, and scripts in that development mode could be blocked from connecting to shared workers running in the normal browser mode.
    4. If *worker global scope* is not null, then run these subsubsteps:
       1. Let *settings object* be the relevant settings object for *worker global scope*.
       2. Let *workerIsSecureContext* be the result of executing Is environment settings object a secure context? on *settings object*.
       3. If *workerIsSecureContext* is not *callerIsSecureContext*, then queue a task to fire an event named `error` at *worker* and abort these subsubsteps. [SECURE-CONTEXTS]
       4. Associate *worker* with *worker global scope*.
       5. Create a new `MessagePort` object whose owner is *settings object*. Let this be the *inside port*.
       6. Entangle *outside port* and *inside port*.
       7. Queue a task, using the DOM manipulation task source, to fire an event named `connect` at *worker global scope*, using `MessageEvent`, with the `data` attribute initialized to the empty string, the `ports` attribute initialized to a new frozen array containing only *inside port*, and the `source` attribute initialized to *inside port*.
       8. Append the relevant owner to add given *outside settings* to *worker global scope*'s owner set.
       9. If *outside settings*'s global object is a `WorkerGlobalScope` object, then append *worker global scope* to *outside settings*'s global object's worker set.
    5. Otherwise, in parallel, run a worker given *worker*, *urlRecord*, *outside settings*, *outside port*, and *options*.
12. Return *worker*.

10.2.7 Concurrent hardware capabilities §

IDL

```
interface mixin NavigatorConcurrentHardware {
  readonly attribute unsigned long long hardwareConcurrency;
};
```

MDN ▶NavigatorConcurrentHardware

For web developers (non-normative)

***self*.`navigator`.`hardwareConcurrency`**

Returns the number of logical processors potentially available to the user agent.

The **`navigator.hardwareConcurrency`** attribute's getter must return a number between 1 and ... agent. If this cannot be determined, the getter must return 1.

MDN ▶NavigatorConcurrentHardware/hardwareConcurrency

User agents should err toward exposing the number of logical processors available, using lower values only in cases where there are user-agent specific limits in pl... (such as a limitation on the number of workers that can be created) or when the user agent desires to limit fingerprinting possibilities.

10.3 APIs available to workers §

10.3.1 Importing scripts and libraries §

When a script invokes the **`importScripts(urls)`** method on a `WorkerGlobalScope` object, the user agent must ir... `WorkerGlobalScope` object and *urls*.

MDN ▶WorkerGlobalScope/importScripts

To **import scripts into worker global scope**, given a `WorkerGlobalScope` object *worker global scope* and a `sequence<DOMString>` *urls*, run these steps. The ...omized by supplying custom perform the fetch hooks, which if provided will be used when invoking fetch a classic worker-imported

script.

1. If *worker global scope*'s type is "module", throw a TypeError exception.
2. Let *settings object* be the current settings object.
3. If *urls* is empty, return.
4. Parse each value in *urls* relative to *settings object*. If any fail, throw a "SyntaxError" DOMException.
5. For each *url* in the resulting URL records, run these substeps:
    1. Fetch a classic worker-imported script given *url* and *settings object*, passing along any custom perform the fetch steps provided. If this succeeds, let *script* be the result. Otherwise, rethrow the exception.
    2. Run the classic script *script*, with the rethrow errors argument set to true.

    Note

    script *will run until it either returns, fails to parse, fails to catch an exception, or gets prematurely aborted by the terminate a worker algorithm defined above.*

    If an exception was thrown or if the script was prematurely aborted, then abort all these steps, letting the exception or aborting continue to be processed by the calling script.

Note

Service Workers *is an example of a specification that runs this algorithm with its own options for the perform the fetch hook. [SW]*

10.3.2 The WorkerNavigator interface §

The **navigator** attribute of the WorkerGlobalScope interface must return an instance of the WorkerNavigator interface, which represents the identity and state of the user agent (the client):

MDN ▶WorkerGlobalScope/navigator

IDL

```
[Exposed=Worker]
interface WorkerNavigator {};
WorkerNavigator includes NavigatorID;
WorkerNavigator includes NavigatorLanguage;
WorkerNavigator includes NavigatorOnLine;
WorkerNavigator includes NavigatorConcurrentHardware;
```

MDN ▶WorkerNavigator

10.3.3 The WorkerLocation interface §

IDL

```
[Exposed=Worker]
interface WorkerLocation {
  stringifier readonly attribute USVString href;
  readonly attribute USVString origin;
  readonly attribute USVString protocol;
  readonly attribute USVString host;
  readonly attribute USVString hostname;
  readonly attribute USVString port;
  readonly attribute USVString pathname;
  readonly attribute USVString search;
  readonly attribute USVString hash;
};
```

MDN ▶WorkerLocation

A WorkerLocation object has an associated **WorkerGlobalScope object** (a WorkerGlobalScope object).

The **href** attribute's getter must return the associated WorkerGlobalScope object's url, serialized.

The **origin** attribute's getter must return the serialization of the associated WorkerGlobalScope object's url's origin.

The **protocol** attribute's getter must return the associated WorkerGlobalScope object's url's scheme, followed by ":".

The **host** attribute's getter must run these steps:

1. Let *url* be the associated WorkerGlobalScope object's url.
2. If *url*'s host is null, return the empty string.
3. If *url*'s port is null, return *url*'s host, serialized.
4. Return *url*'s host, serialized, followed by ":" and *url*'s port, serialized.

The **hostname** attribute's getter must run these steps:

1. Let *host* be the associated WorkerGlobalScope object's url's host.

2. If *host* is null, return the empty string.
3. Return *host*, serialized.

The **`port`** attribute's getter must run these steps:

1. Let *port* be the associated `WorkerGlobalScope` object's url's port.
2. If *port* is null, return the empty string.
3. Return *port*, serialized.

The **`pathname`** attribute's getter must run these steps:

1. Let *url* be the associated `WorkerGlobalScope` object's url.
2. If *url*'s cannot-be-a-base-URL flag is set, return the first string in *url*'s path.
3. Return "`/`", followed by the strings in *url*'s path (including empty strings), separated from each other by "`/`".

The **`search`** attribute's getter must run these steps:

1. Let *query* be the associated `WorkerGlobalScope` object's url's query.
2. If *query* is either null or the empty string, return the empty string.
3. Return "`?`", followed by *query*.

The **`hash`** attribute's getter must run these steps:

1. Let *fragment* be the associated `WorkerGlobalScope` object's url's fragment.
2. If *fragment* is either null or the empty string, return the empty string.
3. Return "`#`", followed by *fragment*.

MDN ▶Web Storage API
MDN ▶Web Storage API/Using the Web Storage API

← 9.4 Cross-document messaging — Table of Contents — 11 Web storage →