JUMP START SERIES

sitepoint

JUMP START

Html5 APIs

GET UP TO SPEED WITH HTML5 IN A WEEKEND

Summary of Contents

sitepoint®

JUMP START HTML5: APIS

BY **SANDEEP PANDA**

Jump Start HTML5: APIs

by Sandeep Panda

Product Manager: Simon Mackie
English Editor: Kelly Steele
Technical Editor: Craig Buckler
Cover Designer: Alex Walker

Notice of Rights

Notice of Liability

Trademark Notice

sitepoint®

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066
Web: www.sitepoint.com
Email: business@sitepoint.com
Printed and bound in the United States of America

About Sandeep Panda

Sandeep Panda is a web developer and writer with a passion for JavaScript and HTML5. He has over four years' experience programming for the Web. He loves experimenting with new technologies as they emerge and is a continuous learner. While not programming, Sandeep can be found playing games and listening to music.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

About Jump Start

Jump Start books provide you with a rapid and practical introduction to web development languages and technologies. Typically around 150 pages in length, they can be read in a weekend, giving you a solid grounding in the topic and the confidence to experiment on your own.

To my Mom and Dad who taught me to love books. It's not possible to thank you adequately for everything you have done for me. To my grandparents for their strong support. To my brother Preetish for being a constant source of inspiration. And to my awesome friends Ipseeta and Fazle for always believing in me.

Table of Contents

Preface

HTML5 has dramatically changed the way we write web pages. I am sure you might have heard about many of the new elements that were introduced in HTML5. But HTML5 also offers several JavaScript APIs that enhance the interactivity of your pages. This, in turn, enables us to create cutting-edge and powerful web applications just by using HTML5 and its related JavaScript APIs.

In this book, we'll take a quick tour of five of the most useful and powerful new HTML5 APIs. Specifically, we will cover:

- The Web Workers API
- The Geolocation API
- The Server-sent Events API
- The WebSocket API
- The Cross-document Messaging API

This is a short book, so we'll be unable to cover each of these APIs exhaustively; nor will we be building complex real-life applications with each one. However, we will provide code snippets for the APIs, and provide example use cases for each of them.

Who Should Read This Book

This book is for intermediate web developers. You should be familiar with HTML and the fundamentals of JavaScript and the Document Object Model (DOM). It's unnecessary to have a deep knowledge of JavaScript. Still, you should understand event handling, JavaScript data types, and control structures such as `while` loops and `if-else` conditionals. We'll keep our script examples fairly simple, though, and will explain them line by line.

If you're unfamiliar with JavaScript, you may like to read SitePoint's *Simply JavaScript*[1] by Kevin Yank for an introduction.[2] Mozilla Developer Network[3] also offers fantastic learning resources and documentation for both JavaScript and the DOM.

Conventions Used

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

1 http://www.sitepoint.com/store/simply-javascript/
2 http://www.sitepoint.com/store/simply-javascript/
3 https://developer.mozilla.org/en-US/docs/Web/JavaScript

```
function animate() {
  new_variable = "Hello";
}
```

Where existing code is required for context, rather than repeat all the code, a ⋮ will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-design-real-user-
➥testing/?responsive1");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Supplementary Materials

http://www.sitepoint.com/store/jump-start-html5-apis/
The book's website, containing links, updates, resources, and more.

https://github.com/spbooks/jshtml-apis1
The downloadable code archive for this book.

http://www.sitepoint.com/forums/
SitePoint's forums, for help on any tricky web problems.

`books@sitepoint.com`
Our email address, should you need to contact us for support or report a problem, or for any other reason.

Tools You'll Need

If you don't already have a favorite text editor, you'll need one. Try one of those listed below:

- Aptana[4] (requires Java)
- Brackets[5]
- NetBeans[6] (requires Java)
- Notepad++[7] (Windows only)
- Bluefish[8] (Linux only)

They're all free, and in most cases, open source. Aside from Notepad++ and Bluefish, they're all available on Mac, Linux, and Windows as well.

[4] http://aptana.com/
[5] http://brackets.io/
[6] https://netbeans.org/
[7] http://notepad-plus-plus.org/
[8] http://bluefish.openoffice.nl/

Of course, you'll also need a browser that supports the features we'll talk about. Only the latest versions of Google Chrome[9] and Opera[10] support everything we'll cover in this book. Mozilla Firefox[11] and Apple Safari[12] support most of what we'll talk about, as does Microsoft Internet Explorer 10+.[13] We'll note exceptions where necessary, but will pay littles if any attention to Internet Explorer 8 and 9.

Internet Explorer and Safari are bundled with Microsoft Windows and Mac OS X respectively and are only available on those platforms. Other browsers can be downloaded from their particular vendors' websites.

You also need to use web server software. Apache HTTP Server,[14] Nginx,[15] or Lighttpd[16] are all open-source server packages available for Windows, Mac OS X and Linux respectively.

Mac OS X users can also try MAMP,[17] which bundles MySQL, Apache, and PHP into one easy-to-use package. Windows users can try WAMP[18] or XAMPP[19], which are similar packages for that operating system.

Your operating system may also have a web server installed by default. Check its documentation if you're unsure.

[9] http://google.com/chrome
[10] http://www.opera.com/
[11] http://mozilla.org/
[12] htpp://apple.com/safari
[13] http://microsoft.com/ie
[14] http://httpd.apache.org/
[15] http://nginx.org/
[16] http://www.lighttpd.net/
[17] http://mamp.info
[18] http://www.wampserver.com/en/
[19] http://www.apachefriends.org/en/xampp.html

Chapter 1

APIs Overview

In this chapter, we'll make a quick trip to the world of HTML5 APIs. I'll outline the APIs that are going to be discussed in this book, and what you'll have learned by the end. There'll be no diving into any code in this chapter; rather, it will provide a quick overview of each API so that you can get a clear idea about what you're going to learn.

Some HTML5 APIs are still fairly new, and not every version of every browser supports them, which you'll need to bear in mind while creating HTML5 apps. Whenever you're going to use any HTML5 API, it's always a good idea to check the support for that API in the browser. We'll see how to do that in the last section of the chapter.

A Quick Tour of the HTML5 APIs Covered

Since this is a short book, it's impossible to cover each and every API. Some APIs are already covered in other books in SitePoint's Jump Start HTML5 range. In this book we'll focus on five important JavaScript APIs that you can use to create really cool web apps. (Yes, web apps built with plain HTML5 and JavaScript! How cool is that?) So, let's see what we'll be discussing:

- *The Web Workers API*: Ever thought of bringing multi-threading to the Web? Have you every fancied performing some ongoing task in the background without hampering the main JavaScript thread? If yes, it's probably time for you to get cozy with the Web Workers API because it's designed just for this purpose.
  - **Formal definition:** The Web Workers API is used to run scripts in a background thread that run in parallel to the main thread. In other words, you can perform computationally expensive tasks or implement long polling in the background and your main UI thread will remain unaffected.
- *The Geolocation API*: This new API simply lets you know where your users are. It enables you to find the position of your users—even when they are moving. Furthermore, you can show them customized choices (maybe a nice café or a theater near them) depending on their location, or plot their position on the map.
  - **Formal definition:** The Geolocation API lets your application receive the current geographical position of the user through simple JavaScript.
- *The Server-sent Events API*: The way Facebook pushes new updates to your wall is awesome, isn't it? Prior to the introduction of Server-sent Events (SSE, for short) this type of functionality was achieved using long polling. But with the all new SSEs, the server can automatically push new updates to the web page as they become available. You can access those updates in your script and notify your users.
  - **Formal definition:** The Server-sent Events API lets your clients receive push notifications from the server without the need of long polling.
- *The WebSocket API*: This API helps you build applications that allow bi-directional communication between client and server. The classic use case of the WebSocket API is a chat application where a client sends a message to the server and the server processes it and replies back!
  - **Formal definition:** This API enables low-latency, full-duplex single-socket connection between client and server.
- *The Cross-document Messaging API:* Because of dreaded CSRF attacks, documents from different domains are usually not allowed to communicate with each other.

But with this new Cross-document Messaging API, documents from different origins can communicate with each other while still being safe against CSRF.

- **Formal Definition:** This API introduces a messaging system that allows documents to communicate with each other—regardless of the source domain—without CSRF being a problem.

Cross-site Request Forgery (CSRF)

CSRF is a type of attack that tricks end users to perform sensitive operations on a web application without their knowledge. Typically, websites only verify if the request is coming from the browser of an authenticated user, but they don't verify if the *actual* authenticated user himself is making the request. That's the common cause of a CSRF attack.

To learn more about CSRF attacks, please visit the Open Web Application Security Project.[1]

What You Are Going to Learn

This book will provide clear and concise explanations of the APIs just mentioned. You will learn the purpose of each API and how to use them. By the end of the book, you'll be able to create cool and exciting apps using some amazing HTML5 features.

While explaining the APIs, I will provide some code snippets that describe the general working principle of them. Since this is a short book, it's impossible to cover everything exhaustively, but I will try to cover as much as possible in each chapter. I'll also provide guidance and share example use cases for each of the APIs.

Getting Started

Before diving into the world of HTML5, there is a caveat of which to be aware. As mentioned, some HTML5 APIs are quite new so you should always make sure the specific feature is supported in the browsers. If there's no (or limited) support, you should handle it gracefully by falling back to another technique. This is known as ***graceful degradation***; in other words, your app is designed for modern browsers

[1] https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

but it is still functional in older browsers. On the other hand you could also opt for ***progressive enhancement techniques***, where you create an app with limited functionalities and add new features when the browser supports them. For more information on this, have a look at Craig Buckler's article on sitepoint.com.[2]

In this section, we will discuss addressing the browser compatibility issues and setting up the development environment.

Checking Browser Compatibility

It's always good to obtain an overview of the list of HTML5 features supported by the browsers. Before using any API, refer to a compatibility chart that shows which browsers support that API. Personally, I prefer caniuse.com[3] where you just start typing the name of the API and it shows the browser versions that support it; however, although a compatibility chart gives an overview, you should *always* check for browser compatibility of API features in your JavaScript code.

Each HTML element is represented by an object inside the DOM, and each object has several properties. So, in browsers that support HTML5 APIs, certain objects will have a set of unique properties. This is the key to determining the browser's support for various HTML5 features.

Support for some HTML5 features can be detected just by checking the existence of certain properties on the `window` or `navigator` objects. For example, you can write the following code to check the support for Web Workers:

```
if(!!window.Worker){
    //proceed further
}
else{
    //do something else
}
```

As you can see, in Web Worker-enabled browsers there will be a property called `Worker` in the `window` object.

[2] http://www.sitepoint.com/progressive-enhancement-graceful-degradation-choice
[3] http://caniuse.com

Similar techniques can also be used to detect the support for other features. We will see how to detect each feature when we examine the details of each API.

Modernizr

Wouldn't it be great if there was a uniform interface for checking the support for each API? Well, it just so happens there is an open-source JavaScript library that helps detect HTML5 features: Modernizr.[4]

Using Modernizr is a matter of downloading the script and adding the the following code to the `<head>` element:

```
<script type="text/javascript" src="modernizr.min.js">
```

Always put Modernizr in Your `<head>`

Just make sure to load the Modernizr script in the `<head>` section. The reason is because the `HTML5 Shiv`, which enables the styling of HTML5 elements in browsers prior to IE9, must execute before the body loads. Also if you are using Modernizr-specific CSS classes, you might encounter an FOUC (flash of unstyled content)[5] if the script is not loaded in the `<head>`.

Now, let's say you want to detect the support for Web Workers in the browser. The following snippet does that for you:

```
if(Modernizr.webworkers){
    //proceed further
}
else{
    //no support for web workers
}
```

So if the browser supports `webworkers`, the `Modernizr.webworkers` property will be `true`. There are similar tests for all other features. We'll be using Modernizr in this book to check for browser compatibility.

[4] http://modernizr.com/

[5] http://en.wikipedia.org/wiki/Flash_of_unstyled_content

Setting Up the Environment

To create HTML5 apps, you only really need your favorite text editor and browser. But to use certain APIs such as Server-sent Events and WebSocket, you will need a server. So I'll ask you to install WAMP[6] or XAMPP[7] on your machine so that we can easily create a local server to try things out. If you already have a server set up, you're good to go.

In this chapter, we discussed the list of APIs covered in the book and learned how to detect browser support for each HTML5 feature. We also discussed the purpose of each API in brief and finally set up our development environment.

I know you've been waiting to get your hands dirty. Now that you're aware of all the basic stuff, let's start our journey into the world of HTML5 APIs, with Web Workers being the first place to visit.

[6] http://www.wampserver.com/en/
[7] http://www.apachefriends.org/en/xampp.html

Chapter 2

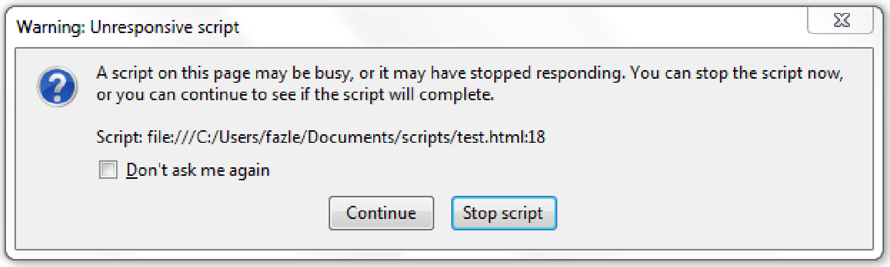
Web Workers

Every HTML5 app is written in JavaScript, but the single and the most crucial limitation of HTML5 apps is that the browsers' JavaScript runtimes are **single-threaded** in nature. Some of you might say that you have run tasks asynchronously in the past using functions like `setTimeout`, `setInterval`, and our all-time favorite, `XMLHttpRequest`. But, in reality, those functions are just asynchronous, not concurrent. Actually, all JavaScript tasks run one after the other and are queued accordingly. Web Workers offer us a **multi-threaded** environment where multiple threads can execute in parallel, offering true concurrency.

In this chapter, we'll first discuss the purpose and how to use Web Workers, before having a look at some its limitations and security considerations.

Introduction and Usage

The Web Worker API allows us to write applications where a computationally expensive script can run in the background without blocking the main UI thread. As a result, unresponsive script dialogs—as shown in Figure 2.1, which is due to the blocking of main thread—can be a thing of the past.


Warning: Unresponsive script

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

Script: file:///C:/Users/fazle/Documents/scripts/test.html:18

Figure 2.1. An unresponsive script message

There are two kinds of Web Workers: dedicated workers and shared workers. The main difference is the visibility. A dedicated worker is accessible from the parent script that created it, but a shared worker can be accessed from **any script** of the same origin. Shared workers have limited browser support: Chrome 4.0+, Safari 5.0+ and Opera 10.6+. Neither Firefox nor IE has support for shared workers.

We'll be discussing dedicated workers in this book, as that's what you'll most probably use.

Detecting Support

Before going any further, I just want to make that sure you detect the browser's support for Web Workers. In the previous chapter, I showed an example where we detected the support for Web Workers using native JavaScript and Modernizr, so we'll skip repeating it here.

To use Web Workers, you just need to call the `Worker` constructor and pass the URI of the Worker script:

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

Worker Script Path

Please note that the Worker script path *must* have the same origin as the parent script, and be relative to the parent script's location.

Here, `myworker.js` is the Worker script that needs to be executed in the background. To communicate with a Worker, you just need to call `postMessage` on the `worker` object, passing a message, if any:

```
// start a worker without any message
worker.postMessage();

// pass a message and start worker
worker.postMessage('Hey, are you in the mood to start work?');
```

But communication doesn't have to be unidirectional. Our Worker can also reply! To receive a message sent by our Worker, we attach an event listener to the `worker` object, like so:

```
//register a callback
worker.addEventListener('message',function(e){
  alert('Got message from worker, '+e.data);
},false);
```

The handler function is also passed a `message` event object. This object has a property called `data` that contains the actual message sent. So in the above code, we accessed the `e.data` property to retrieve the data that was sent by our Worker.

And what happens to the message passed as an argument to `postMessage()` in our main script? Well, that is passed to our Worker script, which can be retrieved at the Worker side by registering the same event listener. The following snippet shows how to do that:

myworker.js

```
//sent from worker
self.addEventListener('message',function(e){
  self.postMessage('Hey, I am doing what you told me to do!');
},false);
```

Workers Are Sandboxed

Workers run in a **sandboxed environment**. This means that they're unable to access everything a normal script can. For example, in the previous code snippet you can't access the global object `window` inside `myworker.js`. So bad things will happen if you try to write `window.addEventListener` instead of `self.addEventListener`. Workers also have no access to the DOM. Why? More on that later.

In case of any error occurring, the `onerror` handler is called. The following callback should be registered in the parent script:

```
//register an onerror callback
worker.addEventListener('error',function(e) {
  console.log(
    'Error occurred at line: '+e.lineno+' in file '+e.filename
  );
},false);
```

Passing JSON data

Web Workers can only be passed a single parameter; however, that parameter can be a complex object containing any number of items. Let’s modify our code to pass JSON data:

parentScript.js

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

worker.addEventListener('message',function(e){
  alert('Got answer: '+e.data.answer+' from: '+e.data.answerer);
},false);

worker.postMessage({'question':'how are you?','askedBy':'Parent'});
```

myworker.js

```
self.addEventListener('message', function(e) {

  console.log(
    'Question: ' + e.data.question +
    ' asked by: '+e.data.askedBy
   );

  self.postMessage(
    {
      'answer': 'Doing pretty good!',
      'answerer':'Worker'
    }
  );

},false);
```

Worker Data Is Copied

The data you pass to the Worker is copied, not shared. The receiver will always receive a copy of data that is sent. It means that just before being sent, the data is serialized and becomes de-serialized on the receiving side. You may wonder why it is implemented this way. Simple! To avoid threading issues.

Web Worker Features

As noted previously, Web Workers run in a sandboxed environment. Their features include:

- read-only access to `navigator` and `location` objects
- functions such as `setTimeout/setInterval` and `XMLHttpRequest` object, just like the main thread
- creating and starting subworkers
- importing other scripts through `importScripts()` function
- ability to take advantage of AppCache

They have no access to:

- the `window`, `parent`, and `document` objects (use `self` or `this` in Workers for global scope)
- the DOM

Why Workers Are Unable to Access the DOM

Browsers (and the DOM) operate on a single thread. That's because your code can prevent other actions, such as a link being clicked. Multiple threads could break that. Also the DOM is not thread-safe. For these reasons, the Worker threads have no access to the DOM—but that won't stop your Workers from modifying main page content. You can always pass a result back to the parent script and let the UI thread update the DOM content. I will show you how at the end of the chapter.

There is just one final issue before we move onto more advanced features: how to close a thread. To terminate a thread, just call `worker.terminate()` from main script or `self.close()` from the worker itself.

More Advanced Workers

Inline Workers

Everybody loves to do things on the fly. Since Web Workers run in a separate context, the `Worker` constructor expects a URI that specifies an external script file to run. But if you want to be really quick, you can create Inline Workers on the fly through `blobs`. Have a look at the following code:

```
var blob = new Blob(["onmessage = function(e) {
➥self.postMessage(e.data); };"]);

var worker = new Worker(window.URL.createObjectURL(blob));

worker.addEventListener('message', function(e){
  alert('Got same Message: '+e.data+' from worker');
},false);

worker.postMessage('Good Morning Worker!!');
```

In this snippet, we wrote the content of our Worker in a `Blob`. `window.URL.createObjectURL` essentially creates a URI (for example, `blob:null/027b645d-be05-4f14-8866-e52604777608`) that references the content of the Worker, and that URI is passed to the `Worker` constructor. Then we proceed as usual.

Since it's inconvenient to put all the Worker content into a `Blob` constructor, we can alternatively put all the Worker code in a separate `script` tag inside the parent HTML. Then, at runtime, we pass that content to the `Blob` constructor.

In the following example, we write the Worker in the parent HTML page itself. Once the Worker starts, it will execute a function every second and return the current time to the main thread. The main thread will then update the `div` with the time (remember we talked about updating the DOM?):

parentPage.html

```
<!DOCTYPE html>
<html>
<head>
<!--
  The following script won't be parsed by the JavaScript engine
  because of its type
-->
<script type="text/javascript-worker" id="jsworker">

  setInterval(function(){
    postMessage(getTime());
  }, 1000);

  function getTime(){
    var d = new Date();
    return d.getHours()+":"+d.getMinutes()+":"+d.getSeconds();
  }

</script>

<script>
  var blobURI = new Blob(
    [document.querySelector("#jsworker").textContent]
  );

  var worker=new Worker(window.URL.createObjectURL(blobURI));

  worker.addEventListener('message',function(e){
    document.getElementById('currTime').textContent=e.data;
  },false);

  worker.postMessage();
</script>
</head>
<body>
<div id="currTime"></div>
</body>
</html>
```

This code is fairly self-explanatory. Once the Worker starts, we register a function that is executed every second and returns the current time. The same data is retrieved and the DOM is updated by the main thread. Furthermore, `#currTime` can be cached for better performance.

If you're creating many blob URLs, it's good practice to release them once you're done (I'd recommend that you avoid creating too many blob URLs):

```
window.URL.revokeObjectURL(blobURI); // release the resource
```

Creating Subworkers Inside Workers

In your Worker files you can further create subworkers and use them. The process is the same. The main benefit is that you can divide your task between many threads. The URIs of the subworkers are resolved relative to their parent script's location, rather than the main HTML document. This is done so that each Worker can manage its dependencies clearly.

Using External Scripts within Workers

Your Workers have access to the `importScripts()` function, which lets them import external scripts easily. The following snippet shows how to do it:

```
// import a single script
importScripts('external.js');

//import 2 script files
importScripts('external1.js','external2.js');
```

URIs are Relative to the Worker

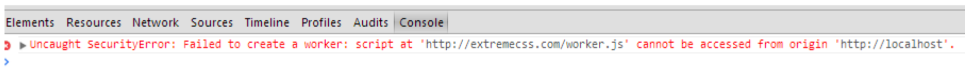
When you import an external script from the Worker, the URI of the script is resolved relative to the Worker file location instead of the main HTML document.

Security Considerations

The Web Worker API follows the **same origin principle**. It means the argument to `Worker()` must be of the same origin as that of the calling page.

For example, if my calling page is at `http://xyz.com/callingPage.html`, the Worker cannot be on `http://somethingelse.com/worker.js`. The Worker is allowed as long as its location starts with `http://xyz.com`. Similarly, an `http` page cannot spawn a Worker whose location starts with `https://`.

Figure 2.2 shows the error thrown by Chrome when trying to go out of the origin:



```
Uncaught SecurityError: Failed to create a worker: script at 'http://extremecss.com/worker.js' cannot be accessed from origin 'http://localhost'.
```

Figure 2.2. Origin error in Chrome

Some browsers may throw security exceptions if you try to access the files locally (via the `file://` protocol). If you are getting any such exceptions, just put your file in the local server and access it with this: `http://localhost/project/somepage.html`.

Polyfills for Older Browsers

What if the browser does not support the Web Workers API? There are several polyfills available to support older browsers by simulating the behavior of Web Workers. The Modernizr page on Github[1] has a list of such polyfills, but long-running code may fail with these implementations. In those cases, it may be necessary to offload some processing to the server via Ajax.

Conclusion

Web Workers give you a big performance boost because of their multi-threaded nature. All modern browsers, including IE10 and above, offer support for Web Workers.

Here are a few use cases that you can try to implement:

- **long polling**[2] in the background and notifying the user about new updates
- pre-fetching and caching content
- performing computationally expensive tasks and long-running loops in the background
- a syntax highlighter tool

[1] https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills#web-workers
[2] http://techoctave.com/c7/posts/60-simple-long-polling-example-with-javascript-and-jquery

- a spell-checker that runs continuously in the background

There are many more uses. I encourage you to be creative and try implementing new projects using what you have learned so far.

The next chapter will be about the Geolocation API.

Chapter 3

The Geolocation API

The Geolocation API provides an easy way to retrieve the exact position of your users. For example, you can create an app that gives personalized suggestions to the users based on their current location. You may also plot their position on the map to show navigation details.

In this chapter, I will give you an overview of the Geolocation API and show how you can use it to create magical location-based HTML5 apps.

Hitting the Surface

Before using the API, let's just make sure the browser supports it:

```
if (navigator.geolocation) {
  // do something awesome
}
else {
  // provide alternative content
}
```

The same check can be achieved through `Modernizr`:

```
if (Modernizr.geolocation) {
  //do something awesome
}
else {
  //provide alternative content
}
```

Use the following code for the actual position of the user:

```
navigator.geolocation.getCurrentPosition
➥(success_callback,error_callback);

 function success_callback(position){
      console.log("Hey, there you are at Longitude:"
➥+position.coords.longitude+"and latitude:"
➥+position.coords.latitude);
 }
 function error_callback(error){
    var msg="";
     switch(error.code){
        case 1:
            msg="Permission denied by user";
            break;
        case 2:
            msg="Position unavailable";
            break;
        case 3:
            msg="Request Timed out";
            break;
    }
    console.log("Oh snap!! Error logged: "+msg);
 }
```

The function `getCurrentPosition()` is asynchronous in nature. It means the function returns immediately, and tries to obtain the location of the user asynchronously. As soon as location information is retrieved, the success callback is executed. A `position` object is also passed to the callback. All the data related to the user'scurrent position is encapsulated in that object.

The following properties are available inside the `position` object:

- `coords.latitude`: latitude of the position

- `coords.longitude`: longitude of the position
- `coords.accuracy`: informs the developer how accurate the location information is (this result is in meters)
- `coords.altitude`: the current altitude in meters
- `coords.altitudeAccuracy`: used to establish the accuracy of the altitude given (previous point)
- `coords.heading`: the direction in which the user is heading
- `coords.speed`: the speed of the end user (device) in meters per second
- `timestamp`: the timestamp indicating when the position is recorded

You may not need all the information contained in the position object. In most cases, all you'll ever need are the first three properties, as those are sufficient to plot a user's position on the map.

The next point to note is that you should handle any error gracefully. When your app tries to retrieve the location, the browser asks the user if the requesting page should be allowed to access the user's location. If the user denies permission, the error callback (see the previous piece of code) is called and an `error` object is passed. The `code` property indicates the type of error that has occurred. There could also be an error because the request timed out.

Now let's see what else you can do with `getCurrentPosition()`. This function takes an optional third parameter, `PositionOptions`, which is a JavaScript object representing additional options. The available properties are:

1. *enableHighAccuracy*: If this is set `true`, it will instruct the hosting device to provide the best possible location details. If the device is a smartphone, it may use GPS to provide highly accurate information.
2. *timeout*: This essentially indicates how many milliseconds you want to wait to obtain a position. For example, if you set the value to 5,000 milliseconds and the app is unable to detect the current position of the user within that interval, it will fail with an error.

3. *maximumAge*: This property indicates if you want the device to cache the last known location. Let's assume the device detects the current location of the user at 5.00 p.m. One minute later (at 5.01 p.m.), your app again calls `getCurrentPosition()` with `maximumAge` of 100,000 (means 100 seconds). Since the device knows where the user was 60 seconds ago and this is less than the `maximumAge` property, it will answer back with that last known location. This is useful if you're satisfied with a previously known location rather than firing a new request for the current position (conserving battery life in many cases). It's like saying "hey device, give me your current position, and I don't mind if that position is *x* milliseconds old." Note that a value of `0` indicates no caching should be done.

The following snippet demonstrates using `PositionOptions`:

```
navigator.geolocation.getCurrentPosition(
  success_callback,
  error_callback,
  {enableHighAccuracy: true, timeout: 35000, maximumAge: 60000}
);
```

Continuously Monitoring Position

If you want to monitor the user's position continuously, `getCurrentPosition()` will fail to work for you; instead, you should use `watchPosition()`. Both functions have the same signatures but they differ in the way they work.

In the case of `watchPosition()`, the success callback is called *every time* the user's position changes. Yes, it's that good! If your app needs to plot the user's location on the map as the user moves, this is the best way to do it. You don't even have to bother about when the position changes. The API takes care of that for you and executes your callback at the appropriate time.

The following code demonstrates the use of `watchPosition()`:

```
var watchId;
function startWatchingPosition() {
  watchId=navigator.geolocation.watchPosition(plotPosition);
}

function plotPosition(position){
  console.log(
     'got position: Latitude='+
     position.coords.latitude+
     ', longitude='+
     position.coords.longitude
   );

   // your code to update the position on map
 }

 function stopWatchingPosition(){
   navigator.geolocation.clearPosition(watchId);
 }
```

Here, the callback `plotPosition()` will be called every time a new location is retrieved. Note that the `watchPosition()` function returns a number that you can store and later use with `clearPosition()` to stop monitoring; it works much like `setInterval()` and `clearInterval()`.

Fast-moving Users

If the user is moving very fast, the callback may execute frequently and slow down the system. Some **event throttling** may be necessary to reduce the number of times our callback runs. There is a small JavaScript library[1] designed by Jonatan Heyman that reduces the number of callbacks we receive from `watchPosition()`.

Accuracy of Geolocation

Geolocation accuracy may be important to your app. As discussed, you can always choose to enable high location accuracy using `PositionOptions.enableHighAccuracy`. But that's just hinting to the device to use a little more power so as to return a more accurate position. The device may silently disregard it. In many situations, the location retrieved may not be accurate enough or even be wrong. And sometimes the user may have no interest in the retrieved location. In those cases, you may want to allow the users to override the location.

[1] https://github.com/heyman/geolocation-throttle

Conclusion

The Geolocation API is a great tool for the developer who wants to build cool location-based applications that give real-time feedback to the users; however, you should remember that the user always has a choice. As mentioned, the user has to explicitly grant permission to your application for you to actually access the location. In those cases, you should be ready with alternative content.

Here are a few small projects you can try to implement on your own:

- Detect your position and plot it on a Google Map.
- Continuously monitor your own position and plot them on the map.
- Detect the position of a user and show them theaters nearby.
- Let your users check in at different places and plot these on a map to later show them the places they visited that day.

Chapter 4

Server Sent Events

Server Sent Events is an API by which browsers receive push notification from servers through the HTTP connection. Once a connection is established, the server can send updates to the client periodically as new data is available. In other words, a client subscribes to the events generated on the server side. Once a particular event occurs, the server sends a notification to the client. With Server Sent Events (SSEs), you can create rich applications that provide real-time updates to the browser without significant HTTP overhead.

In this chapter, we'll discuss the concepts behind SSEs and learn how to use them to build real-time HTML5 apps.

The Motivation for SSEs

Before moving any further, it's important to understand the need for this API. To explain, let me first ask you a question: how would you design a real-time app that continuously updates the browser with new data if there were no Server Sent Events API? Well, you could always follow the traditional approach of long polling through `setInterval()/setTimeout()` and a little bit of Ajax. In this case, a callback executes after a specified time interval and polls the server to check if new data is available.

If data is available. it's loaded asynchronously and the page is updated with new content.

The main problem with this approach is the overhead associated with making multiple HTTP requests. That's where SSEs come to rescue. With SSEs, the server can push updates to the browser as they're made available through a single unidirectional connection. And even if the connection drops, the client can reconnect with the server automatically and continue receiving updates.

Keep in mind that SSEs are best suited for applications that require unidirectional connection. For example, the server may obtain latest stock quotes periodically and send the updates to the browser, but the browser doesn't communicate back to the server, it just consumes the data sent by server. If you want a bi-directional connection, you'll need to use Web Sockets, which are covered in the next chapter.

Okay, enough talking. Let's code!

The API

Here's some example code showing the use of SSEs:

```
if (!!window.EventSource) {
  var eventsource=new EventSource('source.php');
  eventsource.addEventListener('message',function(event) {
    document.getElementById("container").innerHTML = event.data;
  }, false);
}
else{
    // fallback to long polling
}
```

Using Modernizr

Note that we're checking for browser support of SSEs. The same check can be achieved using Modernizr:

```
if (Modernizr.eventsource) {
  // proceed further
}
```

```
else{
  // fallback to long polling
}
```

To use SSEs, you just call the `EventSource` constructor, passing the source URL. The source may be any back-end script that produces new data in real time. Here I have used a PHP script (**source.php**), but any server-side technology that supports SSEs can be used.

You can then attach event listeners to the `eventsource` object. The `message` event is fired whenever the server pushes some data to the browser and the corresponding callback is executed. The callback accepts an `event` object and its `data` property contains our data from the server. Once you have the data, you can perform tasks such as updating a part of the page with new information automatically in real time.

You can be aware of when the connection opens and when an error occurs, as follows:

```
eventsource.addEventListener('open', function(event) {
  // connection to the source opened
},false);

eventsource.addEventListener('error', function(event) {
  // Bummer!! an error occurred
},false);
```

The EventStream Format

There needs to be something in your server's response to help the browser identify the response as a Server Sent Event. When the server sends the data to the client, the `Content-Type` response header has to be set to `text/event-stream`. The content of the server's response should be in the following format:

```
data: the data sent by server \n\n
```

`data:` marks the start of the data. `\n\n` marks the end of the stream.

\n is the Carriage Return Character

You should note that the \n used above is the carriage return character, not simply a *backslash* and an *n*.

While this works for single-line updates, in most cases we'll want our response to be multiline. In that case, the data should be formatted as follows:

```
data: This is the first line \n
data: Now it's the second line \n\n
```

After receiving this stream, client-side `event.data` will have both the lines separated by `\n`. You can remove these carriage returns from the stream as follows:

```
console.log(e.data.split('\n').join(' '));
```

How About a Little JSON?

In most real-world apps, sending a JSON response can be convenient. You can send the response this way:

```
data: {generator: "server", message: "Simple Test Message"}\n\n
```

Now in your JavaScript, you can access the data in the `onmessage` callback quite simply:

```
var parsedData = JSON.parse(event.data);
console.log(
  "Received from " + parsedData.generator +
  " and the message is: " + parsedData.message
);
```

Associating an Event ID

You can associate an event `id` with the data you are sending as follows:

```
id: 100\n
data: Hey, how are you doing?\n\n
```

Associating an event `id` can be beneficial because the browser tracks the last event fired. This `id` becomes a unique identifier for the event. In case of any dropped connection, when the browser reconnects to the server it will include an HTTP header `Last-Event-Id` in the request. On the server side, you can check the presence of the `Last-Event-Id` header. If it's present, you can try to send only those events to the browser that have `event id` greater than the `Last-Event-Id` header value. In this way, the browser can consume the missed events.

Creating Your Own Events

One of the most crucial aspects of SSEs is being able to name your events. In a sports app, you can use a particular event name for sending score updates and another for sending other information related to the game. To do that, you have to register callbacks for each of those events on the client side. Here's an example:

Response from server:

```
event: score \n
data:  Some score!! \n\n
event: other \n
data:  Some other game update\n\n
```

Client-side JavaScript:

```
var eventsource=new EventSource('source.php');

// our custom event
eventsource.addEventListener('score',function(e) {
  // proceed with your logic
  console.log(e.data);
}, false);

//another custom event
eventsource.addEventListener('other',function(e) {
  // proceed in a different way
  console.log(e.data);
}, false);
```

Having different event names allows your JavaScript code to handle each event in a separate way, and keeps your code clean and maintainable.

Handling Reconnection Timeout

The connection between the browser and server may drop any time. If that happens, the browser will automatically try to reconnect to the server by default after roughly five seconds and continue receiving updates; however, you can control this timeout. The following response from the server specifies how many milliseconds the browser should wait before attempting to reconnect to the server in case of disconnection:

```
retry: 15000 \n
data: The usual data \n\n
```

The `retry` value is in milliseconds and should be specified once when the first event is fired. You can set it to a larger value if your app does not produce new content rapidly and it's okay if the browser reconnects after a longer interval. The benefit is that it may reduce the overhead of unnecessary HTTP requests.

Closing a Connection

We always reach this part in almost every API; when resources are no longer needed and it's better to release them. So, how do you close an event stream? Write the following and you'll no longer receive updates from the source:

```
eventsource.close(); //closes the event stream
```

A Sample Event Source

To finish with, how about some Chuck Norris jokes? Let's create a simple `EventSource` that will randomly fetch a joke and push it to our HTML page. Here's a simple PHP script that pushes the new data:

source.php

```
<?php
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache');
ob_implicit_flush(true);
ob_end_flush();
while (true) {
  sleep(2);
  $curl=curl_init('http://api.icndb.com/jokes/random');
  curl_setopt($curl, CURLOPT_RETURNTRANSFER, true);
  $result=curl_exec($curl);
  echo "data:$result\n\n";
}
?>
```

Some Notes on the PHP Script

The data from the API is in JSON format. SSEs require you to prepend your data with `data:` and mark the end with `\n\n`. If you miss these points, the `EventSource` won't work, and remember to also pay attention to the headers. The reason for the loop is to keep the connection open. Without the loop, the browser will attempt to open a new connection after five seconds or so.

As we are looping continuously in the PHP script, the execution time of the script may exceed PHP's `max_execution_time`. You may also face problems because of the Apache user connection limit. Technologies such as Node.js may be a better choice for these types of real-time apps.

Here's the HTML page that displays the data:

jokes.html

```
<!DOCTYPE html>
<html>
<head>
<title>A Random Jokes Website</title>
<script>
if(typeof(EventSource)!=="undefined"){
  var eventsource=new EventSource('source.php');
  eventsource.addEventListener('message', function(event) {
    document.getElementById("container").innerHTML =
    ➥JSON.parse(event.data).value.joke + '<br/>' +
    ➥document.getElementById("container").innerHTML;
  },false);
}
else console.log("No EventSource");
</script>
</head>
<body>

  <div id="container"></div>
```

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

Debugging

In case you encounter problems, you can debug your application using the browser's JavaScript console. In Chrome, the console is opened from **Menu** > **Tools** > **JavaScript Console**. In Firefox, the same can be accessed from **Menu** > **Web Developer** > **Web Console**. In case the event stream is not working as expected, the console may report the errors, if any.

You need to pay special attentions to the following:

- sending the header `Content-Type: text/event-stream` from the server
- marking the start and end of content with `data:` and `\n\n` respectively
- paying attention to same-origin policies

Conclusion

Server Sent Events have solved the long polling hack that we previously used to achieve real-time update functionality. Here are a few simple projects that you might want to implement:

- Creating a page that has a clock updated from the server side.
- Reading and displaying the latest tweets with the help of the Twitter API.
- Fetching a random photo with the Flickr API and updating the page.

The Mozilla Developer Network has a good resource for learning more about SSEs.[1] Here, you can also find some demo apps and polyfills for older browsers.

[1] https://developer.mozilla.org/en-US/docs/Server-sent_events

Chapter 5

The WebSocket API

I think that WebSockets are one of the coolest APIs available in HTML5. The real strength of the WebSocket API comes to the fore when you want your clients to talk to the server through a **persistent connection**. This means that once the connection is opened, the server and the client can send messages to each other anytime. Gone are the days when clients used to send a message to the server and wait until the server responded back. Clearly, WebSockets eliminate the need for long polling! The API is very useful if you want to create real-time apps where clients and the server talk to each other continuously. For instance, you can build chat systems, multi-player HTML5 games and similar apps with WebSockets.

WebSockets versus Server Sent Events (SSEs)

WebSockets and SSEs can achieve similar tasks but they are different. In the case of SSEs, the server only pushes data to the client once new updates are available. In the case of WebSockets, both client and server send data to each other; to be precise, the communication is bi-directional (full-duplex, if you love more technical terms).

To use WebSockets, you'll need a WebSocket-enabled server. Well, that's the tricky part. Don't worry! There are implementations of WebSockets available for several different languages. I will cover those shortly.

Additionally, you should always bear in mind that WebSockets allows *cross-origin communication*. So, you should always only connect to the clients and servers that you trust.

The JavaScript API

Let's quickly try it out. The developers at WebSocket.org[1] have created a demo WebSocket server at `ws://echo.websocket.org`. We can connect to it and start exchanging messages with it in no time.

The `ws://` Protocol

`ws://` is a protocol that's similar to `http://`, except that it's used for specifying the Web Sockets server URLs.

First, let's see how to connect to a simple WebSocket server using the JavaScript API. After that, I will show how you can create your own WebSocket server and let others connect with you.

```
// test if the browser supports the API
if('WebSocket' in window) {

  var socket = new WebSocket('ws://echo.websocket.org');

  if (socket.readyState == 0) console.log('Connecting...');

  // As soon as connection is opened, send a message to the server
  socket.onopen = function () {
    if (socket.readyState == 1) {
      console.log('Connection Opened');
      socket.send('Hey, send back whatever I throw at you!');
    }
  };

  // Receive messages from the server
```

[1] http://www.websocket.org/

```
  socket.onmessage = function(e) {
    console.log('Socket server said: ' + e.data);
  };

  socket.onclose = function() {
    if (socket.readyState == 2) console.log('Connection Closed');
  };

  // log errors
  socket.onerror = function(err) {
    console.log('An Error Occurred ' + err);
  };

}
else {
  // sadly, no WebSockets!
}
```

Using Modernizr

With Modernizr, we can check for the browser support of WebSockets this way:

```
if (Modernizr.websockets) {
  // proceed
}
else{
  // No WebSocket
}
```

So, you start by passing the socket server URL to the `WebSocket` constructor. The constructor also accepts an optional second parameter, which is an array of sub-protocols. This parameter defaults to an empty string.

Next, we attach different callbacks for different events. As soon as the connection opens, the `onopen` event is fired and our callback executes. You can send a simple message to the server by calling `socket.send()`. You can also send binary data, which we'll see in the next section.

Similarly, the server can also send us messages. In that case, the `onmessage` callback fires. At the moment, the server sends us back whatever we send to it, and we simply log the message received. But you can always capture the message and dynamically update your page with it.

Just paste the code in an HTML file and run it — you'll be delighted to see the server's response!

The `readyState` Property

The variable `socket` in the aforementioned code has a property called `readyState` indicating the status of the connection:

- 0 = connecting
- 1 = opened
- 2 = closed

Sending Binary Data

You can also send binary data to the server. The following program sends the image drawn on `canvas` to a sample WebSocket:

```
// you need to create this socket server
var connection=new WebSocket('ws://localhost:8080');

connection.onopen = function() {

  // get an image from canvas
  var image = canvas2DContext.getImageData(0, 0, 440, 300);
  var binary_data = new Uint8Array(image.data.length);
  for (var i = 0; i < image.data.length; i++) {
    binary_data[i] = image.data[i];
  }
  connection.send(binary_data.buffer); // send the data

}
```

In the code, we read the image from the HTML page and create an `ArrayBuffer` to contain the binary data. Finally, `connection.send()` actually sends the data.

Using Blobs

We can also send the binary data as a blob. For example, you could create a file uploader and read the files through `querySelector()`. Then you can send those files with the help of `connection.send()`. HTML5Rocks has an excellent tutorial on WebSockets[2] that also covers sending binary data to the server through blobs.

Sample Server Implementations

Here are a few WebSockets implementations available for different server-side languages. You can choose a library based on your preferred language:

- PHP: Ratchet[3]
- Node.js: Socket.IO[4]
- Java: jWebSocket[5]

For a complete overview of server-side libraries, Andrea Faulds maintains a comprehensive list.[6]

It's beyond the scope of this short book to discuss each of these libraries in detail. But regardless of the implementation, they all offer a simple API through which you can interact with your clients. For example, they all offer a handler function to receive messages from the browser and you can also communicate back with the client. I encourage you to grab a library for your favorite language and play around with it.

I have written an extensive tutorial on WebSockets on SitePoint.com.[7] In that tutorial, I've shown how to implement a WebSockets-enabled server using jWebSocket and let others connect to it.

[2] http://www.html5rocks.com/en/tutorials/websockets/basics/
[3] http://socketo.me/
[4] https://github.com/learnboost/socket.io
[5] http://jwebsocket.org/
[6] http://ajf.me/websocket/#libs
[7] http://www.sitepoint.com/introduction-to-the-html5-websockets-api/

Conclusion

If you want to learn more about the WebSocket API, the following resources are worth checking out:

- the WebSocket API[8]
- WebSocket tutorial at Mozilla Ddeveloper Network[9]
- WebSocket demo apps[10]

Here are a few use cases of the API:

- creating a simple online chat application
- updating a page as new updates are available on the server and communicating back
- creating an HTML5 game with multiple users.

[8] http://dev.w3.org/html5/websockets/
[9] https://developer.mozilla.org/en-US/docs/WebSockets
[10] http://www.websocket.org/demos.html

Chapter 6

The Cross-document Messaging API

The **Cross-document Messaging API** in HTML5 makes it possible for two documents to interact with each other without directly exposing the DOM. Just imagine the following scenario: Your web page has an iframe that is hosted by a different website. If you try to read some data from that iframe, the browser will be very upset and may throw a security exception. It prevents the DOM from being manipulated by a third-party document, thereby stopping potential attacks such as CSRF[1] or cross-site scripting (XSS).[2] But the Cross-document Messaging API never directly exposes the DOM. Instead, it lets HTML pages send messages to other documents through a `message` event.

The Cross-document Messaging API is useful for creating widgets and allowing them to communicate with third-party websites. For example, let's say that you have a page that serves ads and you allow the end-users to embed this page in their websites. In this case, you can let users personalize the ads or modify the type of ads through the Cross-document Messaging API. Clients can send messages to your page and you can receive those messages too.

[1] http://en.wikipedia.org/wiki/Cross-site_request_forgery
[2] http://en.wikipedia.org/wiki/Cross-site_scripting

The JavaScript API

The Cross-document Messaging API revolves around a single method: `window.postMessage()`. As its name suggests, this method allows you to post messages to a different page. When the method is called, a `message` event is fired on the receiving document side. Before moving further, it's crucial to understand the properties of the `message` event. There are three properties we're interested in:

1. *data:* This holds the message being sent. You have already played with it in previous chapters (remember calling `event.data` in SSEs?).

2. *origin:* This property indicates the sender document's origin; i.e the protocol (scheme) along with the domain name and port, something like `http://example.com:80`. Whenever you receive a message, you should **always, always check** that the message is coming from a trusted origin. I will explain how to do that in the next section.

3. *source:* This is a reference to the sender's window. After receiving a cross-document message, if you want to send something back to the sender, this is the property you'll use.

Basic Usage

You send a message to another document by calling `window.postMessage()`. This function takes two arguments:

- *`message:`* the actual message you want to send

- *`targetOrigin:`* a simple string indicating the target origin—an additional security feature (I'll explain how this is useful in the next section)

The code looks like the following:

```
targetWindow.postMessage(message, targetOrigin);
```

You should note that `targetWindow` refers to the `window` to which you want to send a message. It may be a window you just opened through a call to `window.open()`, or it can also be the `contentWindow` property of some iframe.

A Reference to an Open Window

`window.open('a URL')` returns a reference to the opened window. You can always call `postMessage()` on it.

Let's build a very simple example. Say that we have a parent page that has an iframe inside it. The iframe's `src` points to a third-party website that provides us with a random image.

This is the page referenced by the iframe in our parent page:

child.html

```
.f<!DOCTYPE html>
<html>
<head>
<title>A page that provides a random image</title>
</head>
<body>
  <div id="container">
    <img
      ➥src="http://randomimage.setgetgo.com/get.php?key=2323232"
      ➥id="image"
    />
  <div>
</body>
</html>
```

So far, so good! Now let's have a look at our parent page:

parent.html

```
<!DOCTYPE html>
<html>
<head>
<title>The Parent Document</title>
</head>
  <body>
    <iframe
      ➥src="child.html"
      ➥height="500" width="500" id="iframe">
    </iframe>
    <br/>
  </body>
</html>
```

When you open up the parent page, you can see the image coming from the page **child.html**. For now, both **parent.html** and **child.html** are on the same server (`localhost`) for testing purposes. But they should ideally be on different servers.

But we don't want to keep showing a static image to our users, nor reload our iframe. It would be really great if we could ask the page **child.html** to reload its image when a user hits a button on our parent page; i.e. **parent.html**.

Let's start by adding a button to our parent page. We'll also write a function that responds to the click event and sends a message to **child.html**.

parent.html

```
<!DOCTYPE html>
<html>
<head>
<title>The Parent Document</title>
</head>
  <body>
    <iframe
      ➥src="child.html"
      ➥height="500" width="500" id="iframe">
    </iframe>
    <br/>
    <button id="reloadbtn">Reload</button>

    <script>
      document.getElementById("reloadbtn").
      ➥addEventListener("click", reload, false);

      // reload handler
      function reload(e) {
        // is cross-messaging supported?
        if (window.postMessage) {
            document.getElementById('iframe').
            ➥contentWindow.postMessage(
              Math.random()*1000, 'http://localhost'
            );
        }
```

```
          else {
              console.log('postMessage() not supported');
          }
        }
    </script>

  </body>
</html>
```

Using Modernizr

If you're using `Modernizr` to check browser compatibility, check for the property `Modernizr.postmessage`.

As you can see, when a user clicks on `reload` button our callback `reload()` executes. First, we ensure that `postMessage()` is supported by the browser. Next, we call `postMessage()` on the iframe's `contentWindow`. The `contentWindow` property of an iframe is simply a reference to that iframe's window. Here, our message is a simple random number (we will see why shortly). The second argument to `postMessage()` is `http://localhost`. This represents the `targetOrigin` to which the message can be sent. The origin of the iframe's `src` and this argument must be same in order for `postMessage()` to succeed. This is done so that other unintended domains cannot capture the messages. In this case, if you pass something else as the `targetOrigin`, `postMessage()` will fail.

targetOrigin

Think of `targetOrigin` as a way of telling the browser to which origin the message can be sent. You can also pass `"*"` as the `targetOrigin`. As you might have guessed, * is a wildcard that says the message can be sent to documents from any origin. But using a wildcard means loosening your security system by allowing the message to be sent to any origin. I recommend passing the exact origin as the second argument to `postMessage()` instead of the wildcard.

Now we have to receive the message in **child.html** and take appropriate action. Here's the modified **child.html** this:

child.html

```
<!DOCTYPE html>
<html>
<head>
<title>A page that provides random image</title>
</head>
<body>

  <div id="container">
    <img
      ➥src="http://randomimage.setgetgo.com/get.php?key=2323232"
      ➥id="image"
    />
  <div>

  <script>
    window.addEventListener('message', messageReceiver, false);

    function messageReceiver(event) {

      // can the origin can be trusted?
      if (event.origin != 'http://localhost') return;

      document.getElementById('image').src =
      ➥"http://randomimage.setgetgo.com/get.php?key=" + event.data;

      console.log(
        'source=' + event.source +
        ', data=' + event.data +
        ', origin=' + event.origin
      );

    }
  </script>

</body>
</html>
```

First, we attach a callback to the `message` event. Whenever **parent.html** sends a message to **child.html**, this callback will be executed. The first and most important step is to check whether you are receiving messages from the **intended origin**. After adding an event listener to the `message` event, you can receive messages from documents of any origin. So, it's recommended to always put a check inside your callback to ensure that the message is coming from a trusted origin.

Next, we retrieve the message from `event.data`. This particular API that we're using for random images requires a different random number each time so that the generated image will be a unique one. That's why we're generating a random number on a button click (in **parent.html**) and passing that as a message to **child.html**. In **child.html**, we simply construct a new image URL with the help of the random number and update the image's `src`. As a result, we can see a new image each time we click the reload button from the main page.

Sending a Message Back

If you want to send a message back to **parent.html**, you can always use `event.source.postMessage()` inside your event listener in **child.html**. Consequently, you'll also need an event handler in the parent page.

Detecting the Readiness of the Document

Most of the time, you'll send messages to iframes embedded in your pages. But many times, you may also need to open a new window from your page and post messages to that. In this case, ensure that the opened window has fully loaded. Inside the opened window, attach a callback to the `DOMContentLoaded` event, and in that function send a message to the parent window indicating that the current window has fully loaded.

Getting a Reference

Inside the `DOMContentLoaded` event listener (in the opened window), you can get a reference to the window by accessing `event.currentTarget.opener` and calling `postMessage()` on it as usual. Tiffany Brown explains how to achieve this in an excellent tutorial.[3]

[3] http://dev.opera.com/articles/view/window-postmessage-messagechannel/#whenisdocready

Conclusion

This was the overview of the Cross-document Messaging API. By using this API, two cross-origin documents can securely exchange data. Because the DOM is not directly exposed, it’s now possible for a page to directly manipulate a third-party document.

The Cross-document Messaging API certainly gives you more power. But, as you know, with great power comes great responsibility! If you fail to use this API properly, you may end up exposing your website to various security risks. So, as discussed in this chapter, you should be very, very careful while receiving cross-document messages to avoid security risks. Similarly, while sending messages with `window.postMessage()`, don’t use `*` as `targetOrigin`. Instead, provide a single valid origin name.

Although it’s not possible to cover each and everything about the API in detail, this chapter gives you a head start. You should now be able to experiment with different things on your own. For further reading, I strongly recommend the following resources:

- the Mozilla Developer Network[4]
- the W3C specification.[5]

So, this brings us to the end of our tour through five of the most important and useful HTML5 APIs. I hope you've found this book a useful introduction to these powerful technologies. Do take a look at the sample project ideas that I have shared in the end of each chapter; I also encourage you to get creative and think of some other good use cases that can be implemented.

Happy coding!

[4] https://developer.mozilla.org/en-US/docs/Web/API/Window.postMessage
[5] http://www.w3.org/TR/webmessaging/