DZone Refcardz

CONTENTS INCLUDE:

HTML5 Web Workers

Multithreaded JavaScript for High-Performance Web Apps

By: Gerard Gallant

INTRODUCTION

Prior to HTML5 Web Workers, all JavaScript code would run in the same thread as the user interface of the browser window.

The result was that all long-running scripts would cause the browser window to become unresponsive to user input until the processing finished. If the script took long enough, the browser would even prompt the user to see if they wanted to stop the unresponsive script.

Web Workers came to the rescue by allowing JavaScript code to run in a separate thread from the browser window's UI thread. This allows long-running scripts to do their processing without interfering with the responsiveness of the browser's UI (the user can still click on buttons and scroll content for example).

A couple of things to be aware of with Web Workers is that, because they are running in a separate thread from the UI, the worker threads have no access to the DOM or other UI functionality. In addition, Web Workers are not intended to be used in large numbers and are expected to be long-lived since they have a high start-up performance cost and high memory cost per worker instance.

Types of Workers

There are two types of workers available: dedicated workers and shared workers.

Dedicated workers are linked to their creator and are a bit simpler to work with than shared workers.

Shared workers, on the other hand, allow any script from the same origin/domain to obtain a reference to the worker and communicate with it.

Since worker threads take time to start initially and use a lot of memory per worker, being able to share a single worker rather than creating one for each open window can improve performance.

The trade off, however, is that shared workers are a bit more involved to set up and work with than dedicated workers.

Shared Worker Scope

You may be curious about what happens when the window that created a shared worker is closed. Is the worker thread still accessible by other windows from the same origin?

A shared worker will remain active as long as one window has a connection to it. For example:

Window 1 creates a shared worker and Window 2 connects to it.

Later, Window 1 is closed but, because Window 2 is still connected, the shared worker thread remains active even though Window 2 didn't originally create it.

Browser Support

http://caniuse.com: This handy website can tell you if a feature is supported by a browser, and in which version of the browser the desired feature became available. This can save considerable testing time needed to determine if the desired feature is available in the browsers you intend to support. The site is updated regularly.

At the time of this article's writing, dedicated workers have the most browser support including:

Internet Explorer 10+, Firefox, Chrome, Safari, Opera, iOS Safari, BlackBerry, Opera Mobile, Chrome for Android, and Firefox for Android.

Shared worker support currently includes:

Chrome, Safari, Opera, iOS Safari, BlackBerry, and Opera Mobile

CREATING WORKERS

Test if the feature exists

Unless you know for certain that your users will be using a specific browser version, there is a chance that your website will be accessed by a browser that does not have the HTML5 feature you wish to use.

It is always best to test if the browser supports the feature you intend to use before you try using the feature.

Both dedicated and shared workers will exist as objects on the global window object if the browser supports them.

The following is an example of how you can test to see if the browser supports dedicated or shared workers:

```
if (window.Worker) { /* Dedicated Workers are supported */ }
if (window.SharedWorker) { /* Shared Workers are supported */ }
```

Creating a Worker

To create dedicated worker, you simply create a new instance of the Worker object passing in a string that specifies the location of the JavaScript file that contains the worker thread's main code.

The following is an example of how a dedicated worker can be created:

```
var aDedicatedWorker = new Worker("DedicatedWorker.js");
```

A shared worker is created the same way you create a dedicated worker with the only change being that, rather than specifying a Worker object, we specify a SharedWorker object:

```
var aSharedWorker = new SharedWorker("SharedWorker.js");
```

The worker objects were designed to accept a string indicating the location of the JavaScript file (rather than a function) in order to prevent the creation of a closure that would allow the worker to gain direct access to the browser's DOM or other page elements.

Receiving Messages from a Worker

Communication between workers and the thread that created them is accomplished by passing messages.

Since several HTML5 technologies pass messages, a separate specification exists that is specifically for messages called HTML5 Web Messaging. I mention this because certain details critical to how Web Workers work are not found within the web worker specifications but rather within the web messaging specifications.

In order to receive messages from a worker, you need to attach to the onmessage event handler of the worker instance.

The following is an example of how you can set up an event listener for a dedicated worker:

```
aDedicatedWorker.onmessage = OnWorkerMessage;
```

Setting up the event listener for a shared worker object is similar to the dedicated worker but in this case you need to access the port object as in the following example:

```
aSharedWorker.port.onmessage = OnWorkerMessage;
```

For both dedicated and shared workers, you can also attach to the message event handler event type by using the addEventListener method as in the following example:

```
aSharedWorker.port.addEventListener("message", OnWorkerMessage,
false);
```

One thing to be aware of if you use the addEventListener method is that, for shared workers, you will also need to start the worker:

```
aSharedWorker.port.start();
```

The onmessage event listener method will be passed by a MessageEvent parameter which will have a .data property containing the message passed from the worker thread.

The following is our onmessage event listener (there is no difference between dedicated and shared workers here):

```
function OnWorkerMessage(evt){
   alert("Message received from the worker: " + evt.data);
}
```

Passing Messages to a Worker

To pass a message to a dedicated worker, you call the postMessage function of the worker instance as in the following example:

```
aDedicatedWorker.postMessage("Hello from the UI thread!");
```

Passing a message to a shared worker is similar but in this case the postMessage function is found on the port object:

```
aSharedWorker.port.postMessage("Hello from the UI thread!");
```

The postMessage function is not limited to just passing strings between threads.

You can also pass structured data like JSON objects, JavaScript objects like numbers and Dates, as well as certain data objects like File Blob, FileList and ArrayBuffer.

The following is an example of passing an object:

```
var objData = {
   "employeeId": 103,
   "name": "Sam Smith",
   "dateHired": new Date(2006, 11, 15)
};
aDedicatedWorker.postMessage(objData);
```

Because the objects are passed between the two threads so seamlessly (we pass the object in from one side and receive it on the other side without any special work on our part), it feels like the data is passed directly but by default that is not actually the case. By default, all data is actually copied between the threads (also referred to as structured cloning).

When you pass data to another thread, behind the scenes the object is serialized into a string, passed to the other thread, and then de-serialized back into an object on the other side.

For the most part you probably won't notice too much of a performance hit if you're just passing small amounts of data, but if you need to pass large amounts of data, this cloning may result in noticeable delays.

Fortunately, some browsers now support a concept called transferable objects where the ownership of the data is transferred to the other thread. The data is moved rather than copied, which can result in large performance improvements.

When you transfer an object like an ArrayBuffer to another thread, for example, the contents are transferred to the other thread leaving an empty ArrayBuffer object in the calling thread. To transfer an object to a thread you still use the postMessage method but also include an array in the optional second parameter indicating which items in the message should be transferred.

Hot Tip

At the time of this writing, not all browsers support the second parameter of the postMessage method and will actually throw an exception if you attempt to use the second parameter. It is recommended that you use a try/catch statement around your postMessage call if you wish to use this technique.

If you're passing in just the one object, as in the following example, then you can specify the object itself in the array:

```
var abBuffer = new ArrayBuffer(32);
aDedicatedWorker.postMessage(abBuffer, [abBuffer]);
```

Not all objects are transferable but, if the object you're passing contains a transferable object, you can specify just the portion of the object that you want to be transferred in the transfer parameter array as in the following example:

```
var objData = {
   "employeeId": 103,
   "name": "Sam Smith",
   "dateHired": new Date(2006, 11, 15),
   "abBuffer": new ArrayBuffer(32)
};
aDedicatedWorker.postMessage(objData, [objData.abBuffer]);
```

If the object you are passing to the other thread contains multiple transferable objects, you can specify each item by comma separating them in the transfer parameter array.

Not all browsers support transferable objects. The following example demonstrates a way to detect if a browser supports transferable objects by actually trying to pass a small amount of data in an ArrayBuffer to another thread. If the buffer's length is zero after the postMessage call, then we know that the data was transferred rather than simply copied:

```
var abBuffer = new ArrayBuffer(32);
aDedicatedWorker.postMessage(abBuffer, [abBuffer]);
if (abBuffer.byteLength) {
   alert("Transferrable Objects are not supported by your
browser");
}
```

Error Handling

For both dedicated and shared workers you can attach to the onerror event handler of a worker instance so that you can be informed if a runtime error occurs (e.g., maybe there is a network error preventing the creation of the worker object).

For dedicated workers, the onerror event handler will also tell you about any unhandled exceptions from within the worker thread.

The following is an example of how you would add an event listener for a onerror event on a worker object:

```
aDedicatedWorker.onerror = function(evt) { alert(evt.message); }

aSharedWorker.onerror = function(evt) { alert(evt.message); }
```

ErrorEvent properties	Description
message	A human-readable error message
filename	The name of the script file in which the error occurred (in some browsers this property is undefined and in other browsers the property is an empty string)
lineno	The line number of the script file on which the error occurred (in some browsers this is set to zero)

Closing a Worker

Both a dedicated and shared worker thread can be closed by calling terminate on the worker instance as in the following example:

```
aDedicatedWorker.terminate();

aSharedWorker.terminate();
```

One thing to be aware of with calling terminate is that it does not give the worker thread an opportunity to finish its operations or to clean up after itself.

The following is a summary of the JavaScript that creates a dedicated and shared worker:

```
var gDedicatedWorker = null;
var gSharedWorker = null;

$(document).ready(function () {
   // If Dedicated Workers are supported by this browser...
   if (window.Worker) {
      gDedicatedWorker = new Worker("DedicatedWorker.js");
      gDedicatedWorker.onerror = handleError;
      gDedicatedWorker.onmessage = handleMessage;

      gDedicatedWorker.postMessage("Hello from the UI thread!");
   } // End if (window.Worker)

   // If shared Workers are supported by this browser...
   if (window.SharedWorker) {
      gSharedWorker = new SharedWorker("SharedWorker.js");
      gSharedWorker.onerror = handleError;
      gSharedWorker.port.onmessage = handleMessage;

      gSharedWorker.port.postMessage("Hello from the UI
thread!");
   } // End if (window.SharedWorker)
});

// OnError event listener
function handleError(evtError) {
   if (typeof evtError === "string") {
      updateResults("Error: " + evtError);
   }
                    ------Snippet cont'd on next column--------->
```

```
   }
   else if (evtError.message) {
      updateResults("Error...Message: " + evtError.message +
         ", File name: " + evtError.filename +
         ", Line number: " + evtError.lineno);
      }
   else {
      updateResults("Unknown error");
   }
}

// OnMessage event listener
function handleMessage(evtMessage) {
   updateResults("Message received from the worker: " +
evtMessage.data);
}

// A helper to update a div on the page with a new message
function updateResults(sMessage) {
   var $divResults = $("#divResults");
   $divResults.html(($divResults.html() + "<br />" + sMessage));
}
```

WITHIN A WORKER

Including JavaScript Files

Worker threads have access to an importScripts function which allows the thread to pull in all necessary JavaScript files.

The importScripts function can be used to import one or more files. If multiple files are specified, they are downloaded in parallel but will be loaded and processed synchronously in the order specified.

Also, the importScripts file will not return until all of the files specified have been loaded and processed.

The following is an example of using the importScripts to import 3 JavaScript files into the worker (this example stopped at 3 files but you can specify any number of files):

```
importScripts("file1.js", "file2.js", "file3.js");
```

Receiving Messages

In order for a dedicated worker to receive messages from its creator it must attach to the onmessage event as in the following example:

```
self.onmessage = function(evt) {
   // Data from calling thread can be found in the .data property
   // of the parameter
}
```

Within shared workers, you need to handle the onconnect event which is triggered whenever a thread connects to the shared worker.

Within the onconnect event listener you can then attach to the onmessage event of the first port object in the ports array (there is only ever the one port in the array):

```
self.onconnect = function(evt){
   evt.ports[0].onmessage = function(evtMessage) {
      // Data from calling thread can be found in the .data
      // property of the parameter.

      // The calling thread's port can be found in the .target
      // property of the parameter
   }
}
```

Passing Messages

A dedicated worker is able to pass a message to its creator by means of the postMessage function:

```
self.postMessage("Hello from the dedicated worker");
```

From a shared worker, in order to pass a message to a connected thread, you need to use the postMessage event on the port of the connected thread that you wish to send the message to.

There are a few options available to obtain the port object. One option could be the creation of a global array to hold each connected port from the onconnect event handler.

Another option for obtaining a port object is to access the target property of the onmessage event object which holds a reference to the calling thread's port as in the following example:

```
evt.target.postMessage("Hello from the shared worker");
```

Timeouts

Worker threads support timeouts (setTimeout, clearTimeout, setInterval, and clearInterval), which are very useful if you only want to do processing at set intervals rather than constantly processing.

One scenario could be if you wanted to check a 3rd party server to see if it has new information but you don't want to hammer the site with requests. What you could do is set up a timeout to check to see if there is new information every 5 minutes or so.

XMLHttpRequest

Within a worker thread, it's possible to make an XMLHttpRequest and, since a worker is separate from the UI thread of the browser window and won't affect performance, it's up to you if you want to make the request synchronously or asynchronously.

One thing to be aware of, however, is that within a worker the XMLHttpRequest object will only return data via the responseText property (the responseXML property will always return null when the XMLHttpRequest object is used in a worker).

The following is an example of the use of an XMLHttpRequest object within a worker:

```
// Data we want to send to server-side code
var objData = { "sPageIndex": iPage };

// Make a synchronous call to the server
var xhrRequest = new XMLHttpRequest();
xhrRequest.open("POST", "../main.aspx/GetNextPage", false);
xhrRequest.setRequestHeader("Content-Type", "application/json");
xhrRequest.send(JSON.stringify(objData));

// Return the response to the UI thread
self.postMessage(xhrRequest.responseText);
```

location

Worker threads have access to a location object which indicates the absolute URI that was set at the time of the thread's creation.

The following location properties are available: href, protocol, host, hostname, port, pathname, search, and hash.

navigator

The navigator object is available within a worker giving some details about the browser.

Some of the properties that are available are: appName, appVersion, platform, and userAgent.

Hot Tip This information should not be used in place of feature detection.

Another property available in the navigator object is onLine which will return false if the user agent is definitely offline and will return true if the user agent might be online.
One thing to note about the onLine property of the navigator object is that, at the time of this article's writing, not every browser supports it. As a result, if you choose to use this property, it would be best to check to see if the property exists first.

Error Handling

A worker thread has the option of attaching to a global onerror event handler to listen for unhandled exceptions.

This is especially useful for shared workers since several browsers do not pass unhandled exceptions to the onerror event handler of the worker object itself.

The following is an example of attaching to the onerror event handler within a worker thread:

```
self.onerror = function(evt) {
   // Handle the error possibly by doing a postMessage call to
this
   // thread's creator so that a message can be displayed
}
```

Closing Itself

A worker can close itself by calling the close method:

```
self.close();
```

The following is a summary of the JavaScript within a dedicated worker:

```
// Optionally load in some common JavaScript
// self.importScripts("common.js");

// Attach to the onmessage event to listen for messages from
// the caller
self.onmessage = function (evt) {
   // Pass a message back
   self.postMessage("Text received from the UI: " + evt.data);
}

// Attach to the onerror event to listen for unhandled exceptions
self.onerror = function (evtError) {
   var sMsg = "Unknown Error";

   // Test the error event object to see if it has a message
   // property. If it does not, check to see if it's a string.
   if (evtError.message) { sMsg = evtError.message; }
   else if (typeof evtError === "string") { sMsg = evtError; }

   // Pass the message to this worker's creator
   postMessage("Dedicated Worker - Error Message: " + sMsg);
}

// At some point we can close the thread ourselves
self.close();

The following is a summary of the JavaScript within a shared
worker:
// Optionally load in some common JavaScript
// self.importScripts("common.js");

// Global variable that will hold a reference to the last thread
// that connects to this shared worker. Could also use an array
to
// store each connected thread's port.
var gLastPort = null;

// Attach to the onconnect event to listen for threads that
connect
// to this shared thread
self.onconnect = function (evtConnect) {
   // Grab a reference to the connected thread's port (only
   // ever the 1 port in the array) and attach to the
   // onmessage event
   gLastPort = evtConnect.ports[0];
   gLastPort.onmessage = function (evtMessage) {
      evtMessage.target.postMessage("Text received from the UI: "
+ evtMessage.data);
   }
}

// Attach to the onerror event to listen for unhandled exceptions
self.onerror = function (evtError) {
   var sMsg = "Unknown Error";

   // Test the error event object to see if it has a message
   // property. If it does not, check to see if it's a string.
   if (evtError.message) { sMsg = evtError.message; }
   else if (typeof evtError === "string") { sMsg = evtError; }
                    -----Snippet cont'd on next page---------->
```

```
   // Pass the message to this worker's creator
   gLastPort.postMessage("Shared Worker – Error Message: " +
sMsg);
}

// At some point we can close the thread ourselves
self.close();
```

EXTRAS

Creating Multiple Shared Workers

One of the advantages of shared workers is that you can share the use of a single worker thread across several open windows but you're not limited to a single shared worker.

The SharedWorker object accepts a 2nd parameter in the constructor that specifies the name of the worker.

The 2nd parameter is case sensitive so giving the shared worker a name of Worker in one window and worker in another window will actually result in two shared worker threads being created instead of one that is shared between the two windows because of the difference in the case of the 'W'.

The following is an example of how you can create two shared workers:

```
// Window 1 – Shared worker 1 & 2
var aSharedWorker1 = new SharedWorker("SharedWorker.js",
"Worker1");
var aSharedWorker2 = new SharedWorker("SharedWorker.js",
"Worker2");

// Window 2 – Shared worker 1 & 2
var aSharedWorker1 = new SharedWorker("SharedWorker.js",
"Worker1");
var aSharedWorker2 = new SharedWorker("SharedWorker.js",
"Worker2");
```

Spawning Additional Workers from Within a Worker

If need be, a worker thread can create additional worker objects (sometimes referred to as subworkers) in the same manner that they're created in the UI thread.

Subworkers need to belong to the same origin/domain as the parent page. Also, URIs within subworkers are resolved relative to the parent thread's location rather than the location of the main page.

JSONP within a Worker

Before I go any further with this topic, it's important to understand that there are security concerns with importing JavaScript files/content from a 3rd party site.

When you import data using script injection, the browser will download the JavaScript data received, evaluate it—turning the text received into functions, variables, objects, etc.—and any global code within the payload will be run.

Other than the potential of having arbitrary code run, there is also the possibility of information being leaked to the 3rd party site via the HTTP traffic being exchanged in the process (cookies or other header information).
Traditionally with JSONP you use script injection by modifying the DOM. In the case of Web Workers you don't have access to the DOM so how do we accomplish script injection?

It turns out that it is surprisingly simple to make a JSONP request from within a worker by using the importScripts function as in the following example:

```
function MakeRequest(){
   importScripts("http://SomeServer.com?jsonp=HandleRequest");
}
function HandleRequest(objJSON){
   // do something with the returned JSON object
}
```

Inline Workers

There are times when you might want to take advantage of Web Workers but at the same time you don't want to deal with separate worker files. Inline workers have several advantages including:

- You can make a server request that returns a string containing custom worker code on the fly to meet the current need of your code
- By including the worker code as a string within your JavaScript file, or as part of the markup of the page, you can eliminate a network request which can speed up load time

To accomplish inline workers you populate a Blob object with a string containing the worker code. You then create a URL for the Blob and with that URL you can then create your Worker object. To create a URL for the Blob we use the window.URL.createObjectURL method which gives us a URL similar to the following:

```
blob:http://localhost/72cdddc4-7989-4104-87ef-cf65ed5f5bf9
```

The blob URLs will be unique and last for the lifetime of your application. If you no longer need the URL, you can release it by calling the window.URL.revokeObjectURL passing in the blob's URL.

The following is one example of how an inline worker can be created:

```
var bInlineWorker = new Blob(["onmessage = function(evt) {
postMessage(\"Text received from the UI: \" + evt.data); }"]);
var sBlobURL = window.URL.createObjectURL(bInlineWorker);

var aDedicatedWorker = new Worker(sBlobURL);
aDedicatedWorker.onmessage = handleMessage;
aDedicatedWorker.postMessage("Hello to our inline worker!");
```

The following example is a modification to the above approach, which defines the worker code within the markup of the page allowing for more natural editing of the code as compared to building it as a string directly within a Blob object:

```
<script id="scriptWorker" type="javascript/worker">
   // This block of code is ignored by the JavaScript engines
   // of the browsers because of the type specified
   self.onmessage = function(evt) {
      self.postMessage("Text received from the UI: " + evt.data);
   };
</script>
```

The JavaScript code of the main page would then populate the blob dynamically from the content of the script tag:

```
var bInlineWorker = new Blob([document.
getElementById("scriptWorker").textContent]);
var sBlobURL = window.URL.createObjectURL(bInlineWorker);

var aDedicatedWorker = new Worker(sBlobURL);
aDedicatedWorker.onmessage = handleMessage;
aDedicatedWorker.postMessage("Hello to our inline worker!");
```

ADDITIONAL RESOURCES

Some example code has been written to show a possible real-world use case for dedicated web workers.

The example demonstrates using a dedicated worker thread to pre-fetch data needed by a vertical list of items that the user is able to page through.

Most of code is within the 'main.js' and 'DedicatedWorker.js' files and can be found in the following github repository:

https://github.com/dovicoapi/HTML5WebWorkerExample

The **W3C spec** is located here: http://www.w3.org/TR/workers/

For more information about **HTML5 Web Messaging**, which is the technology used to pass messages to and receive messages from Web Workers: http://www.w3.org/TR/webmessaging/
Check **http://caniuse.com/** for detailed breakdown of current browser support.

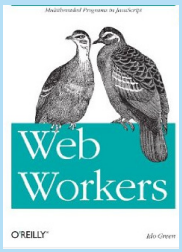
Because both specs and browsers are developing so quickly, it's especially good in this case to know the spec itself pretty well. Tutorials and step-by-step howto guides can become out of date, but the basic principles behind Web Workers are pretty solid. As a general rule, applicable to HTML5 technologies including but not limited to Web Workers, try to cross-reference the spec against any howto guides or code snippets you find -- and be especially conscious of writing good test code.
The web is replete with HTML5 developer sites but, I have found that the following sites stand out and are very helpful in understanding some of the details of the various technologies:

- **HTML5 Zone:** www.DZone.com/mz/html5
- **Mozilla Developer Network:** https://developer.mozilla.org/en-US/docs/HTML/HTML5
- **HTML5 Rocks:** www.html5rocks.com

ABOUT THE AUTHORS

Gerard Gallant is a Senior Software Developer / Software Architect with Dovico Software. For over a decade, he has played a major role in developing most software products released by Dovico ranging from the creation of the Microsoft Project link ActiveX control to the creation of the new Hosted Services API. He is married with two beautiful and smart girls. He unwinds by shooting hoops or watching a movie and also enjoys traveling extensively. Recent publications can be found on his blog: http://cggallant.blogspot.ca/

RECOMMENDED BOOK



Web apps would run much better if heavy calculations could be performed in the background, rather than compete with the user interface. With this book, you'll learn how to use Web Workers to run computationally intensive JavaScript code in a thread parallel to the UI. Yes, multi-threaded programing is complicated, but Web Workers provide a simple API that helps you be productive without the complex algorithms.

Buy Here

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. "**"DZone is a developer's dream",**" says PC Magazine.

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-75-2
ISBN-10: 1-936502-75-5
50795
9 781936 502752

$7.95

Version 1.0