Chapter 63: Workers

Section 63.1: Web Worker

A web worker is a simple way to run scripts in background threads as the worker thread can perform tasks (including I/O tasks using XMLHttpRequest) without interfering with the user interface. Once created, a worker can send messages which can be different data types (except functions) to the JavaScript code that created it by posting messages to an event handler specified by that code (and vice versa.)

Workers can be created in a few ways.

The most common is from a simple URL:

```
var webworker = new Worker("./path/to/webworker.js");
```

It's also possible to create a Worker dynamically from a string using `URL.createObjectURL()`:

```
var workerData = "function someFunction() {}; console.log('More code');";

var blobURL = URL.createObjectURL(new Blob(["(" + workerData + ")"], { type: "text/javascript" }));

var webworker = new Worker(blobURL);
```

The same method can be combined with `Function.toString()` to create a worker from an existing function:

```
var workerFn = function() {
    console.log("I was run");
};

var blobURL = URL.createObjectURL(new Blob(["(" + workerFn.toString() + ")"], { type:
"text/javascript" }));

var webworker = new Worker(blobURL);
```

Section 63.2: A simple service worker

main.js

> A service worker is an event-driven worker registered against an origin and a path. It takes the form of a JavaScript file that can control the web page/site it is associated with, intercepting and modifying navigation and resource requests, and caching resources in a very granular fashion to give you complete control over how your app behaves in certain situations (the most obvious one being when the network is not available.)

Source: MDN

Few Things:

1. It's a JavaScript Worker, so it can't access the DOM directly
2. It's a programmable network proxy
3. It will be terminated when not in use and restarted when it's next needed
4. A service worker has a lifecycle which is completely separate from your web page
5. HTTPS is Needed

This code that will be executed in the Document context, (or) this JavaScript will be included in your page via a `<script>` tag.

```
// we check if the browser supports ServiceWorkers
if ('serviceWorker' in navigator) {
  navigator
    .serviceWorker
    .register(
      // path to the service worker file
      'sw.js'
    )
    // the registration is async and it returns a promise
    .then(function (reg) {
      console.log('Registration Successful');
    });
}
```

sw.js

This is the service worker code and is executed in the ServiceWorker Global Scope.

```
self.addEventListener('fetch', function (event) {
  // do nothing here, just log all the network requests
  console.log(event.request.url);
});
```

Section 63.3: Register a service worker

```
// Check if service worker is available.
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/sw.js').then(function(registration) {
    console.log('SW registration succeeded with scope:', registration.scope);
  }).catch(function(e) {
    console.log('SW registration failed with error:', e);
  });
}
```

- You can call `register()` on every page load. If the SW is already registered, the browser provides you with instance that is already running
- The SW file can be any name. `sw.js` is common.
- The location of the SW file is important because it defines the SW's scope. For example, an SW file at `/js/sw.js` can only intercept `fetch` requests for files that begin with `/js/`. For this reason you usually see the SW file at the top-level directory of the project.

Section 63.4: Communicating with a Web Worker

Since workers run in a separate thread from the one that created them, communication needs to happen via `postMessage`.

Note: Because of the different export prefixes, some browsers have `webkitPostMessage` instead of `postMessage`. You should override `postMessage` to make sure workers "work" (no pun intended) in the most places possible:

```
worker.postMessage = (worker.webkitPostMessage || worker.postMessage);
```

From the main thread (parent window):

```
// Create a worker
```

```
var webworker = new Worker("./path/to/webworker.js");

// Send information to worker
webworker.postMessage("Sample message");

// Listen for messages from the worker
webworker.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from the worker
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
});
```

From the worker, in `webworker.js`:

```
// Send information to the main thread (parent window)
self.postMessage(["foo", "bar", "baz"]);

// Listen for messages from the main thread
self.addEventListener("message", function(event) {
    // `event.data` contains the value or object sent from main
    console.log("Message from parent:", event.data); // "Sample message"
});
```

Alternatively, you can also add event listeners using `onmessage`:

From the main thread (parent window):

```
webworker.onmessage = function(event) {
    console.log("Message from worker:", event.data); // ["foo", "bar", "baz"]
}
```

From the worker, in `webworker.js`:

```
self.onmessage = function(event) {
    console.log("Message from parent:", event.data); // "Sample message"
}
```

Section 63.5: Terminate a worker

Once you are done with a worker you should terminate it. This helps to free up resources for other applications on the user's computer.

Main Thread:

```
// Terminate a worker from your application.
worker.terminate();
```

Note: The `terminate` method is not available for service workers. It will be terminated when not in use, and restarted when it's next needed.

Worker Thread:

```
// Have a worker terminate itself.
self.close();
```

Section 63.6: Populating your cache

After your service worker is registered, the browser will try to install & later activate the service worker.

Install event listener

```
this.addEventListener('install', function(event) {
    console.log('installed');
});
```

Caching

One can use this install event returned to cache the assets needed to run the app offline. Below example uses the cache api to do the same.

```
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        /* Array of all the assets that needs to be cached */
        '/css/style.css',
        '/js/app.js',
        '/images/snowTroopers.jpg'
      ]);
    })
  );
});
```

Section 63.7: Dedicated Workers and Shared Workers

Dedicated Workers

A dedicated web worker is only accessible by the script that called it.

Main application:

```
var worker = new Worker('worker.js');
worker.addEventListener('message', function(msg) {
    console.log('Result from the worker:', msg.data);
});
worker.postMessage([2,3]);
```

worker.js:

```
self.addEventListener('message', function(msg) {
    console.log('Worker received arguments:', msg.data);
    self.postMessage(msg.data[0] + msg.data[1]);
});
```

Shared Workers

A shared worker is accessible by multiple scripts — even if they are being accessed by different windows, iframes or even workers.

Creating a shared worker is very similar to how to create a dedicated one, but instead of the straight-forward communication between the main thread and the worker thread, you'll have to communicate via a port object, i.e., an explicit port has to be opened so multiple scripts can use it to communicate with the shared worker. (Note that dedicated workers do this implicitly)

Main application

```
var myWorker = new SharedWorker('worker.js');
myWorker.port.start();  // open the port connection

myWorker.port.postMessage([2,3]);
```

worker.js

```
self.port.start(); open the port connection to enable two-way communication

self.onconnect = function(e) {
    var port = e.ports[0];  // get the port

    port.onmessage = function(e) {
        console.log('Worker received arguments:', e.data);
        port.postMessage(e.data[0] + e.data[1]);
    }
}
```

Note that setting up this message handler in the worker thread also implicitly opens the port connection back to the parent thread, so the call to `port.start()` is not actually needed, as noted above.