Offline First

HTML5 technologies for a faster, smarter, more engaging web

draft

1

intro-1

Offline Apps with HTML5

What if I told you that you don't have to be online to use the Web?

There's a general (and understandable) belief by many developers, not to mention most users, that websites and web applications have a very serious limitation – they can only be used when the browser has a web connection. Indeed, this is routinely cited as one of the real advantages of so-called native apps over the Web.

As counter-intuitive as it sounds, in almost every modern browser on any device (including IE10 and up), it's no longer the case that users need to be connected to the Web to use our websites and applications, provided we developers do a little extra work to make our site or application persist when a browser is offline.

This opens up a whole range of opportunities, leveling the field with native apps that can be installed on the user's phone, tablet, laptop or desktop computer, or indeed any other device capable of running apps. But there are many more benefits to offline technologies than simply allowing websites and apps to work offline, as we'll soon discover.

Offline First

Recently the team behind the "Hoodie" framework published "Offline first"[1] a new way of thinking about developing for the Web. They observe that even in the developed world, mobile bandwidth and connectivity, increasingly the primary way in which people connect to the Web, are not always guaranteed to be there, or be reliable, and, they state:

"We can't keep building apps with the desktop mindset of permanent, fast connectivity, where a temporary disconnection or slow service is regarded as a problem and communicated as an error."

And when we think of offline and online, we typically only focus on the client, but servers go offline as well, both for routine maintenance, or in times of crisis, or under heavy stress. What if your user could continue to use all, or the core of your site's functionality even when your site is offline? In the last chapter we'll look at a case study where this is one of the principle benefits of the application working offline.

There's more to changing this mindset that the Web only works when the user (and server) is offline than simply adopting technology. There are a range of interaction design concerns as well. We'll touch on some of these in this book, but above all, we'll focus on the technologies that now enable us to build offline first web experiences.

What we'll cover

This book covers the technologies and practices you'll need to make your apps work as well offline, as they do online. In addition to the HTML5 Application Cache, which allows our apps to work offline, covered in Chapter 2, we'll also cover:

- WebStorage, a simple, in-browser database that replaces cookies and other hacks for keeping data between sessions in the browser, in Chapter 3

- Offline events and ways we can try to determine browser connectivity, in Chapter 4
- The HTML5 File API, that lets us read files from the local file system, in Chapter 5
- A case study demonstrating the real world benefits to a very successful Web service that is fully functional offline in Chapter 6
- Finally, our two appendices bring together all the browser support information for each of the features covered in the book, as well as a detailed cheat sheet for the features we cover.

We'll begin in Chapter 1 by looking at HTTP Caching which has been around for many years, but which still has relevance, indeed, arguably increasing relevance as we rely more and more on Content Distribution Networks (**CDN**s) to deliver our web content more quickly.

But I just build websites, why should I care?

Now, you might be thinking: "I don't build apps, just plain old websites. Why do I need to worry about offline?"

There's an additional sweetener associated with these offline web technologies. By reducing the amount of downloading (and uploading) our sites and apps need to do, this helps them load much more quickly. Indeed, that's often what they're most used for, even by services like Google Search and Bing. In addition to helping improve performance, these technologies can also:

- improve the security of the sites we build, by minimizing the need for a number of traditional security weak points, such as HTTP Cookies, and file uploading more generally,
- reduce the amount of work we need as developers to do compared with more traditional approaches to caching,
- as well as opening up new possibilities for improving the experience of our users.

In short, offline web technologies are fantastic for developers.

Ready? Let's begin!

Convinced it's worth a little effort to learn about HTML5's offline capabilities? Great! Let's begin by taking a look at how we have been able to manage caching with HTTP, in many ways a precursor to HTML5 Application Cache.

2

chapter 1-1

Caching, (nearly) as old as the Web itself

In this chapter we'll look at HTTP Caching and how browsers have cached resources to improve the performance of websites since the earliest days of the Web.

We'll cover:
- **what HTTP Caching is, and how it works**
- **some of the most common cache related HTTP headers**
- **why we might still use HTTP Caching**

As much as we worry about the performance of JavaScript engines, CSS Selectors and I/O bound operations (such as reading and writing files), the key limiting factor to the performance of the Web has *always* been the network. The combination of limited bandwidth (how much information can travel between the server and the client in any given time) and high latency (the time it takes for requests to travel between client and server) makes the impact of network performance orders of magnitude greater than the sorts of things we often obsess about as

developers when it comes to performance. While network performance has increased dramatically in the last two decades, it is still, and will always be our single largest performance bottleneck (if only because the speed of light provides a fixed performance limit for communication over a network).

Browser developers, always striving to make their browser feel faster, have long used all kinds of techniques to minimize network use. Principally, browsers have, for many years, cached resources on the user's local file system, so that when the resource is next required, the locally stored version will be used. The challenge here is for the browser to determine when it makes sense to use the cached resource, and when it has become stale and needs refreshing.

Enter HTTP Caching

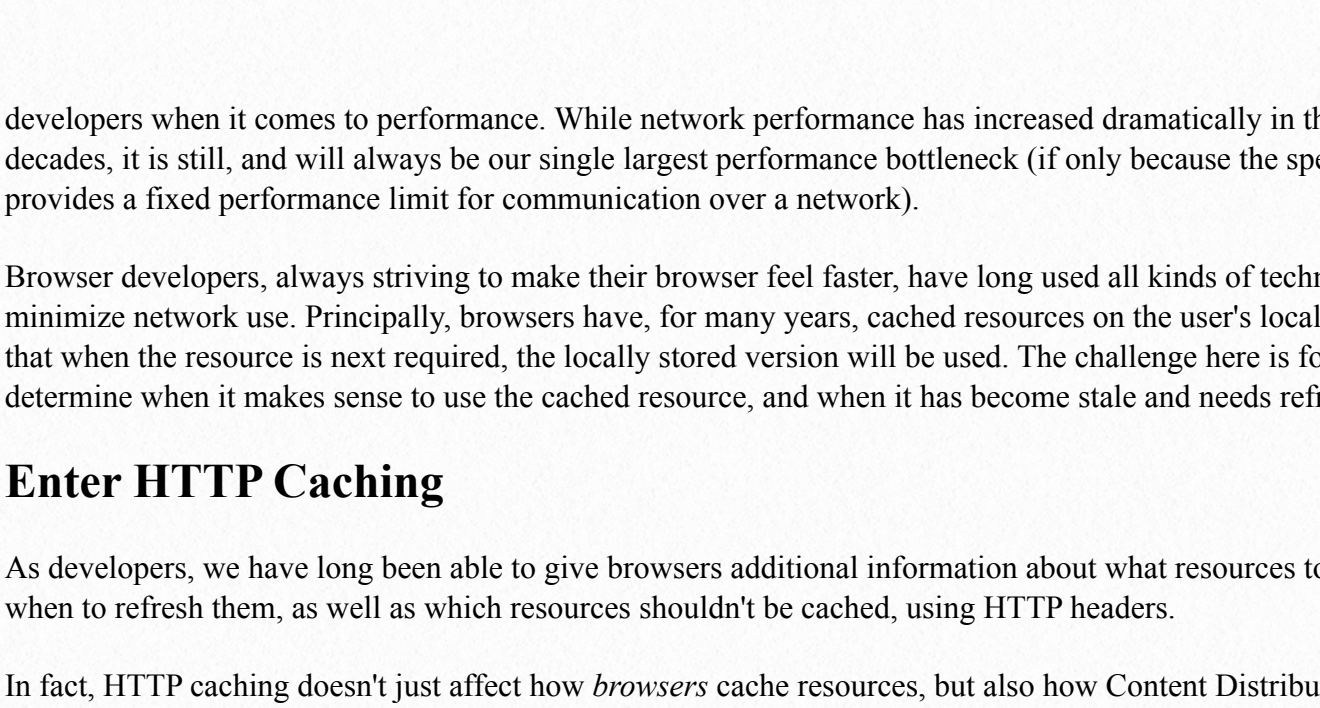
As developers, we have long been able to give browsers additional information about what resources to cache and when to refresh them, as well as which resources shouldn't be cached, using HTTP headers.

In fact, HTTP caching doesn't just affect how *browsers* cache resources, but also how Content Distribution Networks (CDNs) and Proxy Servers cache as well. In the next chapter, we'll be looking in detail at HTML5's Application Cache, which means we have far less need for HTTP caching. But, because in many ways proxies and CDNs are becoming increasingly important, as we serve more and more (as well as larger and larger) files over the Web, HTTP caching still has its place.

Caching is always a trade-off, between having the most up-to-date resources, and making as few requests as possible to the server, whether to verify that we have the freshest resources or to download them. Let's see how we can achieve this balance, using HTTP Headers.

HTTP Headers

When your browser requests content from a server and the server returns a resource, it's not just the content which passes between the client and server. The request for a resource is an HTTP header, and the response contains a header, with information about the content, along with the content itself. You're most likely familiar with the response codes an HTTP server returns about a resource (most famously, 404) and content type (for example `text/html`) but there's a



Browser developer tools let us inspect the HTTP headers for any request or response

whole lot more information about the server, the encoding of the associated content (for example is it compressed? What type of compression?), and among this, information that the browser, a proxy cache or a CDN can use to make decisions about the best trade-off for caching the resource.

Inspecting HTTP headers in the browser

The developer tools in modern browsers make it easy to see the headers for a resource. For example, in Google Chrome Developer Tools (Network tab, click a resource, then the Headers section), we can see both the request and response headers for any resource. Firefox, Safari, Internet Explorer and Opera all have similar tools.

The header fields we're interested in are in the response. These include `Cache-Control` `Expires`, `ETag` and `Last-Modified`. Let's look at the most important ones in turn.

`Cache-Control`

The `Cache-Control` header contains directives which browsers must follow about the cacheability of a resource. These directives include:

- `public`: the resource *may* be cached by the browser and any other cache (proxies and CDNs). We can't **force** a browser or proxy to cache a resource this way, only give them permission to do so.
- `private`: this resource must **not** be cached by a shared cache, such as a proxy. Browsers may cache these resources.
- `no-cache`: this directive is rather confusingly named. In essence, the browser (and other caches) *may* cache the resource, but must not use a cached version without checking with the server that the resource is up to date. This process of checking whether a resource is up to date is called *validating* the cache.

In addition, `Cache-Control` may also direct that a browser or proxy not cache the resource at all, with the `no-store` directive. This is useful for ensuring sensitive data isn't cached.

How do these headers impact performance? By giving the browser hints as to what is cacheable (resources served with `public` and `private` directives), what we always want validated before using from the cache (`no-cache`), and what we never want cached (`no-store`), the browser can make smaller requests for resources, but also ensure that frequently updated resources are refreshed as required.

It's also worth noting that for some status codes the HTTP specification allows browsers and proxies to cache resources by default, unless these are explicitly marked as not to be cached.

Expiring caches

In addition to the `Cache-Control` directives specifying whether and how resources may be cached, HTTP headers may include information about the expiration of a resource. This means we can help a browser or proxies to more intelligently check whether a resource needs refreshing after a certain date or period of time.

`Expires`: this header contains a date that stipulates when the resource will require refreshing from the server. The browser may cache a resource until the expiry date, and use it **without validating the resource with the server**. So, we need to be careful when specifying an expiration date. Once a browser or proxy has cached a resource with an `Expires` header, it **stays cached until the expiry date** (or the cache has no more space, in which case even an unexpired resource may be discarded and need refreshing). Once cached, to override this we need to give the resource a different URL. A common technique for this is to add version numbers to file names. For example, if we have a JavaScript file called script.js, we might append a version number, such as `script1.1.js`.

The `Cache-Control` header may also include additional information about the freshness of a resource, which browsers and proxies can use to more intelligently cache resources. These include:

`max-age`: a value in seconds that specifies how long the resource should be cached and reused before revalidating. It's similar to the `Expires` header, though where a date is an absolute value, this value is relative to when the resource was requested. `max-age` overrides `Expires` where both are set. The maximum recommended value for `max-age` is 31536000, which is a year.

Again, take note that as with the `Expires` header, once a browser has cached a resource with `max-age` it will not validate, that is check whether the resource needs refreshing until this period of time has passed (unless as we saw the

cache is full, and the resource is deleted). Once more the only way to override this is to change the URL of the resource.

`Last-modified`: This header tells the browser when a resource was last changed on the server. When a browser has cached a resource served with a `Last-modified` header, the next time it needs this resource, instead of initially requesting the resource, it can ask the server when the resource was last modified, and only then download the resource again, if that date is more recent.

`ETag`: this is a unique identifier for a version of a resource. If the file associated with a particular URL changes, then a new unique `ETag` is generated. The `ETag` value is used just like the `Last-modified` header. If the `ETag` of a resource has changed since the resource was last cached, the resource will be refreshed. A downside of using ETags[1] is that they are typically uinque to a particular server, and by default, 'Apache and IIS embed data in the ETag that dramatically reduces the odds of the validity test succeeding on web sites with multiple servers'

Most servers will generate `Last-modified` and `ETag` headers automatically.

There's quite a bit more we could go into about HTTP caching, but as mentioned, with HTML5 Application Cache, HTTP caching is more of relevance to proxy caching and CDNs than it is to browser caching. As we'll see shortly, HTML5 Application Caches overrides HTTP caching almost completely. But HTTP caching still, as we've mentioned, has relevance for proxies and CDNs.

Why would we use them?

While HTML5 Application Cache (mostly) overrides HTTP caching, HTTP caching is still relevant for both:

- older browsers (including Internet Explorer 9 and lower) which don't support Application Cache
- CDNs and proxies which don't support Application Cache.

Performance

As we saw, we can improve page load performance by reducing the amount of downloading, as well as reducing the number of requests to the server, not just for resources, but also requests to check whether a resource needs refreshing at all. So, to improve the performance of our page load, we'll try to limit both the initial requests for resources, as well as the downloading of resources. To understand how we can do this, let's look at the lifecycle of an HTTP request. To simplify, we'll leave CDNs and proxies out of the equation, but the process is very similar.

HTTP Request Lifecycle

The first time a browser requests a particular resource (that is, it doesn't already have a cached version), it sends an `HTTP GET` request which looks something like this:

```
GET /wds13/index.html HTTP/1.1
Host: www.webdirections.org
```

The Server Responds with something along these lines:

```
HTTP/1.1 200 OK
Date: Mon, 22 Jul 2013 06:06:22 GMT
Server: Apache/2.2.24
Last-Modified: Wed, 19 Jun 2013 01:21:17 GMT
ETag: "943d-4df77a5773d40"
Cache-Control: public, must-revalidate, proxy-revalidate
Expires: Mon, 22 Jul 2013 11:06:22 GMT
Content-Type: text/html
```

If the `Cache-control` directives permit caching the resource, then the browser **may** cache it, along with header information, such as the `Cache-control` directives, `ETag`, `Last-modified` and `max-age`.

When the same resource (that is, the identical URL) is requested subsequently and was previously cached, the browser inspects whether the resource **needs** validating (`Cache-control: no-cache`).

If not, then it will inspect the `max-age` value (if there is one) or `Expires` header value to determine whether the resource is stale. If the resource isn't stale, the resource is served directly from the cache, with no request sent to the server.

If the resource *is* stale, rather than immediately requesting the resource again from the server, a conditional `GET` request is sent, asking for the resource to be re-sent only if the `ETag` or `Last-modified` date of the resource currently at the server has changed.

```
GET /wds13/index.html HTTP/1.1
Host: www.webdirections.org
If-Modified-Since: Wed, 19 Jun 2013 01:21:17 GMT
If-None-Match: "943d-4df77a5773d40"
```

If the `ETag` is identical or the `Last-modified` date is the same as the currently cached version (depending on which condition we attached to our GET request), the server returns a 304 response:

```
HTTP/1.1 304 Not Modified
```

If, on the other hand, the `ETag` is different or the `Last-modified` date different from the cached version, the resource is once again downloaded and cached according to the current `Cache-control` directives, just like the first time it was downloaded (noting that the directives may have been changed in the meantime).

Suggested Caching Practices

As we saw, caching is a delicate balancing act between:

- reducing network traffic, and so increasing performance
- ensuring content is always up to date

The first choice we need to make is whether a resource should be cacheable or not. Where resources shouldn't be cached, `Cache-Control: no-store` specifies that neither the browser, nor a proxy should cache them. Where resources won't change for a long period we can set a long expiry period, using `max-age: 31536000` (that is one year in seconds, the maximum recommended value for `max-age`). But remember in that case, the browser won't validate them again for a year, so if the resource in fact does change, we'll need to change its URL.

The big challenge when it comes to caching is for resources that change, but not necessarily all that frequently, such as CSS and JavaScript files and the HTML for some websites. If we're too aggressive in our caching, we won't have these resources refresh when they are updated. Not aggressive enough and they'll be downloaded more frequently than they need to be. Or we'll be making too many conditional GET requests to check whether the resource is stale.

There are a number of recommended practices for this situation, which are covered in some of the links in "Further reading", below. In particular, Mark Nottingham's **Caching Tutorial**, by one of the world's leading experts in HTTP, has detailed information on best practices for caching.

Meta Elements and Caching Instructions

One challenge of HTTP caching is that it requires changes to the server environment, something which varies from server to server, and which is definitely somewhat of a specialized skill. It's also not uncommon that we, humble front-end developers, may not have sufficient server access, or knowledge to do this.

As a way around this, we can, in fact, include the same information from HTTP headers directly in our HTML, using `meta` elements. We use the attribute `http-equiv` with a value of the header name, as well as a `content` attribute with the value of the directive. For example, the equivalent of `Cache-Control: no-store` would be:

```
<meta http-equiv="Cache-Control" content="no-store">
```

However, these will only be read by the browser, as CDNs and proxy servers will rarely, if ever, read the content of a resource, only the HTTP headers. Of course, this will only apply to HTML documents, not CSS, images and JavaScript

3

chapter 3-1

Web Storage

In this chapter we'll look at Web Storage, a simple, in-browser database that gets rid of much of the need for cookies. With it, we can dramatically reduce the need for server side functionality. Google, Bing and other high traffic sites also use it for caching on the client.

We'll cover:
- **what is Web Storage and why do we need it**
- **sessionStorage**
- **localStorage**
- **storage events**
- **more sophisticated browser based databases with IndexedDB**
- **Browser support**

Until recently, the only ways to maintain a user's data between visits to your site have been to store it on the server or use cookies in the browser.

Both present significant security challenges and quite a good deal of effort for us as developers.

Server-side Data

Storing data on the server requires the creation and management of user accounts, sanitizing data sent to the server, worrying about server-side security risks and about security in the transmission of data between the client and the server. For many applications, storing data on the server is required, but in many other cases, simply keeping data for the client locally, during a session or between sessions, without the need to send it back and forward to the server means a lot less development work, and potentially fewer vectors for security breaches. On top of that, if our site can work offline using appcache, then even when we need to send data to the server, if the client is offline, or the server is down, we can store this locally, and then synchronise once the client reconnects, or the server comes back online.

What about cookies?

Cookies, while long used to keep data on the client during and between sessions, were *actually* designed for communication between the browser and a server that persists between sessions, so that the server could keep track of the state of previous interactions with this client (technically, they're called "HTTP Cookies"). They're typically used for identifying a user on return visits and storing details about that user (such as 'are they still logged in?'). Cookies are sent between the browser and server **in plain text, unencrypted**, each time the user opens a page. So, unless an application encrypts cookie contents, it's quite trivial to read them, particularly on public Wi-Fi networks, when used over standard HTTP (though much less easily over encrypted HTTPS).

Storing all client data on the server creates usability issues as well, as users need to be logged in each time they use that site. The heavy lifting of ensuring data is secure during transmission and on the server is left to you as the developer. The round trip between browser and server will impact on the performance of your site or application, and

it’s rather tricky to build apps which work when the user is offline if the user’s data is all stored on the server.

For all these reasons, as web applications become increasingly sophisticated, developers need ways to keep data around in the browser (particularly if we want our applications to work when the user is offline). And we want this data to be secure.

Two closely related W3C technologies exist to help keep track of information entirely in the browser. Together known as Web Storage, they allow us to store far more structured data than cookies, are much easier to develop with than cookies, and the information stored can only be transmitted to a server explicitly by the application.

- `sessionStorage` stores data during a session and is deleted by the browser once a session is finished.
- `localStorage` is almost identical, but the data stored persists indefinitely, until removed by the application.

Let’s start with `sessionStorage`, keeping in mind that we use `localStorage` almost identically.

sessionStorage

What is a session?

The key feature of `sessionStorage` is that data only persists for a session. But just what is a session? HTML5 has the concept of a “top-level browsing context”. This is in essence a browser window or tab. A session lasts for that top-level browsing context while it is open and while that top-level browsing context is pointed at the same fully qualified[1] domain (or strictly speaking, the same origin). So, the user can:

- visit different URLs within the same the domain
- visit a different domain, then return to the original domain

and they would still be in the same session.

During the session, a user may visit other pages of the same domain or other sites entirely, then return to the original domain. Any data saved in `sessionStorage` during that session will remain available, but only to pages in the original domain, and only until the tab or window is closed.

If the user opens a link to your site in another tab or window, then the new tab or window has no access to this `sessionStorage`, since this new tab or window is a new session. That window will have its own, entirely separate `sessionStorage` for the particular domain.

It’s worth noting that `sessionStorage` is also shared with pages inside subframes in the same domain as the top-level document in the window.

So, just to clarify, if we:

- visit `http://webdirections.org` in a tab and save data to sessionStorage
- then follow a link to `http://westciv.com` in this same tab
- and then come back to `http://webdirections.org` still in the same tab
- we return to the same session for `http://webdirections.org`, and the data in the original `sessionStorage` is still available

If however we:

- visit `http://webdirections.org` in a tab and save data to sessionStorage
- then follow a link to `http://webdirections.org` in a new tab or window,

the data in the original `sessionStorage` is not available to this new tab.

The one exception to this is when a browser crashes and is restarted. Typically, browsers will, in this case, reopen all the windows that were open when the browser crashed. The specification allows, in this situation, for `sessionStorage` to persist for reopened windows from before the crash (Safari, Blink, Chrome, Firefox and Opera browsers support this, IE8 does not, though IE9 and up do).

Which may sound like a great boon for the user, but, as an application developer, you may wish to consider whether you in fact **want** to persist session data after a crash. A user may consider that when their browser crashes while using a service like webmail or online banking at an internet café or other public computer that their login details have been purged. But if these were stored in sessionStorage then the next user to launch the browser will resume the session that was current when the user crashed. Ruh-roh.

What could we do about this?

Well, when a document loads we get a `load` event. Why not have an event handler, that deletes the current `sessionStorage` when this event fires?

```
window.addEventListener("load", clearSessionStorage, false);

function clearSessionStorage(){
  window.sessionStorage.clear();
}
```

We'll look at the `clear` method of `sessionStorage` more shortly.

What good is `sessionStorage`?

One very useful application would be to maintain sensitive information during a transaction, sign up, sign in and so on, which will be purged as soon as the user closes the window or tab.

It can be used to create a multi-page form or application, where the information in each page can persist and then be sent to the server all at once when the transaction is complete. It also moves some of the heavy lifting for protecting sensitive data away from application developers to the browser developer.

Applications like an email reader could use it to keep local copies of emails, which will be automatically purged as soon as the user closes the window or tab.

Using `sessionStorage`

`sessionStorage` is a property of the `window` object in the DOM. Because it is as yet not universally supported, we'll want to check that this property exists before we use it:

```
if('sessionStorage' in window) {
 //we use sessionStorage
}

else {
 //we do something else, perhaps use cookies, or another fallback
}
```

Right, so now we have our `sessionStorage` object, how do we use it?

Key-Value Pairs

`sessionStorage` stores key-value pairs. Each pair is a piece of information (the value), identified by a unique identifier (the key). Both the key and the value are strings (more on the implications of this in a moment).

We use the `setItem` method of the `sessionStorage` object to store data like so:

```
//get the value of the input with id="name"
var name = document.querySelector('#name').value;

//store this value with the key "name"
window.sessionStorage.setItem('name', name);
```

Now we've stored the value of the input "name" in an item of the `sessionStorage` object also called 'name'. It will remain there until this window or tab is closed and it will then automatically be purged by the browser when the user closes that window or tab[2].

Notice that we haven't had to create a `sessionStorage object`, initialize it or even create an item. Where supported, `sessionStorage` is waiting there, ready for us to use. And simply setting an item using `setItem` creates that item if it doesn't exist.

Reading from `sessionStorage`

There's not much point in storing these details if we can't get them back at some point. We do this by using the function `getItem` of the `sessionStorage` object, using a single parameter, the key we used to set the item.

So, to get the value of the item with the key "name", we'd use:

```
var name = window.sessionStorage.getItem('name');
```

Nonexistent items

Now, what happens if for some reason there's no item in `sessionStorage` with the key we are trying to access? In place of a string value, it returns `null`, **not** the empty string. So, it's worthwhile testing whether the result returned is not null before using it:

```
var savedEmail = window.sessionStorage.getItem('email');
if(savedEmail !== null){
 document.querySelector('#email').value = savedEmail;
}
```

Saving Data Between Sessions

When information is less sensitive, it often makes sense to store it between sessions. Particularly as websites become more application–like, and can increasingly work offline, saving preferences or the state of a document can make for much better usability. For these situations we have `localStorage`. In almost every way identical to `sessionStorage`, the key differences are that

- where the contents of a particular `sessionStorage` are only available to the window or tab were they were saved, and only for the fully qualified domain in which they were saved, with `localStorage`, any window or tab at the same fully qualified domain can access the `localStorage` for that domain.
- the data stored in localStorage persists between sessions

Best of all, using `localStorage` for persistence **between** sessions, is almost identical to using `sessionStorage`.

Using localStorage

As we've mentioned, all the methods of `localStorage` are the same as the methods of `sessionStorage`.

- we set items with `setItem`
- we get items with `getItem`

But let's look at some further features of both `sessionStorage` and `localStorage:`

localStorage.`removeItem()`

Because items in `localStorage` will otherwise persist forever, there are times we may want to delete them. We can do this with `localStorage.removeItem(key)`, using the key for the item we want to remove. We can also use this with `sessionStorage`, but since that is purged completely when the user closes the window or tab, we're less likely to want to do that.

localStorage.clear()

If we want to delete the entire `localStorage`, we can use `localStorage.clear()`. But be warned, anything your app has saved to `localStorage` for this user is gone for good. We saw a little earlier that `sessionStorage` too has a `clear` method, which we used on page load to ensure that if the browser has crashed, the `sessionStorage` isn't restored. We won't necessarily want to do that, but if there's sensitive information the user might assume is deleted when the browser crashes, you may want to do this.

localStorage.key()

As we saw, we access `localStorage` and `sessionStorage` with keys, which are strings. Web Storage provides a way of getting the keys, using the `key()` method. This takes an integer argument and returns the associated key value. For example, let's suppose we did this:

```
window.localStorage.setItem("title", "Mr");
window.localStorage.setItem("name", "John");
window.localStorage.setItem("familyName", "Allsopp");
```

Then we ask for the `window.localStorage.key(2)`, we'll get "`familyName`" (remember, indexes to arrays in JavaScript are zero-based). What good is this? Well, combined with the `length` property, which we'll see just below, we can now iterate over all the items in `localStorage` or `sessionStorage`.

localStorage.length()

We can determine how many items `sessionStorage` or `localStorage` is currently storing using the `length` property. We could then use this, along with the `key` method, to iterate over the items in the storage. Here, we'll get every item in the `localStorage` and add it to an array. I'm not saying you're going to want to do this very often, though as we'll see shortly, `localStorage` and `sessionStorage` are synchronous, and we can't be sure some or all of the items aren't stored on disk, so working with them may be slow. This is one way of moving them into memory before working on them.

```
var currentKey;
var currentItem;
var allItems = [];
for (var i=0; i < window.localStorage.length; i++) {
  currentKey = window.localStorage.key(i);
  currentItem = window.localStorage.getItem(currentKey);
  allItems.push({key: currentKey, item: currentItem});
};
```

Gotchas, Tips and Tricks

While Web Storage is not as burdened with gotchas as AppCache, there are a number of quirks and issues you'll want to be aware of to work most effectively with it. Let's take a look at some of these.

sessionStorage and localStorage store all data as strings

As mentioned earlier, the values stored in `localStorage` and `sessionStorage` are strings, which has a number of implications for developers.

Among other things, when we store boolean values, integers, floating point numbers, dates, objects and other non-string values, we need to convert to and from a string when writing to and reading from storage. Perhaps the most effective way of doing this is to use the JSON format.

JSON and localStorage

As we've just seen, when working with non-string values, if we want to store these in `localStorage` or `sessionStorage`, we'll need to convert them to strings. Then, we'll need to convert them back from strings to their original format when we get them out of storage. The most straightforward way to do this is to use the browser's JSON

object to convert to and from a string value.

If you're not familiar with it, JSON (JavaScript Object Notation) is a format for representing JavaScript values (numbers, booleans, arrays and objects) as strings. The standard JavaScript `JSON` object can convert to and from JSON strings and JavaScript values.

The JSON object has two methods:

- `JSON.stringify()`, which converts a JavaScript value to a JSON formatted string. You may be wondering why it's not `JSON.toString`. In JavaScript all objects have a `toString` method which returns the string representation of the object itself. In this case, we don't want the string representation of the JSON object, which is what `JSON.toString` would give us.

- `JSON.parse()`, which takes a string and recreates the object, array or other value that this string represents (provided the string is valid JSON)

So, when saving any non-string value to `localStorage`, we'll want to convert it to JSON, and when reading from `localStorage`, we'll want to parse it back from the JSON formatted string something like this:

```
var person = JSON.parse(window.localStorage.getItem("john"));
window.localStorage.setItem("john", JSON.stringify(person));
```

There's also a more subtle side effect of storing values as strings. JavaScript strings are 16-bit, so each character, even an ASCII character, is 2 bytes (in UTF-8, characters are one byte). This effectively halves the available storage space.

localStorage and privacy settings

While we know localStorage is a different technology from cookies, browsers largely treat them as the same from a user's privacy perspective. Where a user chooses to block a site from storing cookies, attempts to access localStorage for that site (both writing, and reading previously saved data) will report a security error.

We can test for whether localStorage is available by attempting to set an item, and catching any exceptions

```
function storageEnabled(){
      //are cookies enabled? try setting an item to see if we get an error

      try {
            window.localStorage.setItem("test", "t");
            return true
            }

      catch (exception) {
            //it's possible we're out of space, but it's only 1 byte,
            //so much more likely it's a security error
            //most browsers report an error of 18, if you want to check
            return false
            }
}
```

Private Browsing

Many browsers now have private (or 'incognito') browsing modes, where no history or other details are stored between sessions. In this situation, what happens with `sessionStorage` and `localStorage` varies widely by browser.

- Safari returns null for any item set using `localStorage.setItem` either before or during the private browsing session. In essence, neither sessionStorage nor localStorage are available in private browsing mode. Safari throws the same error as when it has exceeded the limit for that domain, `QUOTA_EXCEEDED_ERR` (see below for more) rather than a security error, as it does when cookies are disabled.
- Chrome and Opera return items set previous to private ("incognito") browsing commencing, but once private browsing commences, they treat localStorage like sessionStorage (only items set on the localStorage by that session will be returned) but like localStorage for other private windows and tabs.

- Firefox, like Chrome will not retrieve items set on localStorage prior to a private session starting, but in private browsing treats localStorage like sessionStorage for non-private windows and tabs, but like localStorage for other private windows and tabs.

Getters and Setters

In addition to using `getItem` and `setItem`, we can use a key directly to get and set an item in `sessionStorage` and `localStorage`, like so (where the key here is "`familyName`"):

```
var itemValue = window.localStorage.familyName;
window.localStorage.familyName = itemValue;
```

If we want to set or get an item using a key value we calculate within the program itself, we can do so using array notation, and the key name. The equivalent to the above example would be:

```
var keyname = "familyName"
var itemValue = window.localStorage[keyname];
window.localStorage[keyname] = itemValue;
```

localStorage and sessionStorage Limits

The Web Storage specification recommends browsers implement a limit on the amount of data `localStorage` or `sessionStorage` can save for a given domain. If you try to exceed the limit that various browsers have in place (for some browsers users can change this allowance) `setItem` throws an error. There's no way of asking `localStorage` for the amount of space remaining, so it's best to set item values within a try and catch for any error:

```
try {
  window.localStorage.setItem(key, value);
}

catch (exception) {
  //test if this is a QUOTA_EXCEEDED_ERR
}
```

If the available space for this localStorage is exceeded, the exception object will have the name `QUOTA_EXCEEDED_ERR` and an error code of 22.

As mentioned, in JavaScript strings are 16-bit, which means that each and every 1 byte character is 2 bytes. Typically on the web we use UTF-8 encoding, a 1 byte encoding. So, when saving the string "John", 4 bytes in UTF-8, we are actually storing 8 bytes. This effectively halves the available storage space.

Currently, major browsers have these limits per domain[3] on Web Storage. Note these are the sizes in bytes, and so the number of characters you can store uncompressed is half this:

- Chrome: 5MB
- Firefox: localStorage - 5MB, sessionStorage - unlimited
- Opera: 5MB
- Safari iOS: 5MB
- Internet Explorer: 10MB
- Android: localStorage - 5MB, sessionStorage - unlimited
- Safari: localStorage - 5MB, sessionStorage - unlimited

If the storage needs of your application are likely to exceed 5MB, then web databases are possibly a better solution. However, the situation with web databases is complicated, with two different standards. One, Web SQL, is widely supported but deprecated. The other, IndexedDB, is currently supported in Firefox, Chrome, Opera, Android 4.4+ and IE10. We'll look at these in a little more detail shortly.

Storage Events

One of the features of `localStorage` is that the same database can be shared between multiple open tabs or windows. Which also raises the issue of how these different "top-level browsing contexts" (that's the technical term for a window

or tab in HTML5) can keep data synchronized. Here's where storage events come into play. When `localStorage` changes, a `storageChanged` event is sent to the **other** windows and tabs open for that domain (there's a reason for the emphasis).

We can create an event handler, so that when a storage object has been changed then we can be notified and respond to those changes.

```
window.addEventListener('storage', storageChanged, false);
```

Now, when localStorage is changed (by setting a new item, deleting an item or changing an existing item) our function `storageChanged(event)` will be called. The event passed as a parameter to this function has a property `storageArea`, which is the window's localStorage object (note this doesn't work for sessionStorage because sessionStorage is restricted to a single window or tab). What other information do we get in our event handler? The event has these storage specific properties:

- `key`: the key of the item changed
- `oldValue`: the value changed from
- `newValue`: the value changed to
- `url`: the URL of the page whose `localStorage` was changed

There are two things to be aware of with storage events.

- The event **only fires if the storage is changed** (not if it is simply accessed and not if we set an item to the same value that it currently has)
- In the specification, **the event is not received in the window or tab where the change occurred**, only in other open windows and tabs that have access to this localStorage. Some browsers have implemented storage events in such a way that the event is also received by the window or tab which causes the change, but don't rely on this.

While it may be useful to know a stored value has been changed if the user has two or more tabs/windows open for your site or app, storage events can be more useful than that. We can in fact use them to very simply pass messages between different open windows that are pointed to the same domain. Now, you might be thinking that we already have `postMessage` for this very purpose, but here we can kill two birds with one stone – persist the state of an application in localStorage, as well as pass a message to other open windows for the domain about the state change. Another reason this is in some ways superior to `postMessage` is that unlike with `postMessage` we don't have to know about the existence of other windows to send them messages.

How might we use storage events? Well, suppose the user logs out of our app in one window, but has other windows open for the app. We could listen for changes to `localStorage` and then log the user out in all open windows of the app.

Here's how we might listen to whether the user's signed in or out of our service in our storage event handler. We'll use an item with the key `status` to save the current status. To make things simpler (so we don't need to convert to and from a boolean value), we'll use a string value, "signed in" when the user is signed in.

```
function storageChanged(storageEvent) {
  if(storageEvent.key === "status" && storageEvent.newValue === "signed in")
  {
    //the user just signed in
    }

   else if (storageEvent.key === "status"){
    //the user just signed out
   }
}
```

Web Storage Performance Concerns

A number of high profile, widely read articles critical of `localStorage` have been published, centering on its asserted

performance shortcomings. The key criticism relates to the fact that Web Storage is synchronous. This means a script using `sessionStorage` or `localStorage` waits while `getItem`, `setItem` and other storage methods are invoked. In theory, this can impact both the browser's response to user input and execution of JavaScript in a page. In practice, I'd argue that this is not likely to be a significant problem for most cases.

To consider these concerns, I conducted tests across a number of devices and browsers which demonstrates that even for poorly implemented code that does a very significant number of `getItem` and `setItem` operations, the performance of Web Storage is unlikely to have significant impact. Yes, if you are writing hundreds of large (10s or 100s of KB of data per access) to localStorage frequently, it may not be the ideal solution. But in most situations for which Web Storage was designed, I'd suggest it's going to be adequate.

Origin restrictions

We said earlier that that `sessionStorage` and `localStorage` are restricted to windows or tabs in the same domain, but in fact, the restriction is tighter than simply the top-level domain (such as `webdirections.org`).

To have access to each other's Web Storage, tabs or windows must have the same "fully qualified domain", that is top-level domain (for example `webdirections.org`), subdomains (for example `test.webdirections.org`), and protocol (`https://webdirections.org` has a different localStorage from `http://webdirections.org`).

At first glance this might seem overly restrictive but imagine `john.wordpress.org` having access to the localStorage of `james.wordpress.org`?

Browser Support

Web Storage is supported in all versions of Internet Explorer since IE8, and Firefox, Chrome and Safari for many versions, as well as on Safari for iOS and the stock Android browser for many versions as well. The challenge for backwards compatibility is essentially limited to IE7 and older.

For browsers that don't support Web Storage there are several polyfills[4], which provide support for the localStorage API in these browsers.

JSON is supported natively in all browsers which support localStorage.

IndexedDB

We've seen that Web Storage has a number of limitations, chiefly:

- it's synchronous, so reading or writing large amounts of data may block the browser's responsiveness
- it only allows us to store strings, meaning we have to convert to and from strings for other data types
- there is typically a limit of 5MB for each domain (which, as JavaScript strings are 2 bytes per char typically, means in effect 2.5MB maximum per domain)

So, while Web Storage is a good solution for when we are storing a relatively small amount of simple data, for more complex situations, it's not an ideal solution. When a more sophisticated database solution is required, there is in fact a solution (well, two actually).

- IndexedDB is a draft standard from the W3C, and is supported in all modern browsers and devices other than Android prior to 4.4, iOS and Safari.
- Web SQL is an abandoned standard supported only in WebKit browsers.

Both are low-level APIs, and rather than use them directly, you're more likely to want to use more high level JavaScript based solutions like PouchDB, BD.js or LawnChair built on top of them than these APIs directly. We saw with Web Storage that one of the advantages was its simplicity – we don't need to create the database or the individual items, we simply set and get them. With IndexedDB, on the other hand, we need to:

- create databases explicitly
- open databases to use them
- create dataStores for databases (Web Storage has no concept of dataStores)
- create indexes explicitly (indexes with Web Storage are created when we call `setItem`)
- create a transaction for each read and write operation (Web Storage has no concept of a transaction)

In addition, IndexedDB is asynchronous, which improves performance, but means we need to create callback functions for when a read or write operation, among others, completes. So it's somewhat more complicated to use.

Using a solution like PouchDB or Lawnchair, means we get the performance and other benefits of IndexedDB, with a simpler API and more high-level functionality.

And, as Safari on the desktop and iOS devices still don't support IndexedDB, you'd have to also use Web SQL, the now abandoned SQL-based browser database standard, to cover all modern devices and browsers. Luckily, an additional advantage of PouchDB and Lawnchair is that you don't have to worry about these details, as they use the database technologies available to them, with no additional work required by us as developers. We can simply use the same API across all browsers and devices, and let these tools worry about the low-level details.

IndexedDB and WebSQL go beyond the scope of what I've set out to do with this book, and to cover in as much depth as we've covered Web Storage in this chapter would require considerably more detail than this chapter already has. For more more see the Further Reading at the end of the chapter.

The Wrap

webStorage solves a long standing challenge for web developers – reliably and more securely storing data between sessions entirely on the client-side. While there are assertions that performance limitations make localStorage "harmful", in the real world, services like Google and Bing are using localStorage, and performance experts like Steve Souders and Nicholas Zakas defend and advocate their use. That's not to say webStorage is perfect or ideal in all situations. The synchronous nature of the API and potential limits per origin do mean that in certain circumstances an alternative may be required. Web Storage is however eminently usable for a great many client side data storage needs.

Where the limitations of webStorage are too great, there's also IndexedDB and webSQL based solutions.

Resources and further reading

The ever reliable HTML5 Doctors
http://html5doctor.com/storing-data-the-simple-html5-way-and-a-few-tricks-you-might-not-have-known/

Opera Developer Network
http://dev.opera.com/articles/view/web-storage/

Mark Pilgrim's excellent "Dive into HTML5"
http://diveintohtml5.info/storage.html

Remy Sharp on the state of support for offline events
http://remysharp.com/2011/04/19/broken-offline-support/

MSDN's introduction to webStorage
http://msdn.microsoft.com/en-us/library/bg142799(v=vs.85).aspx

The road to IndexedDB
https://hacks.mozilla.org/2010/06/beyond-html5-database-apis-and-the-road-to-indexeddb/

An early walk-through of IndexedDB
https://hacks.mozilla.org/2010/06/comparing-indexeddb-and-webdatabase/

4

chapter 4-1

Are we online?

In this chapter we'll look at how we can determine whether the user actually is online or not.

We'll cover:

- **the online property of the navigator**
- **the online and offline events sent to the navigator**
- **the W3C's Network Information API**
- **other ways we might determine whether the user is online**

When we develop a website or app that might be used when either online or offline, it can be useful to know whether the browser is currently connected. Then, we might enable or disable upload or submit buttons as appropriate, or otherwise adapt a user interface or other functionality based on the current connection state.

You might have thought it was straightforward for a browser to know whether it is online or not, and then to let us know. Sadly, this isn't the case.

HTML5 gives us two ways in which we can try to determine the current online status of the browser. We can check the `onLine` attribute of the `navigator` object, or we can listen for `online` and `offline` events. We can also use the currently still-in-draft Network Information API standard supported in some browsers. Additionally there are some other hacks for trying to determine whether the user is connected as well, which we'll cover.

navigator.onLine

The `navigator` object is part of the DOM (or more accurately the BOM or "Browser Object Model") that represents the browser itself. It's traditionally most commonly used to access information about the browser version, but increasingly we're seeing device level APIs like DeviceMotion associated with the `navigator` object. One of the properties of the navigator is `onLine`, which is true if the browser is online and false if offline. Well, as we said, it's not quite that simple.

`navigator.onLine` will be `false` if the browser is definitely not connected to a network. However, it's true if it is connected to a network, even if that network is not connected to the Internet. So, while a value of false indicates we're definitely offline, a value of true does not necessarily mean we will be able to connect to a web server.

There's an added complication in Firefox and Internet Explorer. These browsers have an offline mode that allows the user to disconnect the browser from the Internet, even while the system they're running on is connected. In Firefox, and Internet Explorer 8+, `navigator.onLine` means both that the local system is connected to a network (as described above) ***and*** the browser is not in this offline mode.

In Internet Explorer 7, a value of false indicated ***solely*** that the user was in offline mode while a value of `true` indicates nothing about whether the system itself was connected to a network.

In summary, the value of `navigator.onLine` is of limited value. We can use it to determine (in all browsers from Internet Explorer 8+) that the browser is definitely offline, but not to determine that the browser definitely has a connection to the Internet. We'll see shortly that this still has some benefits.

Online and offline events

While it's good to be able to check whether the user is (probably? possibly?) online or offline, it would be nice to not have to actually constantly ask the navigator object whether the user is connected or not, but receive a notification when the user goes online or offline. We can in fact do this by providing an event handler for two different events, `offline` and `online`. In theory, we can attach this handler to the `window`, `document` or even `body` objects. But in practice, the only way to attach an event handler that works across all modern browsers which support online events is to attach it to the `window`, using `addEventListener` (but we can't use `window.online = function reference`). So, we'd ask the window to call the function `updateUI()` when the user goes offline like this:

```
window.addEventListener("offline", updateUI);
```

Again, as with `navigator.offLine`, with WebKit browsers, the events are fired when the user connects to, or loses connection to a local area network (for example by turning off Wi-Fi or unplugging from Ethernet) or, with Firefox and Internet Explorer 8+, also when the user goes into or out of offline mode.

How might we use these events or the `navigator.onLine` property?

A simple way to improve our application or page user experience would be to disable options that require the user to be online (for example, a submit button for a form) and to inform the user somehow why this is disabled.

For operations which happen without user intervention between browser and server (for example, synchronizing data using `XMLHttpRequest`), rather than attempting the operation, and waiting for a timeout, we could determine whether the user is definitely offline or not, and if offline, save the data to synchronize in localStorage, then try the operation once the browser receives an online event.

W3C Network Information API

Clearly, as web applications become more sophisticated we'd like to know more than whether the browser ***might*** be connected to the web. To this end, the W3C is currently developing the Network Information API[1], which provides information about the current network bandwidth availability.

navigator.connection

The Network Information API adds a `connection` object to the `navigator`. The `connection` object has two properties:

- `bandwidth`: a number that represents the current ***download*** bandwidth. If the browser is offline, this is `0`. If the bandwidth is unknown, the value is `Infinity`.
- `metered`: is `true` if the connection is metered, and so the user is currently paying for their bandwidth

The connection object also receives a `change` event when the connection changes. This could be because the user has moved from a metered to non-metered connection or because the network speed has changed. If a change occurs, we could determine whether the current `bandwidth` is `0`, and if so, we know the browser has now gone offline. For subsequent `change` events, we could check to see whether this value is no longer 0, in which case the browser will now be online.

Currently, the Network Information API is still a draft specification, though it is supported in Firefox and Android[2]. So why mention it? Well, as we saw, `navigator.onLine` and the online and offline events only really tell us whether a browser is definitely offline or possibly online.

But, because Mozilla based browsers support a draft version of the Network API, we can use this API to determine whether Firefox is ***really*** online.

```
navigator.connection.addEventListener('change', connectionChanged, false);
//add the change event listener

function connectionChanged(event){
        if(navigator.connection.bandwidth !== 0) {
                //We're online
        }

        else {
                //we're offline
        }
}
```

Other ways of determining whether we're offline or online

Because of the shortcomings of `onLine` and of the offline and online events, as well as lack of widespread support for the Network Information API, several developers, including Remy Sharp and Paul Kinlan have come up with a number of clever ways to try and detect whether a browser is online or offline. These include using the AppCache errors and XMLHttpRequest. Here's a quick overview of these techniques.

Using the AppCache error event

In Chapter 2 we saw that if we request an update to the AppCache and something goes wrong we get an error. Now it could be that the manifest file or one of the resources in the manifest are missing, but if we know they aren't, we might reasonably guess that the user, our server or both are offline, which, for the purposes of our site, is largely the same thing most of the time.

So, let's add an event handler for the error event to the AppCache, so that when the AppCache is updated, we record the current status.

```
window.applicationCache.addEventListener("error", onlineStatus, false);

function onlineStatus(event) {
  //we're probably offline as we got an AppCache error
}
```

One downside to this is we'll only check the online status when the page originally loads, so if the user comes online, we'll have to manually trigger an AppCache update.

```
function checkOnline() {
  //try to update the appcache. If that throws an error, it will call onlineStatus
  window.applicationCache.update();
}
```

Using XMLHttpRequest

One of the more traditional ways of determining whether a browser is online is to make an XMLHttpRequest (XHR) for a resource on a server. The idea is quite straightforward, though the code involved is somewhat convoluted, so rather than reproduce it here, in the further reading section there's a link to Remy Sharp's polyfill using XHR, so you don't have to implement your own, and Paul Kinlan's detailed HTML5 Rocks article which has a fully working example of the XHR approach (along with the AppCache approach).

Wrapping up

Reliably detecting whether a browser is online is still far from straightforward, particularly doing so across several browsers. In this chapter we looked at various ways we can get some information about the offline status of a browser, but as yet, there's no reliable standards based way of doing so across multiple browsers. There are however techniques you can use that will help you determine this with some confidence should it be something vital you need to know

about.

Resources and further reading

Working off the grid
Paul Kinlan investigates a variety of ways of detecting whether the user really is offline.
http://www.html5rocks.com/en/mobile/workingoffthegrid/

Offline and online events
From Mozilla Developer Network.
https://developer.mozilla.org/en/docs/Online_and_offline_events

Remy Sharp
Remy has some fantastic polyfills including an XHR based polyfill for offline events.
https://github.com/remy/polyfills/blob/master/offline-events.js

5

chapter 5-1

What about the File System?

In this chapter we'll look at how the browser and local file system are coming closer together with HTML5.

We'll cover:
- **the file input type**
- **the HTML5 FileList and File object**
- **how we can read the contents of local files**
- **creating URLs for local files**
- **dragging and Dropping files and folders from the local file system**
- **saving files to the local file system**

As the applications we build with web technologies aspire more and more to match it with native apps, one area where the browser continues to remain challenged is in access to files on the user's system. While this is less of an issue with smartphone and tablet platforms, where access to the file system is constrained for native apps as well, it's still an issue particularly on desktop systems. Even on tablets and smartphones, we may want to access photos that have been taken on the device, or music or video files to play inside our application.

So, how can we access images, video and other files stored on the device our browser is running on?

Browsers have, for many years, allowed users to select files to upload to a server in HTML forms, with `<input type="file">`. But the content of these files has always been hidden from any JavaScript running in the browser. So, if we have wanted to play audio or video files, or edit images that are on the user's local system, we've needed to first upload the file to a server. However if we could read the contents of these files, we can rely less on server side functionality. As you might have guessed, with HTML5, we *can* now read files from the local file system, making applications that both work better offline and work more efficiently by cutting out the need for file uploads in some circumstances.

input type="file"

The HTML `input` element of type `file` allows users to select one or more files from the local file system. It's been around for years, but before HTML5, the purpose of the file input was solely to allow users to select files to be uploaded via a `form`, and it gave developers no access to information about the files selected at all. In HTML5, the `input` elements of type `file` now give developers access to metadata about the files selected[1]. For each selected file we can access:

- the file name
- the last modification date
- the file size in bytes

How do we get these for a file?

Let's see this in action. If we want the user to be able to choose multiple files at a time, we need to give the `input` a `multiple` attribute. Then, we'll create a `change` event handler for our `file` input (which we'll give an `id` of `fileChooser`), which is called when the value of the file input changes.

```
document.querySelector("#fileChooser").addEventListener('change',filesChosen, false)
```

Now, when the user chooses one or more files with the `file input`, our event handler is called. The event handler, as usual, receives the event as an argument. The `target` attribute of the event is the `file` input, and this has an attribute `files`, a `FileList` of `file` objects selected by the user (a `FileList` is effectively an array.) Note that even if we haven't set a `multiple` attribute on our `input`, the `files` attribute is still a `FileList`.

Now, we can iterate over the `FileList`, getting each selected file in turn, and get its name, size, and last modified date. Here, we'll just get the details for the first file in the list.

```
function filesChosen(evt){
  var chosenFile = evt.target.files[0];
  var fileName = chosenFile.name; //the name of the file as a string
  var fileSize = chosenFile.size; //the size of the file in bytes, as an integer
  var fileModifiedDate = chosenFile.lastModifiedDate; //a Date object
}
```

How might we use these? Well, we could use `localStorage` to remember details about files which have been uploaded to a server, and alert users when we have already uploaded a file with the same name, size and modification date as this one, saving them an upload. Or, before uploading a file, we might determine those which may take a long time to upload due to their size, and warn the user. When the Network Information API we saw in the previous chapter is well supported, we could even estimate quite accurately given the size of a file and current network speed, how long an upload will take.

However, this information is still quite limited. We know the name of the file, but *not* the local file path of the file. And we still can't access the contents of the file. Luckily though, HTML5 gives us more than just this expanded `input` of type `file`. It also provides a way of *reading* the content of a file.

The file input is really ugly

Styling form elements with CSS has a long and painful history. While this has improved significantly in modern browsers, and with the Shadow DOM will improve more still, `input`s of type `file` are still difficult to style. So much so that the best way to style them across browsers is to hide them, use a different element for their appearance, and then use the `click()` method of the `input` element to simulate a click on the `file input.`

We hide the `input type="file"` using `opacity: 0` or `visibility: hidden`, but not `display: none` since some browsers don't allow us to call the `click` method of an `input` if it has a `display` of `none`.

Then we provide another element for users to interact with, for example a button. When users activate this element, we call `click()` for the file input.

Here's our HTML for this situation

```
<button id="fileChooserProxy" onclick="clickFileChooser()">Choose File</button>
<input type="file" id="fileChooser" onchange="chooseFiles()">
```

And the JavaScript

```
function clickFileChooser() {
      document.querySelector('input#fileChooser').click();
}
```

Reading files

To read files from the local file system, we use the `FileReader` object. This allows us to read the content of a file once we've got a file from the user (they might have used an `input` of type `file` that we just covered, or dragged a file into an element with a drag handler, which we'll look at in a moment). For security, the user must actively give us the reference to the file in order for it to be readable with the `FileReader` in one of these ways.

Unlike with localStorage, we need to create a `FileReader` object in order to read files. Why the difference? Why not just have `window.fileReader`? Well, `FileReader` reads files asynchronously, which means we can actually read multiple files simultaneously. In that case, we'd create multiple `FileReaders` and read one file with each.

A `FileReader` has a number of methods we can use to get the contents of a file in different formats:

- `readAsDataURL(file)`: returns the content of the file as a dataURL, which, for example, might then be used as the `src` of an `img` element, as an image in a style sheet or wherever dataURLs can be used.
- `readAsText(file [,encoding])`: reads the file as a string, with the given optional encoding (it defaults to `UTF-8`).
- `readAsArrayBuffer(file)`: reads the contents of a file as an ArrayBuffer (see further reading for more on the ArrayBuffer datatype)

Because files may potentially be large and because JavaScript is single threaded and, as we saw, to allow potentially reading several files simultaneously, the `FileReader` read methods are asynchronous. So, we can stop the reading of a file while it's in progress using `FileReader.abort()`.

And, because these methods are asynchronous, rather than setting the value of a variable to the result of the method as we did with localStorage, like this:

```
var pictureURL = FileReader.readAsDataURL(file);
//we don't do this, as readAsDataURL is asynchronous
```

What we need to do is listen for events that are sent to the `FileReader` object. One of the events the `FileReader` receives is `loaded`, when the file we want has been read. The `target` property of this event is the `FileReader` itself. The `FileReader` object has a property called `result`, where the contents of the file that was read is contained. So, here's how we'd *actually* get the dataURL for the contents of a file:

```
var reader = new FileReader();
//create our FileReader object to read the file

reader.addEventListener("load", fileRead, false);
//add an event listener for the onloaded event

function fileRead(event){
  //called when the load event fires on the FileReader

  var pictureURL = event.target.result;
  //the target of the event is the FileReader object instance
  //the result property of the FileReader contains the file contents
}

reader.readAsDataURL(file);
//when the file is loaded, fileRead will be called
```

Note how we begin by creating a new `FileReader` instance, unlike with localStorage, where our object already exists.

We add the event listener for the load event and in the handler for this event, we get the `target.result`, which is the value of the FileReader operation (so, a DataURL, string or arrayBuffer depending on what operation we asked the FileReader to perform).

We can listen for any of the following events on the FileReader object:

- `loadstart`: when the FileReader starts reading the file

- `progress`: intermittently while the file is loading
- `abort`: when the file reading is aborted
- `load`: when reading operation completes successfully
- `loadend`: when the file has either been loaded or the read has failed. If a read succeeds, both a `load` and `loadend` events are fired. If it fails, both an `error` and `loadend` events are fired.

Let's put this all together. Here, we'll let the user select an image file, then show it as a thumbnail. Here's our `input` element:

```
<input type="file" id="chooseThumbnail" accept="image/*">
```

and we add an event listener for when it changes

```
document.querySelector("#chooseThumbnail").addEventListener('change',showThumbNail, false)
```

When the user makes a selection, we call the function `showThumbNail()`. In HTML5 we can use `accept='image/*'` to accept any image type, rather than having to enumerate each image MIME type. The target of the event is the `input` element, which has a property `files`. This is an array-like object called a FileList. We'll simply get the first element in the FileList, and read it.

```
function showThumbNail(evt){

  var url;
  var file;

  file = evt.target.files[0];

  reader = new FileReader();
  //we need to instantiate a new FileReader object
  reader.addEventListener("load", readThumbNail, false);
  //we add an event listener for when a file is loaded by the FileReader
  //this will call our function 'readThumbNail()'
  reader.readAsDataURL(file);
  //we now read the data
}


function readThumbNail(event) {
  //this is our callback for when the load event is sent to the FileReader
  var thumbnail = document.querySelector("#thumbnail");

  thumbnail.src = event.target.result;
  //the event has a target property, the FileReader with a property 'result',
  //which is where the value we read is located
}
```

URLs for Files

Often, when we access a local file, we won't actually want to use its contents directly, but as in the example above simply use the file for some purpose (for example, if it is an image file, display the image, if it is a CSS file, use it as a style sheet). A particular case is displaying a video stream in a video element when working with `getUserMedia`. In these cases, we can get a temporary, anonymised URL from the browser to use as we would any other URL, for example as the value of an image `src` attribute. We've just seen that we could get a dataURL using `FileReader.readAsDataURL`, but this involves instantiating a `FileReader` object, creating a callback function, then getting the dataURL inside the `load` event handler. Using a URL is much simpler. In the HTML5 File API, the `window` object has a `URL` property. This has two methods:

- `createObjectURL`, which creates a URL for a given file
- `revokeObjectURL`, which destroys the reference between the URL and the file

We can't store a URL created in this way in say localStorage then reuse it in a different window, as it only persists for the life of the document while the window is open (so, the same document in two different windows can't share the same URL object created using `createObjectURL`).

Here's how we might use this, in conjunction with an `input` of type `file` to get an image from the local file system and display it as a thumbnail. The `input` element is exactly as before:

```
<input type="file" id="chooseThumbnail" accept="image/*">

function showThumbNail(evt){

  var url;
  var file;

  var thumbnail = document.querySelector("#thumbnail");
  //this is the image element where we'll display the thumbnail

  file = evt.target.files[0];
  //because there's no 'multiple' attribute set on the input
  //users can only select one
  //we'd want to check this is the right sort of file

  url = window.URL.createObjectURL(file);
  //we create our URL (for WebKit browsers we need webkitURL.createObjectURL)
  thumbnail.src = url;
  //we give our thumbnail image element this URL as its source

}
```

And we're done. No need to worry about asynchronous reads, or callback functions. Compare that with the additional complexity doing exactly the same thing with the `FileReader` that we saw a moment ago. so, when we only want to use a file, and not read its contents, this is by far the preferred solution.

Dragging and Dropping files

We've seen one way that we can get a file to work with is to use an `input` of `type="file"`. But we can also allow users to drag files into the browser window (at least with desktop browsers), using drag and drop events. This has in fact been around in a non-standardized way since Internet Explorer 5.5, but has been since standardised in HTML5 and is supported in all modern browsers.

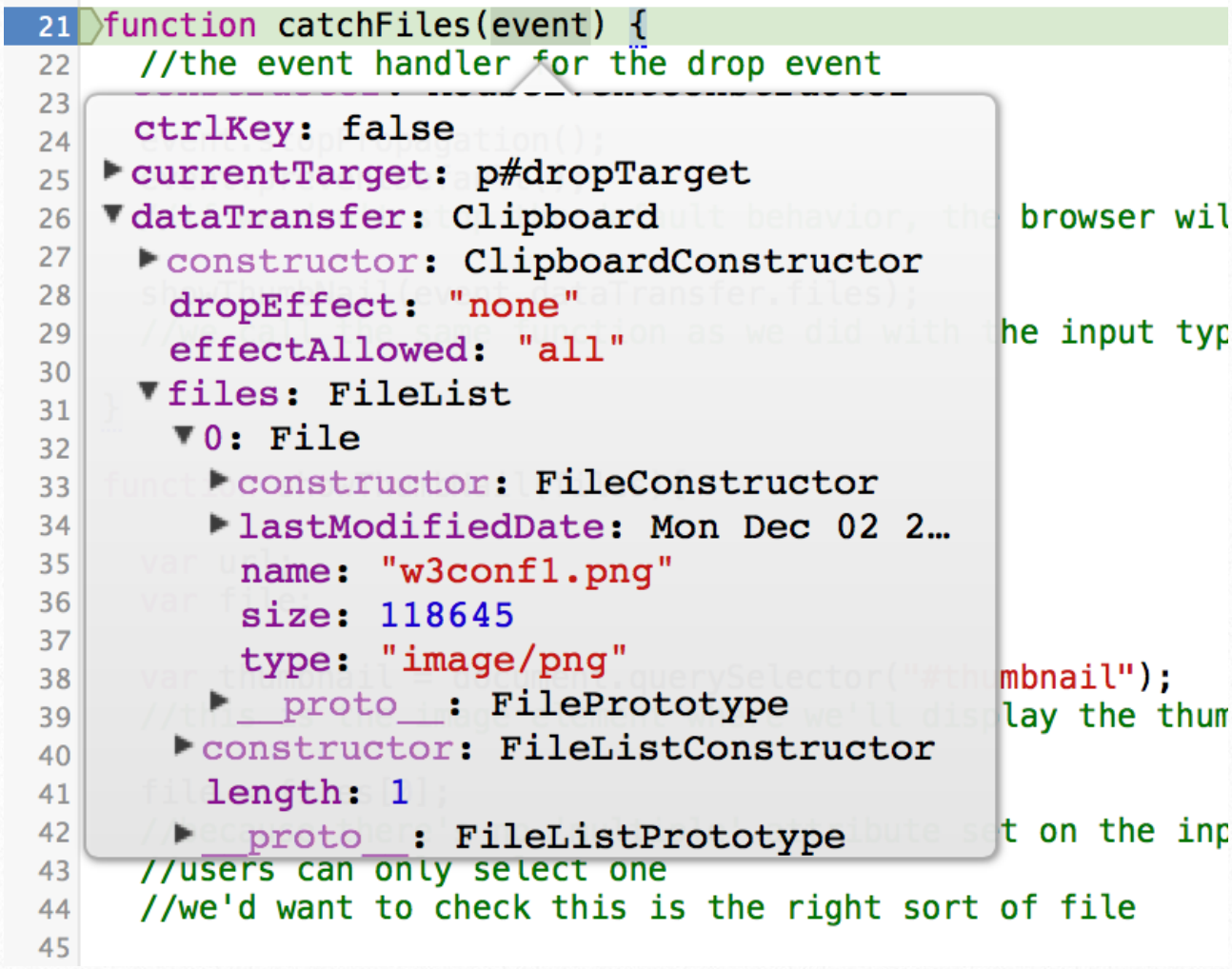
Our first step is to indicate that an element is a drop target. By default, most elements won't receive a drop of a file (and so dropping a file on them will actually load the dropped file in the browser window in place of the current page). We indicate that an element is a drop target by adding an event handler for `dragenter` or `dragover` events. In the handler we prevent the browser's default handling for these events.

```
evt.preventDefault()
```

Now we'll add an event handler for `drop` events. Unlike with the `file input`, we can't simply declare the type of content that can be dropped and let the browser take care of the detail. Rather, we need to inspect the type of content being dropped in the drop handler, to decide whether we want to accept this content.

We can also inspect the type of content being dragged before the user drops it in the `dragenter` and `dragover` events.

When the user drops the content, the `drop` event has a `dataTransfer` property, which contains the dropped content. We'll limit ourselves to just dropping files in this section. The `dataTransfer` object has a property `types`, which, if we're receiving files, will be the string "`Files`".



In developer tools, we can see the `event.target.dataTransfer` properties

So, we'll check that the user is dropping files. Then, the list of files is contained in another property of the `dataTransfer` object called `files`. We can then use this `FileList` object exactly as we did with the `input` of `type="files"`.

```
function initDragAndDrop() {
  //call this from window.onload to initialize D&D
  var dropTarget = document.querySelector("#dropTarget");

  //required to prevent browser from loading the dragged-in content.
  dropTarget.addEventListener('dragover',function(event)
  {
    event.preventDefault();
  },false);
  dropTarget.addEventListener("drop", catchFiles, false);
  //adding this tells the browser this is a drag target
}


function catchFiles(event) {
  //the event handler for the drop event

  event.preventDefault();
  //if we don't stop the default behavior, the browser will load the file in the page

  showDroppedThumbNail(event.dataTransfer.files);

}
```

Taking photos with a web page

As a little bonus, let's quickly look at how you can take a photo from a web page, in an iOS or Android device. We need an input of `type='file'`, which has an accept value of `image/*`" and a boolean `capture` attribute set on the element.

```
<input type="file" accept="image/*” capture id=“camera">
```

Now we add an event listener for change events.

```
document.getElementById('camera').addEventListener('change', photoTaken, false);
```

We’re only taking one photo at a time, so we get the first file, and for example, display it as a thumbnail

```
function photoTaken(event) {

      var file = event.target.files[0];
      var thumbnail = document.querySelector("#thumbnail");

      file = evt.target.files[0];

      url = window.URL.createObjectURL(file);
      thumbnail.src = url;

}
```

This won’t work in desktop browsers (you’ll need `getUserMedia`, part of WebRTC, an emerging HTML5 specification for that), but it will in iOS, Android and Blackberry 10 phones.

What about *saving* to the local file system?

You'll have noticed that so far, the only way we can get a file from the local system is for the user to expressly pass it to us, using `<input type="file">` or dragging and dropping. You'll also notice that even when we can read the file or get a URL to it, we still don't know anything about the location of the file on the local file system. And, the URL that we get from the window using `createObjectURL` is not only anonymized, so removing information about the file system, it's also a one-time URL - we can't for example save it in localStorage and use it the next time the user loads the page. Why? Well, we really don't want to give random web pages much, if any, access to the local file system at all. Imagine if we knew the file path to an image, say `file:///Users/johnallsopp/Documents/books/offline/tests/img.png`. Now we might try and iterate over all kinds of file names to see if we can find other images we hadn't been given explicit access to.

So, given how restrictive *read* access to the local file system is, it's no surprise that the File API gives us no way at all of `writing` to the local file system directly. Imagine an arbitrary web page being able to save a malicious native app without the user knowing about it. So, while we have localStorage and the more powerful IndexedDB to save data to the client, we can't simply write a file to the local system.

File System API

Well, that’s not entirely true. There is in fact a proposed (currently Draft Specification) File **System** API (as distinct from the File API) which provides read and write access to a virtual file system. It however hasn't been adopted by any browser other than Chrome, and the draft specification is unchanged in over 15 months as of the time of writing, so it's not likely we'll see widespread adoption of this API any time soon.

Browser support

The File API is supported in Internet Explorer 10+ and in all recent versions of Firefox, Chrome, Safari and Opera. It’s also supported in iOS Safari since 6.0 and Android from version 3.0.

Data URIs have been supported in Internet Explorer since version 8 (with size limited to 32KB in IE 8), and most versions of other browsers and platforms.

Drag and Drop from the local file system is supported on modern desktop browsers, though on tablets or smartphones. Internet Explorer 10 was the first version of IE to fully support dropping files.

Wrapping Up

6

appendices-1

Offline Apps at a Glance

This book covers many features from HTML5 and beyond that work together to help make web technology based applications work offline, and to better integrate the local file system and browser based applications. Here we'll bring together:

- all the browser support information for all of these features
- a quick reference for every object, method, property and event we covered in the book.

Appendix 1: Browser Support

This information is largely thanks to the invaluable [1]caniuse.com.

With Firefox and Chrome moving to a continuous automatic update cycle, and Safari having frequent automatic updates associated with system updates, and Internet Explorer now on auto update by default, the significant majority of users of users will no longer languish with old versions of these browsers. For the sake of completeness, we list the earliest browser version which supports the given technology,

HTTP Caching

HTTP 1, the original version of HTTP has always been essential to the web and so to web browsers. In addition, all browsers have for many years supported HTTP 1.1 headers. These are effectively universally supported.

Application Cache

HTML5 Application Cache was first fully supported in:

- Internet Explorer 10
- Firefox 3.5
- Safari 4
- Chrome 4
- Opera 10.6
- iOS Safari 3.2
- Android 2.1

As noted, the `Cache-control: no-store` directive, which should cause the browser to ignore the Application Cache is only supported in Firefox (which doesn't fire an error event) and Internet Explorer 10 (which does). Other browsers ignore the directive.

Web Storage

localStorage and sessionStorage have been fully supported since these versions of the following browsers:

- Internet Explorer 8
- Firefox 3.5
- Safari 4
- Chrome 4
- Opera 10.5
- iOS Safari 3.2
- Android 2.1

Web Storage polyfills for older browsers

The Modernizr team maintains a comprehensive list of polyfills for HTML5 features. The Web Storage polyfill list can be found here. [https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills#web-storage-localstorage-and-sessionstorage]

IndexedDB

IndexedDB has been supported since these versions of the following browsers:

- Internet Explorer 10
- Firefox 10 (required moz prefix until version 15)
- Chrome 23 (required webkit prefix only for version 23)
- Opera 15
- Android 4.4

IndexedDB is not supported in:

- Safari
- iOS Safari

Though these browsers have for quite some time supported an older, now abandoned standard, Web SQL. There is a polyfill for IndexedDB, IndexedDBShim http://nparashuram.com/IndexedDBShim, that provides the IndexedDB API on browsers which support Web SQL. In addition, CouchDB and Lawnchair mentioned in Chapter 3 provide cross platform sophisticated databases, though with a more high level API than IndexedDB

navigator.onLine

While this is supported across many browsers, it is of limited use in most. See Chapter 4 for details.

Offline events

As with `navigator.onLine`, while supported in a range of browsers, it's of limited use, as it typically won't tell us whether we are online or offline, but rather only if the user may be online or, in some browsers, whether they are in offline mode. See Chapter 4 for more.

Network Information API

A draft standard, the Network Information API is at present only supported in Firefox, where it requires the prefixed `mozConnection` to be used, and Android 2.2 upwards, where it requires the `webkit` prefix.

File API

The File API was first fully supported in:

- Internet Explorer 10
- Firefox 3.6
- Safari 6
- Chrome 13
- Opera 11.1
- iOS Safari 6
- Android 4.4

URL

`window.URL` was first fully supported in:

- Internet Explorer 10
- Firefox 4
- Safari 6 (requires `webkit` prefix only for version 6.0, unprefixed since 6.1)
- Chrome 8 (requires `webkit` prefix before Chrome 23)
- Opera 15
- iOS Safari 6 (requires `webkit` prefix prior to version 7)
- Android 4.0 (requires `webkit` prefix prior to version 4.4)

File System API

The draft File System API is only available in Chrome 13+, Opera 15+ and Android 4.4, prefixed with webkit in all three.

Drag and Drop

While dragging and dropping within the browser has long been supported by many browsers, dragging and dropping *files from the local file system* was first fully available in:

- Internet Explorer 10

- Firefox 3.5
- Safari 3.1
- Chrome 4
- Opera 12.0

As yet it is not available in:

- iOS Safari
- Android

Appendix 2: APIs and Terminology Cheat sheet

Here's your cheat sheet for every feature covered in the book.

Chapter 1 – HTTP Caching

Terminology

validating: the process by which the browser checks with a server to determine whether a resource has been changed since it was cached.

Cache-control

`Cache-control` specifies *directives* that **must** be followed by the browser (and other caching mechanisms, such as proxies and CDNs) in relation to caching.

Properties

Some values of `Cache-control` most relevant to the browser include:

- `public`: resources served with this directive *may* be cached by the browser
- `private`: the browser may cache these resources
- `no-cache`: the browser may cache these resources, but ***must*** check the resource has not been updated each time with the server before using the cached version (yes, it's not very well named).
- `no-store`: These resources must not be cached by the browser. This is the only directive that Application Cache ***must*** follow (see below for details, and browser support for information on how browsers actually support this directive)
- `max-age`: a value in seconds that tells the browser how long until the resource becomes stale, and need refreshing. The maximum recommended value is 31536000 (1 year).
- `Expire`: an HTTP 1.0 header, specifying, similar to `max-age`, when the resource will become stale. The value of `Expire` is a date, rather than number of seconds since the resource was served.
- `Etag`: Contains a unique value generated by the server each time the resource is changed, allowing the browser to determine whether a resource has been changed since caching, and requires refreshing.
- `Last-modified`: A date string specifying when the resource was last changed on the server.

Chapter 2–Application Cache

manifest file

- has a recommended extension of `.appcache`
- must be served as `text/cache-manifest MIME type`
- must begin with the string `CACHE MANIFEST` (before any content, including comments)
- is associated with an HTML document using `<html manifest="manifest.appcache">`

Manifest files have 3 types of section (there may be more than one of each section).

The CACHE Section lists relative or absolute URLs specifying resources that must be cached by the browser and used from cache, without validating the resource. Each resource to be cached ***must*** be explicitly listed, except master entries (see below). Absolute URLs may be from any domain, unless resources are served over HTTPS, where they must be from the same domain as the manifest.

The NETWORK Section lists resources that belong to an **online whitelist** and which must always be fetched from the server, regardless of whether the user is online or offline. Resources that are in neither the CACHE, nor NETWORK section **will not be used**. See Chapter 2 for more detail. Absolute URLs may be from any domain, unless resources are served over HTTPS, where they must be from the same domain as the manifest.

Entries in the NETWORK section may be relative or absolute URLs, partial URL match patterns, or the *. * means "all resources not included in a CACHE section".

The FALLBACK Section specifies fallbacks for resources that are missing when the user is offline (not for those which return 404 or other errors). Each entry has two parts, separated by a space. The first is a partial or full URL (relative or absolute) that specifies the resource or resources to be replaced. The second is full, relative or absolute URL, which specifies the resource to be used to replace missing resources. Absolute resources ***must*** be from the same domain the manifest file is served from.

Master Entries are HTML documents which link to manifest files using the `manifest` attribute in their `html` element. These are automatically added to the cache associated with a manifest and this cannot be overridden.

window.applicationCache

In HTML5, the `window` object has an `applicationCache` property. This has the following methods, properties and receives the following events.

methods

- `update`: force the browser to check for changes to the AppCache
- `swapCache`: replace the current AppCache with the newly updated AppCache (this swapped cache won't actually be used until the page is reloaded! See chapter 2 for details.)

properties

- `status`: a value from 0 to 5, representing the status of the AppCache. See Chapter 2 for details.

events

- `checking`: received when the AppCache starts checking for the manifest file or an update
- `noupdate`: received when after checking, no update is required
- `downloading`: received when the AppCache starts downloading files
- `progress`: received periodically as AppCache downloads files
- `cached`: received when the AppCache has been built or updated
- `updateready`: received when the new or updated cache is ready to be used
- `obsolete`: received when the manifest file associated with the AppCache returns an HTTP error
- `error`: received when the request for the manifest file or one of the associated resources returns an HTTP error

Chapter 3 – localStorage & sessionStorage

methods

- `getItem(keyname)`: returns the string value stored with the key "keyname". If no item with that key exists, returns `null`

- `setItem(keyname, value)`: store the value with a key of keyname. If there are no items with that key, create it.
- `removeItem(keyname)`: remove the item with the given keyname
- `clear`: remove all items from the localStorage or sessionStorage
- `key(n)`: return the *keyname* of the n-th item in the storage (n is zero-based)
- `length`: returns the number of items currently in the storage.

window events

Where localStorage is supported, an `onstorage` event is received by the `window` when localStorage for the same domain changes **from another window**. When a storage object changes from this window's localStorage, the window does **not** receive this event.

JSON

The JSON object converts native JavaScript values to and from JSON strings. It has two methods:

- `stringify(value)`: converts JavaScript values (boolean, numerical, arrays, objects etc) to strings in JSON format.
- `parse(string)`: converts a JSON string value to the corresponding native JavaScript value.

Chapter 4 – Online Events

navigator.onLine

The property is `true` if the browser is connected to a network (though it doesn't necessarily have a web connection), otherwise false. Of limited usefulness, as in most cases it doesn't actually reflect whether the user is online or offline (see Chapter 4 for details).

online and offline events

These events are sent to the window when the user goes online or offline, but are of limited usefulness, as in most cases they don't actually report whether the user is online or offline. (see Chapter 4 for details).

Network Information API

This draft API adds a `connection` object to the `navigator`, which has two properties and receives events when network conditions change.

`connection` object properties

- `bandwidth`: the speed in Mb/s of the current network connection. It is 0 if there is no connection and the global JavaScript property `Infinity` if the speed can't be determined.
- `metered`: true if the network use is charged for by use (for example with mobile data plans).

events

`change`: this is received by the `connection` object when network conditions change.

Chapter 5 – File API

The File API introduces new objects to enable the browser to access files on the local file system.

file

This represents a file on the local file system. It has 4 properties with information about the associated file.

file properties

- `name`: the name of the file on the local file system
- `size`: the size in bytes of the file
- `lastModifiedDate`: the date as an ISO string that the file was last modified
- `type`: the MIME type of the file

FileList

This is an array of `file` objects selected by the user using an `<input type="file">` or by dragging and dropping from the local file system.

FileList properties

- `length`: the number of `file`s in the `FileList`

FileReader

This is an object that allows us to read the contents of a `file` in various formats.

methods

- `readAsDataURL(file)`: read the contents of the file as a dataURL
- `readAsText(file [,encoding])`: read the file as a string, with the optional encoding which defaults to UTF-8
- `readAsArrayBuffer(file)`: read the file as an ArrayBuffer
- `abort()`: stop reading the associated file

events

- `loadstart`: received when the `FileReader` begins to read a file
- `progress`: received periodically as the `FileReader` reads the file
- `load`: received when the file has been read
- `abort`: recceived when the read was aborted by the `FileReader.abort()` method
- `error`: received when there was an error during while reading the file
- `loadend`: received when the file has been loaded or an error occurred

properties

- `readyState`: this can be 0 (empty, no read method has been called), 1 (loading, the FileReader is currently reading a file) or 2 (done, either the file has been read or an error has occurred)
- `result`: the file contents, in the format they were read (dataURL, string, ArrayBuffer or binary)

URL

This is a new property of the `window` object, for creating URLs associated with files on the local file system. Note that older WebKit browsers require the prefixed `webkitURL` object name.

methods

- `createObjectURL`(file): creates a `URL` object for the file, which can be used wherever URLs can be used. The `URL` is only valid for the window in which it was created (or `iframe`s within that window associated with the same fully qualified domain) for the lifetime of the window. It cannot be stored in `localStorage` and reused in subsequent sessions.
- `revokeObjectURL`(URLObject): removes the association between the URL object and the local file.

Drag and Drop

Drag and drop has been supported for many years to allow users to drag elements around within a window. It's only more recently enabled files to be dropped from the local file system into a page. Here we're focusing on the events and properties associated with dragging files to a window, rather than all of drag and drop.

drag and drop events

- `dragenter`: fires once on an element when a drag moves over the element
- `dragleave`: fires once on an element when a drag moves away from an element
- `dragover`: fires on an element while the drag is over the element
- `drop`: fires when a drop occurs on an element

In order for an element to receive a `drop` event, it must be a drag target. We make an element a drag target by providing a `dragover` or `dragenter` event handler.

drag and drop event properties

Events associated with drag and drop have these properties relevant to the dropping of files:

- `type`: the type of object being dragged. If it is `FileList`, the value is "`Files`" (note the capital).
- `dataTransfer.files`: The event has a property `dataTransfer`, which has its own property `files` (lowercase 'f') that contains a `FileList` of `file` objects.