JUMP START SERIES

JUMP START

Html5

Offline Applications

By Tiffany B. Brown

GET UP TO SPEED WITH HTML5 IN A WEEKEND

Summary of Contents

Preface xi
1. Detecting When the User Is Connected 1
2. Application Cache 7
3. Web Storage: the `localStorage` and `sessionStorage` Objects 21
4. Storing Data With Client-side Databases 39

JUMP START HTML5: OFFLINE APPLICATIONS

BY TIFFANY B. BROWN

Jump Start HTML5: Offline Applications

by Tiffany B. Brown

Copyright © 2013 SitePoint Pty. Ltd.

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

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Published by SitePoint Pty. Ltd.

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

About Tiffany B. Brown

Tiffany B. Brown is a freelance web developer and technical writer based in Los Angeles. She has worked on the Web for more than a decade at a mix of media companies and agencies. Before founding her consultancy, Webinista, Inc., she was part of the Opera Software Developer Relations and Tools team. Now she offers web development and consulting services to agencies and small design teams.

About SitePoint

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

About Jump Start

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

Table of Contents

Preface xi
Who Should Read This Book xii
Conventions Used xii
Code Samples xii
Tips, Notes, and Warnings xiii
Supplementary Materials xiv
Tools You'll Need xiv
What Are Offline Apps? xvi

Chapter 1 Detecting When the User Is Connected 1
Determining Whether the User Is Online 1
Listening for Changes in Connectivity State 2
Online and Offline Events in Internet Explorer 8 3
Limitations of `navigator.onLine` 3
Checking Connectivity With `XMLHttpRequest` 4
What You've Learned 6

Chapter 2 Application Cache 7
Cache Manifest Syntax 8
Saving Files Locally with the `CACHE:` Section Header 8
Forcing Network Retrieval with `NETWORK:` 9
Specifying Alternative Content for Unavailable URLs 10
Specifying Settings 10
Adding the Cache Manifest to Your HTML 11
Serving the Cache Manifest 11

Avoiding Application Cache "Gotchas" 12
Solving Gotcha #1: Loading Uncached Assets from a Cached Document 12
Solving Gotcha #2: Updating the Cache 12
Cache Gotcha #3: Break One File, Break Them All 13
Testing for Application Cache Support 13
The Application Cache API 14
The AppCache Event Sequence 14
Setting Up Our Cache Manifest 15
Setting Up Our HTML 42
Setting Up Our CSS and JavaScript 16

Chapter 3 **Web Storage: the `localStorage` and `sessionStorage` Objects** 21
Why Use Web Storage Instead of Cookies? 22
Browser Support 22
Inspecting Web Storage 23
Testing for Web Storage Support 23
Setting Up Our HTML 42
Saving Values With `localStorage.setItem()` 25
Adding an Event Listener 26
Using `localStorage.setItem` to Update Existing Values 27
Retrieving Values With `localStorage.getItem()` 27
Alternative Syntaxes for Setting and Getting Items 29
Looping Over Storage Items 29
Clearing the Storage Area With `localStorage.clear()` 31
Storage Events 31
Listening for the Storage Event 32
The `StorageEvent` Object 32
Storage Events Across Browsers 33

Determining Which Method Caused the Storage Event 33
Storing Arrays and Objects 34
Limitations of Web Storage 36

Chapter 4 Storing Data With Client-side Databases 39

The State of Client-side Databases 39
About IndexedDB 40
Setting up Our HTML 42
Creating a Database 44
Adding an Object Store 45
Adding a Record 47
Retrieving and Iterating Over Records 50
Creating a Cursor Transaction 50
Retrieving a Subset of Records 51
Retrieving or Deleting Individual Entries 52
Updating a Record 53
Deleting a Database 54
Wrapping Up and Learning More 55

Preface

One of the greatest features introduced with HTML5 is support for offline functionality. Before HTML5, if you wanted any kind of persistent storage for your web-based application, there were three options:

- cookies
- Flash local shared objects
- server-side data storage

As a means to store significant amounts of data, cookies were a poor option historically. Cookies are limited to 4KB of data, and browsers typically limit cookies to 50 per domain. That's about 200MB of data per domain, and we can't control what is deleted with the 51st cookie. But the biggest deal-breaker with cookies for offline apps is that they're restricted to the domain that created them—they fail to work offline.

Local shared objects (LSOs) are a little bit better since they're stored on the client, and aren't sent with each request. LSOs allow an application to store up to 100MB of data before asking the user to store more. Unfortunately, they also depend on the user having Flash installed.

The other alternative, of course, is server-side data storage. Server-side storage allows us to store much larger amounts of data; however, server-side storage often requires a database server, and it always requires an internet connection. If this connection is interrupted somehow—for example, while passing through a tunnel on a mobile device—we'd likely suffer from data loss.

HTML5's offline capabilities solve many of these problems. They make it possible to use applications when internet connectivity is patchy or unavailable. Mind, you'll still need a server to deliver your files to the user, and your users will require an internet connection to download those files. But offline applications allow us to work even when our internet doesn't.

Who Should Read This Book

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

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

Conventions Used

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

Code Samples

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

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

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

example.css

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

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

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

example.css *(excerpt)*

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

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

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

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

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

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

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

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me …

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

Make Sure You Always …

… pay attention to these important points.

Watch Out!

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

Supplementary Materials

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

https://github.com/spbooks/jshtml-offline1
: The downloadable code archive for this book.

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

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

Tools You'll Need

If you don't already have a favorite text editor, you'll need one. Try one of the these listed below. They're all free, and in most cases, open source. Aside from Notepad++ and Bluefish, they're all available on Mac, Linux, and Windows.

- Aptana[3] (requires Java)
- Brackets[4]
- NetBeans[5] (requires Java)

[3] http://aptana.com/
[4] http://brackets.io/
[5] https://netbeans.org/

- Notepad++[6] (Windows only)
- Bluefish[7] (Linux only)

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

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

You will also require web server software. Most of the APIs that we'll cover in this book have origin restrictions and won't work with standard filepaths. Apache HTTP Server[13], Nginx[14], or Lighttpd[15] are all open-source server packages available for Windows, Mac OS X, and Linux.

Mac OS X users can also try MAMP[16], which bundles MySQL, Apache, and PHP into one easy-to-use package. Windows users can try WAMP or XAMPP, which are similar packages for that operating system.

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

---

[6] http://notepad-plus-plus.org/
[7] http://bluefish.openoffice.nl/
[8] http://google.com/chrome
[9] http://www.opera.com/
[10] http://mozilla.org/
[11] htpp://apple.com/safari
[12] http://microsoft.com/ie
[13] http://httpd.apache.org/
[14] http://nginx.org/
[15] http://www.lighttpd.net/
[16] http://mamp.info

What Are Offline Apps?

When we talk about offline apps, we're really talking about a group of APIs along with the `window.navigator.onLine` property. Specifically, we mean:

- the application cache
- web storage (sometimes called DOM storage)
- web databases, such as IndexedDB

The application cache is a way to tell the browser which assets to download and store locally. It consists of a file syntax and a scripting API for managing and updating local copies of files.

Web storage is made up of two similar interfaces: `localStorage` and `sessionStorage`. These are two client-side key–value stores, best suited to small bits of data. They function much the same way, except that `localStorage` persists across sessions until deleted, and `sessionStorage` lasts until the current window or tab (the "browsing context") is closed.

IndexedDB is also a client-side key-value store, but it is suited to larger amounts of data. It functions more like a local database with indexing and search capabilities. There's also the Web SQL API, which is a special case.

The World Wide Web Consortium (W3C) and the Web Hypertext Applications Working Group (WHATWG) have stopped work on the Web SQL specification as it's considered deprecated; however, Web SQL is the only web database that Safari supports. Other browsers either support IndexedDB exclusively (Firefox, Internet Explorer), or support both (Chrome, Opera 15+; Opera versions 10—12 only support Web SQL).

The good news is that we can bridge the gap between browsers that support IndexedDB versus those that support Web SQL with the IndexedDBShim. As a result, we'll focus little—if at all—on Web SQL in this book.

Now that you have an overview of what we'll cover, let's get started.

Chapter 1

Detecting When the User Is Connected

Perhaps the hardest part of building an app that works offline is actually determining when the user is offline.

In this chapter, we'll cover the `ononline` and `onoffline` events of the `window` object and the `window.navigator.onLine` property. We'll also talk about their limits as well as an alternative way to test for connectivity.

Determining Whether the User Is Online

The `window.navigator.onLine` property tells us whether or not the user is connected to a network. It returns a Boolean `true` or `false` value:

```
if(navigator.onLine){
    console.log('We are connected!');
} else {
    console.log('We don't have connectivity.');
}
```

We can use this property with other offline APIs to determine when and whether to store data locally. Then we can sync it with a server once we have connectivity again.

Anytime the value of `navigator.onLine` changes, it fires one of two events:

- `offline`: when the value of `navigator.onLine` would be `false`
- `online`: when the value of `navigator.onLine` would be `true`

We can listen for these events, of course, and take action when one is fired.

Listening for Changes in Connectivity State

In Firefox, Internet Explorer 9+, and Safari, the `online` and `offline` events are fired on the `document.body` object. In Chrome and Opera 15+, however, they are fired on the `window` object.

The good news is that even when fired on the `document.body`, these `online` and `offline` events "bubble up" to the `window` object. This means we can listen for events fired there instead of setting event listeners or handlers for both.

There are two options in listening for our `online` and `offline` events. We can use the event attributes `onoffline` and `ononline`, as shown here:

```
window.ononline = function(){
    // Alert the user that connectivity is back.
    // Or sync local data with server data.
}

window.onoffline = function(){
    // Alert the user that connectivity was lost.
    // Or just silently save data locally.
}
```

Or we can use the `addEventListener` method and listen for the `online` or `offline` event:

```
window.addEventListener('offline', offlineHandler);
window.addEventListener('online', onlineHandler);
```

The `addEventListener` method requires two parameters: the event we want to listen for, and the function to invoke — known as a **callback** — when the event occurs.

For compatibility with some older versions of Firefox (< 6.0) and Opera (< 11.6), you will need to add a third parameter. This parameter tells the browser whether to use event capture or event bubbling. A value of `true` means that events should be captured. A `false` value means that events should bubble. In general, you should set it to `false`.

Most modern browsers use event bubbling by default, making the third parameter optional in those browsers. If you'd like to know more about bubbling versus capture, read Peter-Paul Koch's 2006 article, "Event order."[1]

Online and Offline Events in Internet Explorer 8

The exception to this rule is Internet Explorer 8. IE8 fires `online` and `offline` events at the `document.body` object, but they do not bubble. What's more, Internet Explorer doesn't support `addEventListener()`. You'll need to use the `attachEvent()` method instead, or a polyfill that smooths over the differences.

That said, the only component of offline applications that Internet Explorer supports is web storage (which we'll cover in Chapter 3). Internet Explorer 8 does have some vendor-specific storage features, but in the interest of promoting web standards and shortening the length and complexity of this book, we will not cover them here.

Limitations of `navigator.onLine`

Here's the downside of `navigator.onLine`: it's unreliable. What's more, it works differently depending on the browser and operating system.

Chrome, Safari, Opera 15+, and Internet Explorer 9+ can detect when the user has lost connectivity with the network. Losing or turning off a Wi-Fi connection will trigger an `offline` event in those browsers. Reconnecting will trigger an `online`

[1] http://www.quirksmode.org/js/events_order.html

event. The value of `navigator.onLine` will be `true` if connected to the Internet, and `false` if it is not.

Opera 12 (and older versions) will not fire `online` or `offline` events at all. It does, however, report `false` for `navigator.onLine` when the user is disconnected from a network and `true` when connected.

Firefox for desktop and laptop computers is a special case. Whether it fires `offline` or `online` events depends on user activity. Firefox fires an `offline` event when the user selects the **Work Offline** menu option, and fires an `online` event when the user deselects it. It's wholly unrelated to whether there is a functional internet connection. Even if the user has disconnected from the network, the value of `navigator.onLine` will be `true` if the user hasn't also elected to work offline.

Firefox for Android and Firefox OS work differently, however. Those versions of the browser will fire `online` and `offline` events when connectivity is lost or turned off (as with **Airplane Mode**).

Where all browsers struggle, however, is when there is *network* connectivity but not internet connectivity. If the browser can connect to a local network, the value of `navigator.onLine` will be `true`, even if the connection from the local network to the wider internet is down.

In other words, the `navigator.onLine` property may not tell you what you want to know. So we need to use an alternative.

Checking Connectivity With `XMLHttpRequest`

A better way to determine connectivity is to do it at the time of a network request. We can do this using `XMLHttpRequest` and its `error` and `timeout` events.

In this example, we'll attempt to send the contents of our form to the server. If we have no connection, or if the connection takes too long to complete, we'll save our data to `localStorage` and tell the user.

First, let's define our `error` and `timeout` event handler:

```
var form = document.querySelector('form');
var xhrerrorhandler = function(errorEvent){
    var i = 0;
    // Save the value if we have one
    while(i < form.length){
        if(form[i].value != ''){
            localStorage.setItem(form[i].name, form[i].value);
        }
        i++;
    }
}
```

We'll use the same function for both events, since we want to do the same task in both cases. This function saves each form field name as a `localStorage` key, and each field's corresponding value as a `localStorage` value.

Next, let's define our submit event handler. This function will make the XHR request, monitor it for failure, and add our error handlers:

```
submithandler = function(submitEvent){
    var fd, ls, i=0, key;

    // Prevent form submission
    submitEvent.preventDefault();

    // If we have localStorage data, build a FormData object from it
    // Otherwise, just use the form
    if( localStorage.length > 0){
        fd = new FormData();
        ls = localStorage.length;
        while(i < ls){
            key = localStorage.key(0);
            fd.append(key, localStorage.getItem(key));
            i++;
        }
    } else {
        fd = new FormData(form);
    }

    // Set a timeout for slow connections
    xhr.timeout = 3000;
    xhr.open('POST','return.php');
    xhr.send(fd);
```

```
    // Set our error handlers.
    xhr.addEventListener('timeout', xhrerrorhandler);
    xhr.addEventListener('error', xhrerrorhandler);
}
form.addEventListener('submit', submithandler);
```

In this function, if there is `localStorage` data, we will append it to a `FormData` object. Then we'll send the `FormData` object as the payload for our request.

`FormData` is part of the XMLHttpRequest, Level 2 specification.[2] It accepts a series of key-value pairs using its `append()` method (`append(key,value)`). Alternatively, you can pass a form object as an argument to the constructor. Here we've done both, based on whether or not we have saved anything to `localStorage`. A more robust version of this demo might handle this with separate features: one for sending form data and another for syncing with the server.

This doesn't work if the server response contains an HTTP error code. You can work around this by checking the value of the `status` property in the XHR response, as shown here:

```
var xhronloadhandler = function(loadEvent){
    if( loadEvent.target.status >= 400){
        xhrerrorhandler();
    } else {
        // Parse the response.
    }
}
```

If it's 400 or greater—HTTP error codes start at 400—we'll save the data locally. Otherwise, parse the response.

What You've Learned

In this chapter, we've discussed the advantages and limitations of the `navigator.onLine` property, an alternate way to test for connectivity. Now let's talk about the browser features and APIs that let us create offline web applications. In our next chapter, we'll look at the application cache.

[2] http://www.w3.org/TR/XMLHttpRequest/

Chapter 2

Application Cache

Application Cache—"AppCache" for short—provides a way to save the assets of an application locally so that they can be used without an internet connection. It's especially well-suited to "single-page" applications that are JavaScript-driven and use local data stores such as `localStorage` and IndexedDB.

AppCache consists of two parts: the plain text file known as a *manifest*, and a DOM interface for scripting and help managing updates. By the end of this chapter, you'll know how to write a cache manifest file, how to manage updates, and some of the aspects to watch out for when using this API.

AppCache in IE

Application Cache is not supported in Internet Explorer 9 or below. You'll need to use version 10 or later. For a complete list of browsers supporting AppCache, try CanIUse.com.[1]

[1] http://caniuse.com/#search=appcache

Cache Manifest Syntax

Cache manifest files are plain text files, but they must adhere to the manifest syntax. Let's look at a simple example:

```
CACHE MANIFEST

# These files will be saved locally. This line is a comment.
CACHE:
js/todolist.js
css/style.css
```

Every cache manifest must begin with the line `CACHE MANIFEST`. Comments must begin with a `#`. Notice that each file is also listed as a separate line.

The `CACHE:` line is a section header. There are four defined section headers in total, each with a slightly different function.

- `CACHE:` explicitly states which files should be stored locally
- `NETWORK:` explicitly states which URLs should always be retrieved from the network
- `FALLBACK:` specifies which locally available file should be displayed when a URL isn't available
- `SETTINGS:` defines caching settings

Saving Files Locally with the `CACHE:` Section Header

`CACHE:` is unique because it is an optional section header—and it's the only one that is. Our cache manifest example could also be written without it:

```
CACHE MANIFEST

# These files will be saved locally. This line is a comment.
js/todolist.js
css/style.css
```

In this case, `js/todolist.js` and `css/style.css` will be cached locally, since they follow our `CACHE:` section header. So will the page that linked to our manifest, even though we haven't explicitly said that it should be saved.

In these examples, we are using relative URLs. We can also cache assets across origins but the details are a little bit more complicated.

In recent versions of Firefox, you can cache assets across origins when they share the same scheme or protocol (HTTP or HTTPS). You can, for example, cache assets from http://sitepoint.com on http://example.com. And you can cache assets from https://secure.ourcontentdeliverynetwork.com on https://thisisasecureurl.com. In these cases, the protocol is the same. What you can't do is cache HTTPS URLs over HTTP or vice-versa.

Current versions of Internet Explorer, Safari, Chrome, and Opera allow cross-origin caching regardless of scheme. Caching assets from http://sitepoint.com on https://thisisasecureurl.com will work. This behavior could change, however, since it is out of line with the specification. Don't depend on it. As of this writing, however, Chrome Canary — version 33 — doesn't allow cross-origin requests from HTTP to HTTPS and vice versa.

Forcing Network Retrieval with `NETWORK:`

`NETWORK:` allows us to explicitly tell the browser that it should always retrieve particular URLs from the server. You may, for example, want a URL that syncs data to be a `NETWORK` resource. Let's update our manifest from earlier to include a `NETWORK` section:

```
CACHE MANIFEST

# These files will be saved locally. This line is a comment.
js/todolist.js
css/style.css
```

```
NETWORK:
sync.cgi
```

Any requests for `sync.cgi` will be made to the network. If the user is offline, the request will fail.

You may also use a wildcard—or `*` character—in the network section. This tells the browser to use the network to fetch any resource that hasn't been explicitly outlined in the `CACHE:` section. Partial paths also work with the wildcard.

Specifying Alternative Content for Unavailable URLs

Using the `FALLBACK:` section header lets you specify what content should be shown should the URL not be available due to connectivity. Let's update our manifest to include a fallback section:

```
CACHE MANIFEST

# These files will be saved locally. This line is a comment.
CACHE:
js/todolist.js
css/style.css

FALLBACK:
status.html offline.html
```

Every line in the fallback section must adhere to the *<requested file name> <alternate file name>* pattern. Here, should the user request the **status.html** page while offline, he or she will instead see the content in **offline.html**.

Specifying Settings

To date, there's only one setting that you can set using the `SETTINGS:` section header, and that's the cache mode flag. Cache mode has two options: fast and `prefer-online`. The fast setting *always* uses local copies, while `prefer-online` fetches resources using the network if it's available.

Fast mode is the default. You don't actually need to set it in the manifest. Doing so won't cause any parsing errors, but there's no reason to add it.

To override this behavior, you *must* use `prefer-online`, and add this to the end of your manifest file:

```
SETTINGS:
prefer-online

NETWORK:
*
```

There's a caveat here, however. The master file—the file that *owns* the manifest—will still be retrieved from the local store. Other assets will be fetched from the network.

Adding the Cache Manifest to Your HTML

Now for the simplest part of AppCache: adding it to your HTML document. This requires adding a `manifest` attribute to your `<html>` tag:

```
<html manifest="todolist.appcache">
```

As your page loads, the browser will download and parse this manifest. If the manifest is valid, the browser will also download your assets.

You'll need to add the manifest to every HTML page that you wish to make available offline. Adding it to `http://example.com/index.html` will not make `http://example.com/article.html` work offline.

Verify Your Manifest Syntax

To avoid problems with your manifest file, verify that its syntax is correct by using a validator. One such validator is the straightforwardly named Cache Manifest Validator,[2] an online tool. If you're comfortable with Python and the command line, try Caveman.[3]

Serving the Cache Manifest

In order for the browser to recognize and treat your cache manifest as a manifest, it must be served with a `Content-type: text/cache-manifest` header. Best practice

[2] http://manifest-validator.com/
[3] https://pypi.python.org/pypi/caveman

is to use an `.appcache` extension (e.g. **manifest.appcache** or **offline.appcache**), though it's not strictly necessary.

Without this header, most browsers will ignore the manifest. Consult the documentation for your server (or ask your web hosting support team) to learn how to set it.

Avoiding Application Cache "Gotchas"

As mentioned, AppCache uses local file copies by default. This produces a more responsive user interface, but it also creates two problems that are *very* confusing at first glance:

- uncached assets do not load on a cached page, even while online
- updating assets will not update the cache

That's right: browsers will use locally stored files regardless of whether new file versions are available, even when the user is connected to the network.

Solving Gotcha #1: Loading Uncached Assets from a Cached Document

Application Cache is designed to make web applications into self-contained entities. As a result, it unintelligently assumes all referenced assets have also been cached unless we've said otherwise in our cache manifest. Browsers will look in the cache first, and if they're unable to find the referenced assets, those references break.

To work around this, include a `NETWORK:` section header in your manifest file, and use a wildcard (`*`). A wildcard tells the browser to download any associated uncached file from the network if it's available.

Solving Gotcha #2: Updating the Cache

Unfortunately, if you need to push updates to your users, you can't just overwrite existing files on the server. Remember that AppCache always prefers local copies.

In order to refresh files within the cache, you'll need to update the cache manifest file. As a document loads, the browser checks whether the manifest has been updated. If it has, the browser will then re-download all the assets outlined in the manifest.

Perhaps the best way to update the manifest is to set and use version numbers. Changing an asset name or adding a comment also works. When the browser encounters the modified manifest, it will download the specified assets.

Though the cache will update, the new files will only be used when the application is reloaded. You can force the application to refresh itself by using `location.reload()` inside an `updateready` event handler.

Cache Gotcha #3: Break One File, Break Them All

Something else to be aware of when using AppCache: any resource failure will cause the entire caching process to fail. "404 Not Found," "403 Forbidden," and "500 Internal Server Error" responses can cause resource failures, as well as the server connection being interrupted during the caching process.

When the browser encounters any of these errors, the `applicationCache` object will fire an `error` event and stop downloading everything else. We can listen for this event, of course; however, to date no browser includes the affected URL as part of the error message. The best we can do is alert the user that an error has occurred.

Safari, Opera, and Chrome *do* report the failed URL in the developer console.

Testing for Application Cache Support

To test for support, use the following code:

```
var hasAppCache = window.applicationCache === undefined;
```

If you're going to use several HTML5 features, you may also want to try Modernizr[4] to check for them all.

It's unnecessary to test for AppCache support using JavaScript, unless your application interacts with the cache programmatically. In browsers without support for Application Cache, the manifest attribute will be ignored.

[4] http://modernizr.com/

The Application Cache API

Now that we've talked about the syntax of Application Cache, and a few of its quirks, let's talk about its JavaScript API. AppCache works quite well without it, but you can use this API to create your own user interface for your application's loading process. Creating such a user interface requires understanding the event sequence, as well as a bit of JavaScript.

The AppCache Event Sequence

As the browser loads and parses the cache manifest, it fires a series of DOM events on the `applicationCache` object. Since they're DOM events, we'll use the `addEventListener` method to listen for and handle them.

When the browser first encounters the `manifest` attribute, it dispatches a `checking` event. "Checking" means that the browser is determining whether the manifest has changed, indicating an application update. We can use this event to trigger a message to the user that our application is updating:

```
var checkinghandler = function(){
    updatemessage('Checking for an update');
}
applicationCache.addEventListener('checking',checkinghandler);
```

If the cache hasn't been downloaded before or if the manifest has changed (prompting a new download), the browser fires a `downloading` event. Again, we can use this event to update the user about our progress:

```
var downloadinghandler = function(){
   updatemessage('Downloading files');
}
applicationCache.addEventListener('downloading',downloadinghandler);
```

If the cache manifest file hasn't changed since the last check, the browser will fire a `noupdate` event. Should we have an update, the browser will fire a series of `progress` events.

Every event is an object. We can think of this object as a container with information about the operation that triggered it. In the case of `progress` events, there are two properties to know about: `loaded` and `total`. The `loaded` property tells us where

the browser is in the download queue, while `total` tells us the number of files to be downloaded:

```
var downloadinghandler = function(event){
   var progress   = document.querySelector('progress');
   progress.max   = event.total;
   progress.value = event.loaded;
}
applicationCache.addEventListener('downloading',downloadinghandler);
```

The first time a cache downloads, the browser fires a `cached` event if everything goes well. Updates to the cache end with an `updateready` event.

If there are problems, the browser will fire an `error` event. Typically this happens when a file listed in the manifest returns an error message. But it can also happen if the manifest can't be found or reached, returns an error (such as "403 Forbidden"), or changes during an update.

Let's look at an example using a simple to-do list.

Setting Up Our Cache Manifest

First, let's define our manifest file: `todolist.appcache`. This file tells the browser which files to cache offline:

```
CACHE MANIFEST
# revision 1

CACHE:
js/todolist.js
js/appcache.js
css/style.css
imgs/checkOk.svg
```

Once the browser encounters the `manifest` attribute, it will begin downloading the files listed, plus the master HTML page.

Setting Up Our HTML

In order to set up our application, we first need to create an HTML document. This is a single-page application without a lot of functionality, so our markup will be minimal:

```
<!DOCTYPE html>
<html lang="en-us" manifest="todolist.appcache">
<head>
  <meta charset="utf-8">
  <title>To do list</title>
  <link rel="stylesheet" type="text/css" href="css/style.css">
</head>
<body>
  <section id="applicationstatus"></section>

  <section id="application" class="hidden">
    <form id="addnew">
      <p>
        <label for="newitem">
    What do you need to do today?
  </label>
        <input type="text" id="newitem" required autofocus>
        <button type="submit" id="saveitem">Add</button>
      </p>
      <p>
       <button type="button" id="delete">Delete completed</button>
     </p>
    </form>
    <ul id="list"></ul>
  </section>

  <script src="js/appcache.js"></script>
  <script src="js/todolist.js"></script>
</body>
</html>
```

Setting Up Our CSS and JavaScript

Notice here that we have added a `manifest` attribute to our `html` tag. Our stylesheet controls the appearance of our application, including the styles we'll need for our interactions. The pertinent CSS from our stylesheet is shown:

```
.hidden{
    display:none;
}
```

Our `hidden` class controls the display of our elements. We'll use DOM scripting to add and remove this class name from our elements when the browser has fired a particular event.

First, let's define our global variables:

```
var appstatus, app, progress, p;

appstatus = document.getElementById('applicationstatus');
app = document.getElementById('application');
progress = document.createElement('progress');
p = document.createElement('p');

appstatus.appendChild(p);
appstatus.appendChild(progress);
```

We'll also create an `updatemessage` function that we'll use to update our messages:

```
/* Create a reusable snippet for updating our messages */
function updatemessage(string){
    var message = document.createTextNode(string);
    if(p.firstChild){
        p.replaceChild(message, p.firstChild)
    } else {
        p.appendChild(message);
    }
}
```

Next, let's set up our checking event handler. This function will be called when the browser fires a `checking` event:

```
function checkinghandler(){
    updatemessage('Checking for an update');
}
applicationCache.addEventListener('checking', checkinghandler);
```

Here, we've updated our application to tell the user that the browser is checking for an update.

Let's also add a `progress` event handler. This handler will update our application with the index of the file currently being downloaded:

```
function progresshandler(evt){
    evt.preventDefault();
    progress.max   = evt.total;
    progress.value = evt.loaded;
    updatemessage('Checking for an update');
}
applicationCache.addEventListener('progress', progresshandler);
```

If this is the first time our application cache is downloading our application, it will fire a `cached` event. In this function, we're going to hide `div#applicationstatus`, and show `div#application` by adding and removing the `hidden` class:

```
function whendonehandler(evt){
    evt.preventDefault();
    appstatus.classList.add('hidden');
    app.classList.remove('hidden');
}
applicationCache.addEventListener('cached', whendonehandler);
```

This is what makes our application active. We'll also use this function to handle our `noupdate` event. If there isn't an application update, we'll just hide the status screen and show the application:

```
applicationCache.addEventListener('noupdate', whendonehandler);
```

Finally, we need to take an action when the browser has downloaded an update. Once the cache updates, the web application won't use the latest files until the next page load.

In order to force an update, we reload our page using `location.reload()`. And we can invoke `location.reload()` when the browser fires the `updateready` event:

```
function updatereadyhandler(){
    location.reload();
}

applicationCache.addEventListener('updateready',updatereadyhandler);
```

That's it! We have a loading interface for our application. Now let's take a look at the application itself. In the next chapter, we'll take a look at web storage: the `localStorage` and `sessionStorage` objects.

Chapter 3

Web Storage: the `localStorage` and `sessionStorage` Objects

Web storage[1] (also known as DOM storage) is a simple client-side, key-value system that lets us store data in the client. It consists of two parts: `localStorage` and `sessionStorage`.

The main difference between `localStorage` and `sessionStorage` is persistence. Data stored using `localStorage` persists from session to session. It's available until the user or the application deletes it.

Data stored using `sessionStorage`, on the other hand, persists only as long as the browsing context is available. Usually this means that we lose our `sessionStorage` data once the user closes the browser window or tab. It may, however, persist in browsers that allow users to save browsing sessions.

---

[1] http://dev.w3.org/html5/webstorage/

In this chapter, we'll talk about some advantages web storage offers over cookies. We'll then discuss how to use the Web Storage API and look at some of its limitations.

Why Use Web Storage Instead of Cookies?

There are three main reasons to use `localStorage` or `sessionStorage` over cookies:

- HTTP request performance
- data storage capacity
- better protection against cross-site scripting attacks

Cookies are included with each request to the server, increasing the size and duration of each request. Using smaller cookies helps, but using `localStorage` and `sessionStorage` helps most of all. Data is stored locally, and sent only when requested.

Another advantage of using web storage is that you can store much more data. Modern browsers begin to max out at 50 cookies per domain. At 4 kilobytes per cookie, we can safely store about 200KB before some browsers will start to delete cookies. And no, we're unable to control what's deleted. Web storage limits, on the other hand, range from about 2.5MB in Safari and Android to about 5MB in most other browsers. And we can delete data by key.

Finally, web storage is less vulnerable to cross-site scripting attacks than cookies. Cookies adhere to a same-domain policy. A cookie set for .example.com can be read or overwritten by any subdomain of example.com, including hackedsubdomain.example.com. If sensitivedata.example.com also uses the .example.com cookie, a script at hackedsubdomain.example.com can intercept that data.

A same-origin restriction means that only the origin that created the data can access the data. Data written to `localStorage` by `http://api.example.com` will not be available to `http://example.com`, `http://store.example.com`, `https://example.com`, or even `http://api.example.com:80`.

Browser Support

Of all of the APIs discussed in this book, Web Storage is the most widely supported among major browsers. It's available in Internet Explorer 8+, Firefox 3.5+, Safari

4+, Opera 10.5+, and Android WebKit. There are a few polyfill scripts that mimic the Web Storage API in browsers that lack it. One such script is Storage polyfill by Remy Sharp.[2] It uses cookies and `window.name` to mimic `localStorage` and `sessionStorage` respectively.

Inspecting Web Storage

For this chapter, use Chrome, Safari, or Opera. The developer tools for these browsers let you inspect and manage web storage values. In Chrome (seen in Figure 3.1), Safari, and Opera 15+, you can find web storage under the **Resources** panel of the developer tools. In Opera 12 (the only version of Opera available for Linux users), you can find it under the **Storage** panel of Dragonfly.

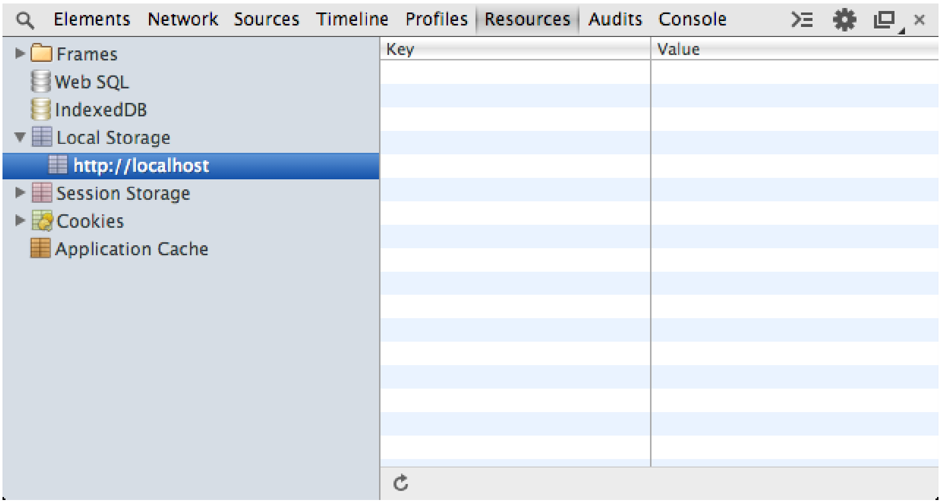

Figure 3.1. The Local Storage inspector in Google Chrome.

Testing for Web Storage Support

Both `localStorage` and `sessionStorage` are attributes of the `window` object. We can test for browser support by testing whether these attributes are `undefined`:

[2] https://gist.github.com/remy/350433

```
if( typeof window.localStorage == 'undefined' ){
    // Use something other than localStorage.
} else {
    // Save a key-value pair.
}
```

Testing for `sessionStorage` works similarly; use `typeof window.sessionStorage == 'undefined'` instead. Typically, though, if a browser supports `localStorage` it also supports `sessionStorage`, and vice versa.

Let's look at using the Storage API by building a simple to-do list. We'll use `localStorage` to save our to-do items and their status. Although we'll focus on `localStorage` in this chapter, keep in mind that this also works for `sessionStorage`. Just swap the attribute names.

Setting Up Our HTML

A basic HTML outline is fairly simple. We'll set up a form with aninput field and a button for adding new items. We'll also add buttons for deleting items. And we'll add an empty list. Of course, we'll also add links to our CSS and JavaScript files. Your HTML should look a little like this:

```
<!DOCTYPE html>
<html lang="en-us">
<head>
  <meta charset="utf-8">
  <title>localStorage to do list</title>
  <link rel="stylesheet" href="css/style.css">
</head>
<body>
  <form>
    <p>
      <label for="newitem">What do you need to do today?</label>
      <input type="text" id="newitem" required>
      <button type="submit" id="saveitem">Add</button>
    </p>
    <p>
      <button type="button" id="deletedone">Delete completed</button>
      <button type="button" id="deleteall">Reset the list</button>
    </p>
  </form>
  <ul id="list"></ul>
```

```
  <script src="js/todolist.js"></script>
</body>
</html>
```

Our form will resemble the one seen in Figure 3.2.

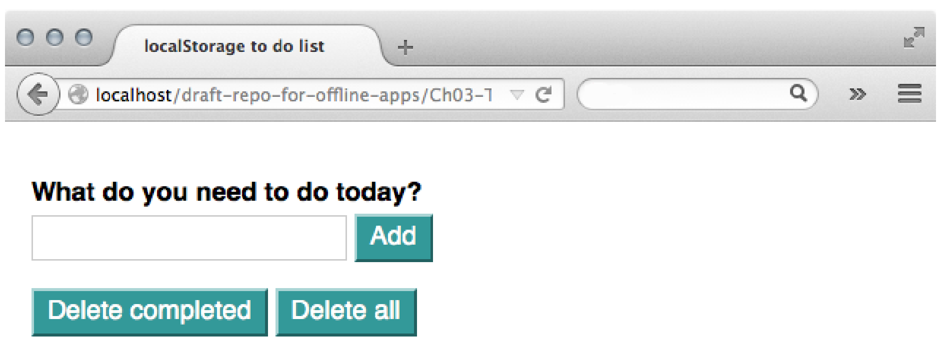

Figure 3.2. Our to-do list interface rendered in Firefox after adding the barest bit of CSS magic.

On form submission—when a `submit` event is fired on our form—we will append the new task to our unordered list element, and save it to `localStorage`.

Saving Values With `localStorage.setItem()`

To add an item to local storage or session storage, we need to use the `setItem` method. This method accepts two arguments: `key` and `value`, in that order. Both arguments must be strings, or be converted to strings:

```
localStorage.setItem(keyname, value);
```

For our to-do list, we'll take the text entered in `<input type="text" id="newitem" required>`, create a new list item, and save it to `localStorage`. That's what we've done in this function:

```
function addItemToList(itemtxt){
  var li = document.createElement('li'),
    list = document.getElementById('list');

  /* Saves the item to storage */
  localStorage.setItem(itemtxt, 0);

  /* Update the innerHTML value of our list item
  and add as a child of our list */
  li.innerHTML = itemtxt;
  list.appendChild(li);
}
```

Notice here that we are setting the value of our key to `0` (`localStorage.setItem(itemtxt, 0)`). For this application, we'll use `0` to indicate a task that needs to be completed and `1` for a task that is complete. When saved, these values will be converted to numeric strings.

Sanitize Your Inputs

In this case, accepting user input for keys is low-risk from a security standpoint. This application and its data is contained entirely within the user's browser. When synchronizing the data with a server, be sure to sanitize and validate your input and escape your output.

Since we want to update our list every time the form is submitted, we also need to add an event listener to our form.

Adding an Event Listener

To add an event listener, use the `addEventListener()` method and define a function that will be invoked when our event is fired:

```
var form, updateList;

form = document.forms[0];

var updateList = function(event){
    /* Prevent the default action, in this case, form submission */
    event.preventDefault();

    /* Invoke the addItemToList function */
    addItemToList( document.getElementById('newitem').value );

    /* Clear the input field */
    document.getElementById('newitem').value = '';
}

form.addEventListener('submit', updateList);
```

Using `localStorage.setItem` to Update Existing Values

Should a key already exist, using `setItem` will overwrite the old value rather than create a new record. For our to-do list, this is exactly what we want. To prevent the value of your key from being overwritten, check whether the key exists before saving the item.

We will also use `setItem` to update our task's status when it's clicked, as we'll see in the next section.

Retrieving Values With `localStorage.getItem()`

To retrieve the value of a key, we use `localStorage.getItem()`. This method accepts one argument, the key of the value we'd like to retrieve, which must be a string.

If the key exists, `getItem` will return the value of that key. Otherwise, it will return `null`. That makes it useful for determining whether a key exists before calling `setItem`. For example, if we wanted to test the presence of a `lunch` key, we might do the following:

```
if( localStorage.getItem('lunch') ){
    // do something.
}
```

In our case, we're going use `getItem` to retrieve the status of our task. Based on the value returned by `getItem`, we will update the value of our key using `setItem`.

In the code below, we've added an event listener to our unordered list rather than to each list item. We're using a technique called **event delegation**. When an event is fired on an element, it "bubbles up" to its ancestors. This means we can set a listener on its parent element, and use the `target` property of the event object to determine which element triggered the event. Because we only want to take an action if the element clicked was a list item, we need to check the `nodeName` property of the `target` object:

```
function toggleStatus(event){
    var status;
    if(event.target.nodeName == 'LI'){
        /*
        Using + is a trick to convert numeric strings to numbers.
        */
        status = +localStorage.getItem(event.target.textContent);
        if(status){
            localStorage.setItem(event.target.textContent,0);
        } else {
            localStorage.setItem(event.target.textContent,1);
        }
    /* Toggle a 'done' class */
        event.target.classList.toggle('done');
    }
}

var list = document.getElementById('list');
list.addEventListener('click',toggleStatus);
```

The `event.target` property is a pointer to the element that was clicked. Every element object also has a `textContent` attribute, which is its child text, if applicable. Each list item's `textContent` matches a `localStorage` key, so we can use it to retrieve the value we want. That's what we're doing with `status = localStorage.getItem(event.target.textContent)`.

localStorage keys and values are strings

Remember, all `localStorage` keys and values are strings. What look like numbers are actually *numeric strings*. To make comparisons with numbers or Boolean values, you'll need to convert the variable type. Here we've used a + sign to convert each key's value to a number. Zero is a **falsy** value, equivalent to but not equal to the Boolean `false`; 1 is a **truthy** value.

Alternative Syntaxes for Setting and Getting Items

Using `setItem` and `getItem` are not the only way to set or retrieve `localStorage` values. You can also use square-bracket syntax or dot syntax to add and remove them:

```
localStorage['foo'] = 'bar';

console.log(localStorage.foo) // logs 'bar' to the console
```

This is the equivalent of using `localStorage.setItem('foo','bar')` and `localStorage.getItem('foo')`. If there's a chance that your keys will contain spaces though, stick to square-bracket syntax or use `setItem()`/`getItem()` instead.

Looping Over Storage Items

On page reload, our list of to-dos will be blank. They'll still be available in `localStorage`, but not part of the DOM tree. To fix this, we'll have to rebuild our to-do list by iterating over our collection of `localStorage` items, preferably when the page loads. This is where `localStorage.key()` and `localStorage.length` come in handy.

`localStorage.length` tells us how many items are available in our storage area. The `key()` method retrieves the key name for an item at a given index. It accepts one argument: an integer value that's greater or equal to `0` and less than the value of `length`. If the argument is less than zero, or greater than equal to `length`, `localStorage.key()` will return `null`.

Indexes and keys have a very loose relationship. Each browser orders its keys differently. What's more, the key and value at a given index will change as items are

added or removed. To retrieve a *particular* value, `key()` is a bad choice. But for looping over entries, it's perfect. An example follows:

```
var i = localStorage.length;
while( i-- ){ /* As long as i isn't 0, this is true */
    console.log( localStorage.key(i) );
}
```

This will print every key in our storage area to the console. If we wanted to print the values instead, we could use `key()` to retrieve the key name, then pass it to `getItem()` to retrieve the corresponding value:

```
var i = localStorage.length, key;
while( i-- ){
    key = localStorage.key(i);
    console.log( localStorage.getItem(key) );
}
```

Let's go back to our to-do list. We have a mechanism for adding new items and marking them complete. But if you reload the page, nothing happens. Our data is there in our storage area, but not the page.

We can fix this by adding a listener for the `load` event of the `window` object. Within our event handler, we'll use `localStorage.length` and `localStorage.key()` along with `localStorage.getItem(key)` to rebuild our to-do list:

```
function loadList(){
    var len = localStorage.length;
    while( len-- ){
        var key = localStorage.key(len);
        addItemToList(key, localStorage.getItem(key));
    }
}
window.addEventListener('load', loadList);
```

Since we're working with existing items in this loop, we want to preserve those values. Let's tweak our `addItemToList` function a bit to do that:

```
function addItemToList(itemtxt, status){
    var li = document.createElement('li');
```

```
    if(status === undefined){
        status = 0;
        localStorage.setItem(itemtxt, status);
    }

    if(status == true){ li.classList.add('done'); }

    li.textContent = itemtxt;
    list.appendChild(li);
}
```

We've added a `status` parameter, which gives us a way to specify whether a task is complete. When we add a task and call this item, we'll leave out the `status` parameter; but when loading items from storage as we are here, we'll include it.

The line `if(status == true){ li.classList.add('done'); }` adds a `done` class to the list item if our task status is `1`.

Clearing the Storage Area With `localStorage.clear()`

To clear the storage area completely, use the `localStorage.clear()` method. It doesn't accept any parameters:

```
function clearAll() {
  list.innerHTML = '';
  localStorage.clear();
}
```

Once called, it will remove *all* keys and their values from the storage area. Once removed, these values are no longer available to the application. **Use it carefully**.

Storage Events

Storage events are fired on the `window` object whenever the storage area changes. This happens when `removeItem()` or `clear()` deletes an item (or items). It also happens when `setItem()` sets a new value or changes an existing one.

Listening for the Storage Event

To listen for the `storage` event, add an event listener to the `window` object:

```
window.addEventListener('storage', storagehandler);
```

Our `storagehandler` function that will be invoked when storage event is fired. We're yet to define that function—we'll deal with that in a moment. First, let's take a look at the `StorageEvent` object.

The `StorageEvent` Object

Our callback function will receive a `StorageEvent` object as its argument. The `StorageEvent` object contains properties that are universal to all objects, and five that are specific to its type:

- `key`: contains the key name saved to the storage area
- `oldValue`: contains the former value of the key, or `null` if this is the first time an item with that key has been set
- `newValue`: contains the new value added during the operation
- `url`: contains the URL of the page that made this change
- `storageArea`: indicates the storage object affected by the update—either `localStorage` or `sessionsStorage`

With these properties, you can update the interface or alert the user about the status of an action. In this case, we'll just quietly reload the page in other tabs/windows using `location.reload()`.

```
function storagehandler(event){
    location.reload();
}
```

Now our list is up-to-date in all tabs.

Storage Events Across Browsers

Storage events are a bit tricky, and work differently to how you might expect in most browsers. Rather than being fired on the *current* window or tab, the `storage` event is supposed to be fired *in other windows and tabs* that share the same storage area. Let's say that our user has http://ourexamplesite.com/buy-a-ticket open in two tabs. If they take an action in the first tab that updates the storage area, the `storage` event should be fired in the second.

Chrome, Firefox, Opera, and Safari are in line with the web storage specification, while Internet Explorer is not. Instead, Internet Explorer fires the `storage` event in every window for every change, including the current window.

There isn't a good hack-free way to work around this issue. For now, reloading the application—as we've done above—is the best option across browsers.

IE's Nonstandard `remainingSpace` Property

Internet Explorer includes a nonstandard `remainingSpace` property on its `localStorage` object. We can handle our storage event differently depending on whether or not `localStorage.remainingSpace` is `undefined`. This approach has its risks, though; Microsoft could remove the `remainingSpace` property without fixing the storage event bug.

Determining Which Method Caused the Storage Event

There isn't an easy way to determine which method out of `setItem()`, `removeItem()`, or `clear()` triggered the storage event. But there is a way: examine the event properties.

If the `storage` event was caused by invoking `localStorage.clear()`, the `key`, `oldValue`, and `newValue` properties of that object will all be `null`. If `removeItem()` was the trigger, `oldValue` will match the removed value and the `newValue` property will be `null`. If `newValue` isn't `null`, it's a safe bet that `setItem()` was the method invoked.

Storing Arrays and Objects

As mentioned earlier in this chapter, web storage keys and arrays must be strings. If you try to save objects or arrays, the browser will convert them to strings. This can lead to unexpected results. First, let's take a look at saving an array:

```
var arrayOfAnimals = ['cat','dog','hamster','mouse','frog','rabbit'];

localStorage.setItem('animals', arrayOfAnimals);

console.log(localStorage.animals[0])
```

If we look at our web storage inspector, we can see that our list of animals was saved to `localStorage`, as shown in Figure 3.3.

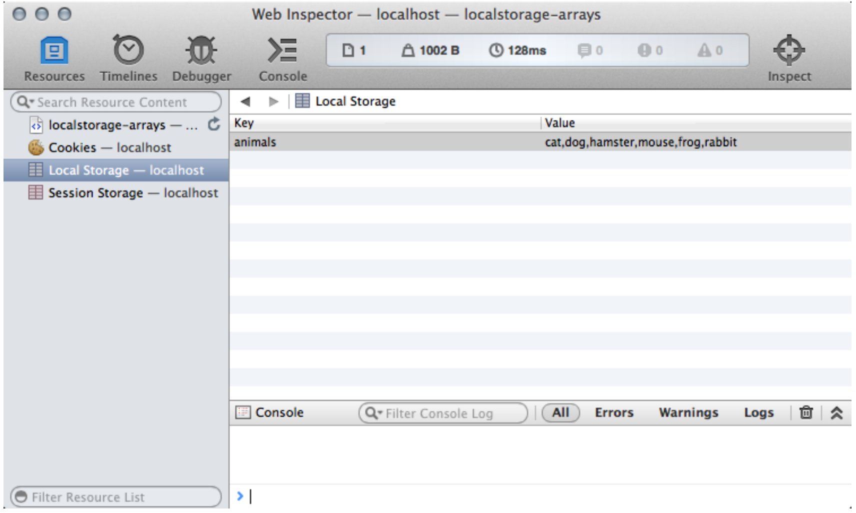

Figure 3.3. An array saved to web storage as a string in Safari's developer tools

But if you try to read the first item of that array, you'll see that the first item is the letter `c` and not `cat` as in our array. Instead, it's been converted to a comma-separated string.

Similarly, when an object is converted to a string, it becomes `[object Object]`. You lose all your properties and values, as shown in Figure 3.4.

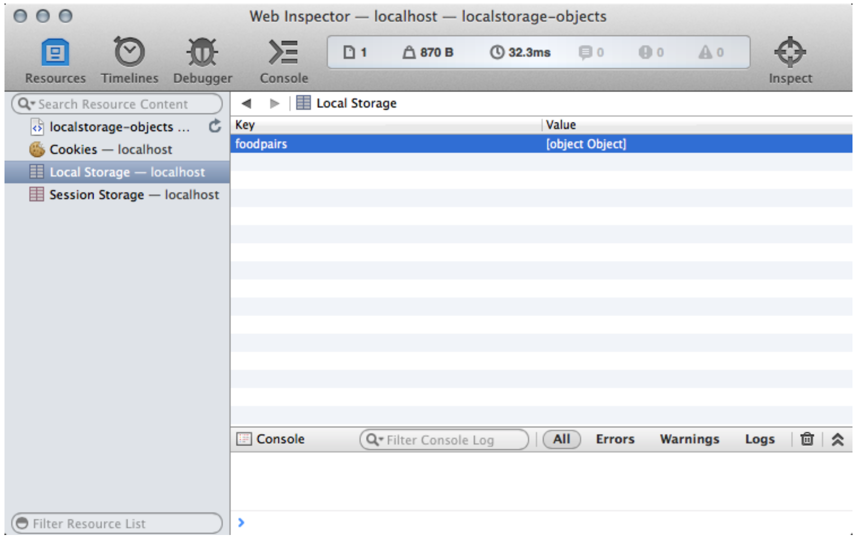

Figure 3.4. The result of saving an object to web storage in Safari's developer tools

To prevent this from happening, use the native `JSON.stringify()` method to turn your objects and arrays into strings before saving:

```
var foodcouples = {
  'ham':'cheese',
  'peanut_butter':'jelly',
  'eggs':'toast'
}

localStorage.setItem(
  'foodpairs_string',
  JSON.stringify(foodcouples)
);
```

Using `JSON.stringify()` will *serialize* arrays and objects. They'll be converted to specially formatted strings that hold indexes or properties and values, as shown in Figure 3.5.

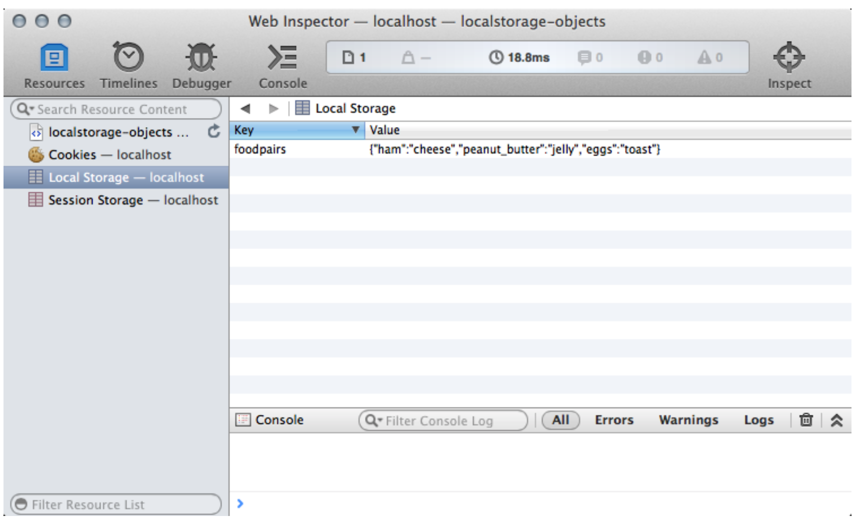

Figure 3.5. Using `JSON.stringify()` to save an object to web storage in Safari's developer tools

To retrieve these values, use `JSON.parse()` to turn it back into a JavaScript object or array. A word of caution: `JSON.parse()` and `JSON.stringify()` can affect the performance of your application. Use them sparingly, especially when getting and setting items.

Limitations of Web Storage

Web storage comes with a couple of limitations: performance and capacity.

Web storage is *synchronous*. Reads and writes are added to the JavaScript processing queue immediately. Other tasks in the queue won't be completed until the engine is done writing to or reading from the storage area. For small chunks of data, this usually is no issue. But for larger chunks of data or a large number of write operations, it will be.

Web storage also has a size limit. Should your application reach that limit, the browser will throw a `DOMException` error. Use a `try-catch` statement to catch and handle this error. In browsers that support it, you can also use `window.onerror` (or use `addEventListener` and the `error` event):

```
try {
    localStorage.setItem('keyname',value);
```

```
    return true;
} catch (error) {
    // Could alternatively update the UI.
    console.log(error.message);
}
```

Size limits vary by browser and version. The web storage specification suggests an initial size of 5MB per origin. However, Safari, currently allows 2.5MB of data to be stored per origin. Chrome, Internet Explorer, and Opera 15+ currently store 5MB of data. Firefox stores 5MB by default, but the user can adjust this limit in the `about:config` menu. Older versions of Opera (12 and below) prompt the user to raise the storage limit.

The good news is that these values may increase. The bad news is that most browsers fail to expose the amount of storage space available to the application.

Now that we've covered the ins and outs of web storage, let's take a look at a web database: IndexedDB.

Chapter 4

Storing Data With Client-side Databases

Web storage (`localStorage` and `sessionStorage`) is fine for small amounts of data such as our to-do list, but it's an unstructured data store. You can store keys and values, but you can't easily search values. Data isn't organized or sorted in a particular way, plus the 5MB storage limit is too small for some applications.

For larger amounts of structured searchable data, we need another option: web databases. Web databases such as Web SQL and IndexedDB provide an alternative to the limitations of web storage, enabling us to create truly offline applications.

The State of Client-side Databases

Unfortunately, this isn't as easy as it sounds. Browsers are split into three camps in the way they support client-side databases:

- both IndexedDB and Web SQL (Chrome, Opera 15+)
- IndexedDB exclusively (Firefox, Internet Explorer 10+)

- Web SQL exclusively (Safari)

IndexedDB is a bit of a nonstarter if you plan to support mobile browsers. Safari for iOS and Opera Mobile 11-12 support Web SQL exclusively; same for the older versions of Android and Blackberry browsers.

If you want your application to be available across desktop browsers, you're about halfway there with Web SQL. It's available in Safari, Chrome, and Opera 15+, but Firefox and Internet Explorer 10+ have no plans to add support.

Here's the thing: the World Wide Web Consortium has stopped work on the Web SQL specification. As a result, there is a risk that browsers will drop Web SQL support, or that further development will proceed in nonstandard ways. Relying on it is risky, so for that reason we will focus on IndexedDB in this chapter, and use a polyfill to support Safari and older versions of Opera.

The bright side is that there are a few JavaScript polyfills and libraries available to bridge the gap between Web SQL and IndexedDB. Lawnchair[1] supports both, and will use `localStorage` if you prefer. There's also PouchDB,[2] which uses its own API to smooth over the differences between Web SQL and Indexed DB. PouchDB also supports synchronization with a CouchDB server, though CouchDB isn't necessary for building an app with PouchDB.

In this chapter, we'll focus on IndexedDB, and use IndexedDBShim[3] for other browsers.

About IndexedDB

IndexedDB is a schema-less, transactional, key-value store database. Data within an IndexedDB database lacks the rigid, normalized table structure as you might find with MySQL or PostgresSQL. Instead, each record has a key and each value is an object. It's a client-side "NoSQL" database that's more akin to CouchDB or MongoDB. Objects may have any number of properties. For example, in a to-do list application, some objects may have a `tags` property and some may not, as evident in Figure 4.1.

---

[1] http://brian.io/lawnchair/

[2] http://pouchdb.com/

[3] http://nparashuram.com/IndexedDBShim/

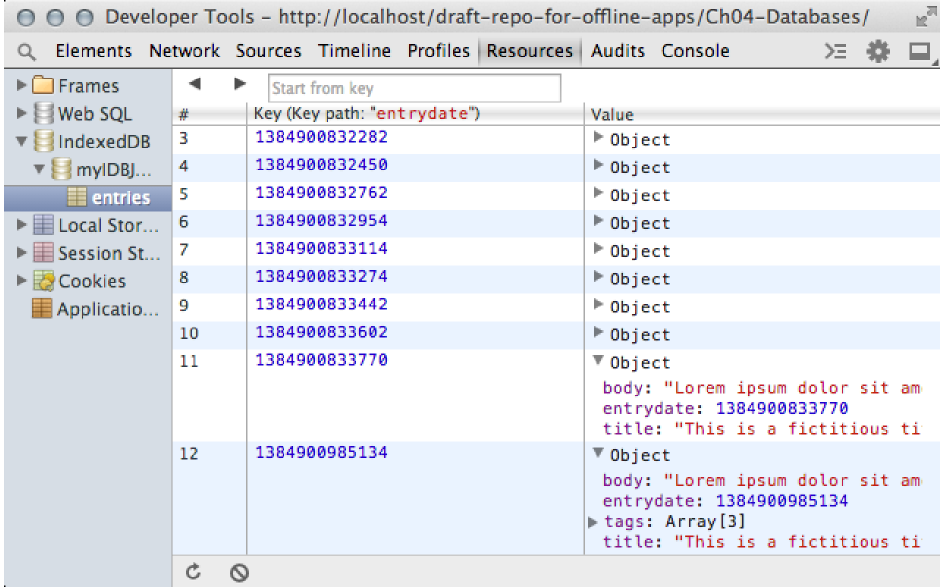

Figure 4.1. Objects with and without a `tags` property

Two objects in the same database can even have completely different properties. Usually, we'll want to use a naming convention for our object property names.

IndexedDB is also **transactional**. If any part of the read or write operation fails, the database will roll back to the previous state. These operations are known as **transactions**, and must take place inside a callback function. This helps to ensure data integrity.

IndexedDB has two APIs: synchronous and asynchronous. In the synchronous API, methods return results once the operation is complete. This API can only be used with Web Workers[4], where synchronous operations won't block the rest of the application. There is no browser support for the synchronous API at the time of writing.

In this chapter, we'll cover the asynchronous API. In this mode, operations return results immediately without blocking the calling thread. Asynchronous API support is available in every browser that supports IndexedDB.

[4] http://en.wikipedia.org/wiki/Web_worker

Examples in this book use the latest version of the IndexedDB[5] specification. Older versions of IndexedDB in Chrome (23 and earlier) and Firefox (16 and earlier) required a vendor prefix. These experimental versions had several inconsistencies between them, which have been largely worked out through the specification process. Since the standardized API has been available for several versions in both Firefox and Chrome, and is available in Internet Explorer 10+, we won't bother with older versions.

Inspecting IndexedDB Records

If you'd like to inspect your IndexedDB records, use Chrome or Opera 15+. These browsers currently have the best tools for inspecting IndexedDB object stores. With IndexedDBShim, you can use Safari's Inspector to view this data in Web SQL, but it won't be structured in quite the same way. Internet Explorer offers an IDBExplorer package[6] for debugging IndexedDB, but it lacks native support in its developer tools. Firefox developer tools are yet to support database inspections.

Now, let's look at the concepts of IndexedDB by creating a journaling application to record diary entries.

Setting up Our HTML

Before we dive into IndexedDB, let's build a very simple interface for our journaling application:

```
<!DOCTYPE html>
<html lang="en-us">
<head>
  <meta charset="utf-8">
  <title>IndexedDB Journal</title>
  <link rel="stylesheet" type="text/css" href="css/style.css">
</head>
<body>
<form>
  <p>
    <label for="tags">How would you like to tag this entry?</label>
    <input type="text" name="tags" id="tags" value="" required>
```

[5] http://www.w3.org/TR/IndexedDB/

[6] http://ie.microsoft.com/testdrive/HTML5/newyearslist/IDBExplorer.zip

```
    <span class="note">(Optional. Separate tags with a comma.)</span>
  </p>
  <p>
    <label for="entry">What happened today?</label>
    <textarea id="entry" name="entry" cols="30" rows="12" required>
    </textarea>
    <button type="submit" id="submit">Save entry</button>
  </p>
</form>

 <section id="allentries" class="hidden">
  <h1>View all entries</h1>
 </section>

<script src="js/IndexedDBShim.min.js"></script>
<script src="js/journal.js"></script>
</body>
</html>
```

This gives us a very simple interface consisting of two form fields and a **Save entry** button, as shown in Figure 4.2. We've also added a view that displays all entries after we've saved our latest one. A production-ready version of this application might have a few more screens, but for this demo this is fine.

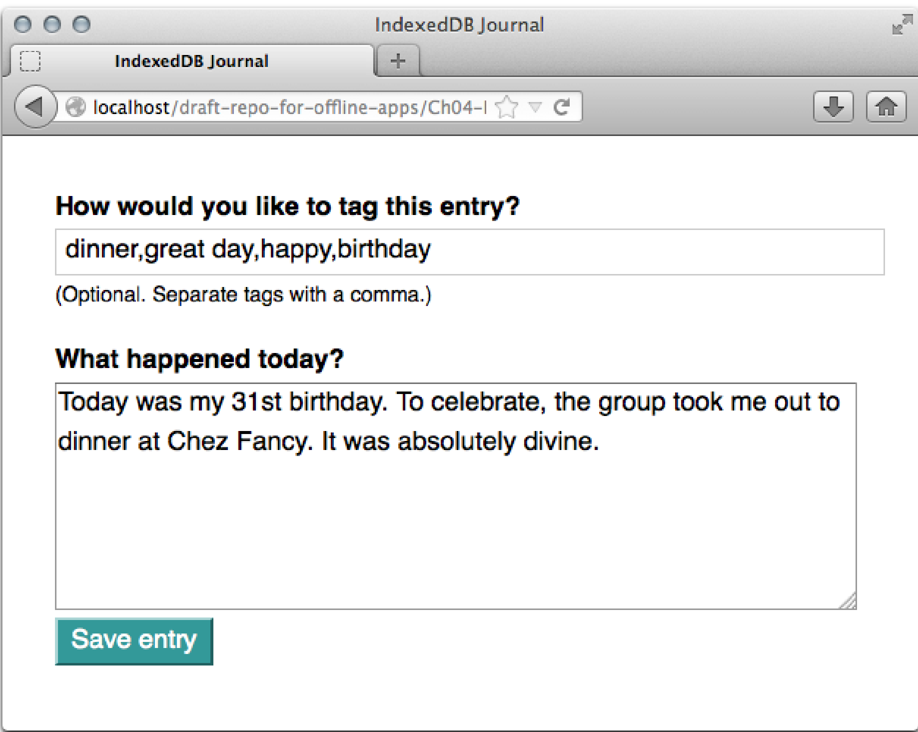

Figure 4.2. Our simple journal interface as shown in Firefox.

Creating a Database

To create a database, use the `open()` method. It accepts two arguments: the name of your database and an optional integer value that sets the version of the database:

```
var idb = indexedDB.open('myIDBJournal',1);
```

Your database name may be any string value, including the empty string (`''`). The name just ties the database to its origin. As with `localStorage`, IndexedDB databases can only be read to from—or written to by—scripts that share its origin. Remember that an origin consists of the scheme (such as `http://` or `https://`), the hostname, and port number). These must match exactly, meaning that a database created at `http://example.com` can't be accessed by scripts at `http://api.example.com` and vice versa.

Version numbers can be any integer from 1 to 2^53 (that's 9,007,199,254,740,991). Floats, or decimal values, won't work. Floats will be converted to integers. For example, 2.2 becomes 2 and 0.8 becomes 0 (and throws a `TypeError`). If you don't include a version number, the browser will use 1.

If the `open` operation is successful, the browser will fire a `success` event. We can use the `onsuccess` callback to perform transactions or, as we've done here, use it to assign our database object to a global variable:

```
// initialize the variable.
var databaseObj;
idb.onsuccess = function(successEvent){
    // set its value.
    databaseObj = successEvent.target.result;
}
```

As with all DOM events, our `successEvent` object contains a `target` property. In this case, the target is our `open` request transaction. The `target` property is also an object, and contains a `result` property, which is the result of our transaction. In this case, the value of `result` is our database.

This is a really important point to understand. *Every transaction requires an `onsuccess` handler*. The results of that transaction will always be a child property of the target object, which is a property of the event object (`event.target.result`).

Now that we've set our global `databaseObj` variable to our database object, we can use `databaseObj` for our transactions.

Adding an Object Store

Creating a database won't automatically make it do anything by itself. To begin storing data, you'll have to create at least one **object store**. If you are familiar with SQL databases, an object store is analogous to an SQL table. Object stores are where we store our **records**, or entries. IndexedDB databases can contain multiple object stores.

To add an object store, we first need to trigger a ***version change***. To trigger a version change, the version argument (the second parameter) needs to be greater than the

database's current version value. For the first release of your application, this value can be `1`.

When the database version number increases (or is set for the first time), the browser will fire an `upgradeneeded` event. Any structural changes—adding an object store, adding an index to an object store—must be made within the `onupgradeneeded` event handler method:

```
idb.onupgradeneeded = function(event){
    // Change the database.
}
```

Let's create an object store named `entries`. To do this, we need to use the `createObjectStore` method of the database object:

```
idb.onupgradeneeded = function(event){
  try {
    event.target.result.createObjectStore(
      ['entries'],
      { keyPath: 'entrydate' }
    );
  } catch(error) {}
}
```

If the object store doesn't exist, it will be created. If it does, the browser will fire a constraint error. We've wrapped this in a `try-catch` block so that we can silently handle an error should one occur. But we could also set an event handler using the `onerror` event attribute:

```
idb.onerror = function(event){
    // Log the error to the console, or alert the user
}
```

At a minimum, `createObjectStore` requires a name argument. It must be a string, but this string can't contain any spaces. The second argument is optional, but it must be a **dictionary**. Dictionaries are objects that have defined properties and values. For `createObjectStore`, those properties and values are as follows:

- `autoIncrement`: Must be either `true` or `false`; auto-generates keys and indexes for each record in the store. Default value is `false`.

- `keyPath`: Specifies which object property to use as an index for each record in the store. Default value is `null`. Makes the named field a required one.

Here we've chosen to set a `keyPath`. This means that every object we add to the database will need to contain an `entrydate` property, and the value of the `entrydate` property will become our key.

You don't have to set `autoIncrement` or `keyPath`. It's possible to add our object store without it. If you choose not to set either, you *must* set a key for every record stored. We'll discuss this in the next section.

Notice that we didn't use our `databaseObj` variable with our `onupgradeneeded` callback? That's because the `onupgradeneeded` event is fired before the `onsuccess` event when working with IndexedDB. It won't be defined when we need it.

Adding a Record

Adding records is a little more complicated. We need to create a **transaction**, and then take an action once the transaction completes. The process is roughly as follows:

1. Open a transaction connection to one or more object stores.
2. Select which object store to use for the transaction request.
3. Create a request by invoking the `put`, `add`, `delete` or `get` methods.
4. Define an `onsuccess` handler to process our results.

These steps need to happen within a callback or event handler of some kind. Here we'll do it when our journal entry form is submitted:

```
document.forms[0].onsubmit = function(submitEvent){
  var entry, i, transaction, objectstore, request, fields;

  fields = submitEvent.target;

  // Prevent form submission and page reload.
  submitEvent.preventDefault();

  /* Build our record object */
  entry = {};
```

```
  entry.entrydate = new Date().getTime();

  for( i=0; i < fields.length; i++){
    if( fields[i].value !== undefined ){
      entry[fields[i].name] = fields[i].value;
    }
  }

  /* Set our success handler  */
  request.onsuccess = showAllEntries;

  /* Start our transaction. */
  transaction = databaseObj.transaction(['entries'],'readwrite');

  /* Choose our object store (the only one we've opened). */
  objectstore = transaction.objectStore('entries');

  /* Save our entry. We could also use the add() method */
  request = objectstore.put(entry);
}
```

There's a lot happening in this function, but the most important parts are the following three lines:

```
/* Start our transaction. */
transaction = databaseObj.transaction(['entries'],'readwrite');

/* Choose our object store (the only one we've opened). */
objectstore = transaction.objectStore('entries');

/* Save our entry */
request  = objectstore.put(entry);
```

In these lines, we've first created a transaction by calling the `transaction` method on our database object. `transaction` accepts two parameters: the name of the object store or stores we'd like to work with, and the mode. The mode may be either `readwrite` or `readonly`. Use `readonly` to retrieve values. Use `readwrite` to add, delete, or update records.

Next, we've chosen which object store to use. Since the transaction connection was only opened for the `entries` store, we'll use `entries` here as well. It's possible to open a transaction on more than one store at a time, however.

Let's say our application supported multiple authors. We might then want to create a transaction connection for the `authors` object store at the same time. We can do this by passing a **sequence**—an array of object store names—as the first argument of the `transaction` method, as shown here:

```
trans = databaseObj.transaction(['entries','authors'], 'readwrite');
```

This won't write the record to both object stores. It just opens them both for access. Which object store will be affected is determined by the `objectStore()` method.

Use Square Brackets

In newer versions of Chrome (33+) and Firefox, you may pass a single object store name to the transaction method without square brackets; for example, `databaseObj.transaction('entries','readonly')`. For the broadest compatibility, though, use square brackets.

`req = objectstore.put(entry);` is the final line. It saves our `entry` object to the database. The `put` method accepts up to two arguments. The first is the value we'd like to store, while the second is the key for that value: for example: `objectStore.put(value, key)`.

In this example, we've just passed the `entry` object to the `put` method. That's because we've defined `entrydate` as our `keyPath`. If you define a `keyPath`, the property name you've specified will become the database key. In that case, you don't need to pass one as an argument. If, on the other hand, we didn't define a `keyPath` and `autoIncrement` was `false`, we *would* need to pass a key argument.

When the `success` event fires, it will invoke the `showAllEntries` function.

`put` Versus `add`

The IndexedDB API has two methods that work similarly: `put` and `add`. You can only use `add` when adding a record. But you can use `put` when adding *or updating* a record.

Retrieving and Iterating Over Records

You probably noticed the line `request.onsuccess = showAllEntries` in our form's `onsubmit` handler. We're yet to define that function, but this is where we'll retrieve all our entries from the database.

To retrieve multiple records there are two steps:

1. run a **cursor** transaction on the database object
2. iterate over the results with the `continue()` method

Creating a Cursor Transaction

As with any transaction, the first step is to create a transaction object. Next, select the store. Finally, open a **cursor** with the `openCursor` method. A cursor is an object consisting of a range of records. It's a special mechanism that lets us iterate over a collection of records:

```
function showAllEntries(){
  var transaction, objectstore, cursor;

  transaction = databaseObj.transaction(['entries'],'readonly');
  objectstore = transaction.objectStore('entries');
  cursor      = objectstore.openCursor();
};
```

Since we want to show our entries once this transaction completes, let's add an `onsuccess` handler to our cursor operation:

```
function showAllEntries(){
  var transaction, objectstore, cursor;

  transaction = databaseObj.transaction(['entries'],'readonly');
  objectstore = transaction.objectStore('entries');
  cursor      = objectstore.openCursor();

  cursor.onsuccess = function(successEvent) {
    var resultset = successEvent.target.result;
    if( resultset ){
      buildList( resultset.value );
    }
```

```
    resultset.continue();
  };
};
```

Within this handler, we're passing each result of our search to a `buildList` function. We won't discuss that function here, but it is included in this chapter's code sample.

That final line—`resultset.continue()`—is how we iterate over our result set. The `onsuccess` handler is called once for the entire transaction, but this transaction may return multiple results. The `continue` method advances the cursor until we've iterated over each record.

Retrieving a Subset of Records

With IndexedDB, we can also select a subset or ***range*** of records by passing a key range to the `openCursor()` method.

Key ranges are created with the `IDBKeyRange` object. `IDBKeyRange` is what's known as a ***static*** object, and it's similar to the way the `Math()` object works. Just as you'd type `Math.round()` rather than `var m = new Math()`, with `IDBKeyRange`, you must always use `IDBKeyRange.methodName()`.

In this example, we're only setting a lower boundary using the `lowerBound` method. Its value should be the lowest key value we want to retrieve. In this case, we'll use `0` since we want to retrieve all the records in our object store, starting with the first one:

```
objectstore.openCursor(IDBKeyRange.lowerBound(0));
```

If we wanted to set an upper limit instead, we could use the `upperBound` method. It also accepts one argument, which must be the highest key value we want to retrieve for this range:

```
objectstore.openCursor(IDBKeyRange.upperBound(1385265768757));
```

By default, `openCursor` sorts records by key in ascending order. We can change the direction of the cursor and its sorting direction, however, by adding a second argument. This second argument may be one of four values:

- `next`: puts the cursor at the beginning of the source, causing keys to be sorted in ascending order
- `prev`: short for "previous", it places the cursor at the end of the source, causing keys to be sorted in descending order
- `nextunique`: returns unique values sorted by key in ascending order
- `prevunique`: returns unique values sorted by key in descending order

To display these entries in descending order (newest entry first), change `objectstore.openCursor(IDBKeyRange.lowerBound(0))` to `objectstore.openCursor(IDBKeyRange.lowerBound(0), 'prev')`.

Retrieving or Deleting Individual Entries

But what if we wanted to retrieve just a single entry instead of our entire store? For that, we can use the `get` method:

```
var transaction, objectstore, request;

transaction    = databaseObj.transaction(['entries'],'readonly');
objectstore    = transaction.objectStore('entries');
request        = objectstore.get(key_of_entry_to_retrieve);

request.onsuccess = function(event){
    // display event.target.result
}
```

When this transaction successfully completes, we can do something with the result.

To delete a record, use the `delete` method. Its argument should also be the key of the object to delete:

```
var transaction, objectstore, request;

transaction = databaseObj.transaction(['entries'],'readwrite');
objectstore = trans.objectStore('entries');
request     = objectstore.delete(1384911901899);
```

```
request.onsuccess = function(event){
    // Update the user interface or take some other action.
}
```

Unlike retrieval operations, deletions are write operations. Use `readwrite` for this transaction type, instead of `readonly`.

Updating a Record

To update a record, we can once again use the `put` method. We just need to specify which entry we're updating. Since we've set a `keyPath`, we can pass an object with the same `entrydate` value to our `put` method:

```
var transaction, objectstore, request;

transaction = databaseObj.transaction(['entries'],'readwrite');
objectstore = trans.objectStore('entries');
request     = objectstore.put(
  {
    entrydate: 1384911901899,
    entry: 'Updated value for key 1384911901899.'
  }
);

request.onsuccess = function(event){
    // Update the user interface or take some other action.
}
```

The browser will update the value of the record whose key is `1384911901899`, as shown in Figure 4.3.

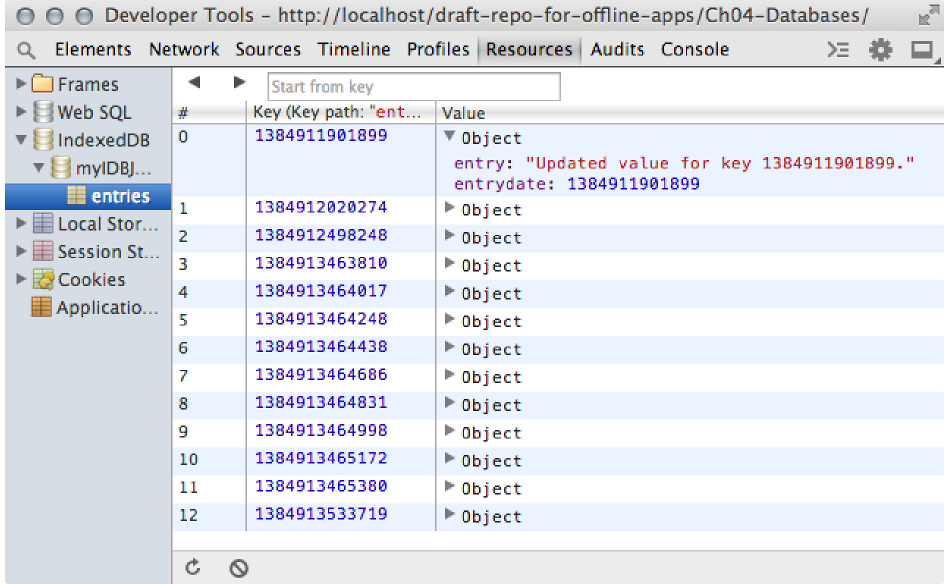

Figure 4.3. The updated value for our key

Updates, like additions, are write operations. When creating an update transaction, the second parameter of the `transaction` method must be `readwrite`.

Be careful when using `put` for updates. Updates replace the *entire* object value with the new one. You aren't simply overwriting individual object properties; you're actually replacing the whole record.

Deleting a Database

Deleting a database is just as easy as opening one. It doesn't require using a callback. Just call the `deleteDatabase` method of the IndexedDB database object:

```
idb.deleteDatabase('myIDBJournal');
```

It accepts one argument: the name of the database you wish to delete. Use it with caution.

We've covered enough to get you started with IndexedDB, but there's more to the API than can be covered in a single chapter. The Mozilla Developer Network[7] has

[7] https://developer.mozilla.org/en-US/docs/IndexedDB

the most robust documentation available to date, including coverage of older versions of the API.

Wrapping Up and Learning More

With offline web technologies, you can develop applications that work without an internet connection. We've covered the basics of offline applications in this book. If you'd like to learn more about any of these topics, a good place to start is with the W3C's WebPlatform.org.[8] It features tutorials and documentation on HTML5, CSS, JavaScript and web APIs.

Vendor-specific sites also have a wealth of information, particularly for documenting browser-specific quirks. Yet Google's HTML5Rocks.com[9] and the Mozilla Developer Network, mentioned above, are particularly good at documenting new technologies in their browsers while also addressing other vendors' implementations. The Microsoft Developer Network[10] includes a wealth of documentation about web technologies, especially as supported in Internet Explorer.

To track specifications, pay attention to the World Wide Web Consortium (W3C)[11] and the Web Hypertext Application Technology Working Group (WHATWG).[12] These bodies develop specifications and documentation for current and future APIs.

And of course, you can always visit Learnable.com[13] or SitePoint.com[14] to stay up to date on web technologies.

[8] http://webplatform.org/
[9] http://html5rocks.com/
[10] http://msdn.microsoft.com/
[11] http://w3.org/
[12] http://whatwg.org/
[13] https://learnable.com/
[14] http://www.sitepoint.com/