O'REILLY®

Client-Side Data Storage

KEEPING IT LOCAL

Raymond Camden

Security Tech EX2021
Walmart v. Security Tech
IPR2025-00809

O'REILLY®

Client-Side Data Storage

One of the most useful features of today's modern browsers is the ability to store data right on the user's computer or mobile device. Even as more people move toward the cloud, client-side storage can still save web developers a lot of time and money, if you do it right. This hands-on guide demonstrates several storage APIs in action. You'll learn how and when to use them, their plusses and minuses, and steps for implementing one or more of them in your application.

Ideal for experienced web developers familiar with JavaScript, this book also introduces several open source libraries that make storage APIs easier to work with.

- Learn how different browsers support each client-side storage API
- Work with web (aka local) storage for simple things like lists or preferences
- Use IndexedDB to store nearly anything you want on the user's browser
- Learn how to support web apps that still use the discontinued Web SQL Database API
- Explore Lockr, Dexie, and localForage, three libraries that simplify the use of storage APIs
- Build a simple working application that makes use of several storage techniques

"Client-side data is increasingly becoming a key part of any modern web application, whether it is targeting the desktop, mobile web, or mobile hybrid. Raymond Camden does an excellent job of covering the full range of options available to developers while also offering practical examples that make the subject both fun and practical."

—**Brian Rinaldi**
Developer Relations, Telerik

Raymond Camden, a developer advocate for IBM, focuses on the MobileFirst platform, hybrid mobile development, Node.js, HTML5, and web standards. He's a published author and speaks at conferences and user groups on a variety of topics. Raymond can be reached at his blog (http://www.raymondcamden.com/).

WEB DEVELOPMENT/DESIGN

US $29.99 **CAN $34.99**

ISBN: 978-1-491-93511-8

9 781491 935118 52999

Twitter: @oreillymedia
facebook.com/oreilly

Client-Side Data Storage

Keeping It Local

Raymond Camden

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Client-Side Data Storage
by Raymond Camden

Copyright © 2016 Raymond Camden. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Meg Foley
Production Editor: Nicholas Adams
Copyeditor: Rachel Monaghan
Proofreader: James Fraleigh
Indexer: Wendy Catalano
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

January 2016: First Edition

Revision History for the First Edition
2015-12-18: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491935118* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Client-Side Data Storage*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-93511-8

[LSI]

Table of Contents

Preface v

1. A Gentle Introduction to Client-Side Data Storage 1

2. Working with Cookies 3
- Cookies? Seriously? 3
- Working with Cookies 4
  - Reading Cookies 5
  - Deleting Cookies 6
- Demos 6
- Inspecting Cookies Within Developer Tools 10
- Support and Recommended Usage 11

3. Working with Web Storage 13
- Web Storage, AKA Local Storage 13
- Working with Web Storage 14
- Demos 15
- Listening for Storage Changes 19
- Inspecting Web Storage with Dev Tools 23
- Support and Recommended Usage 24

4. Working with IndexedDB 27
- Welcome to Deep Data 27
- Key IndexedDB Terms 28
- Checking for IndexedDB Support 29
- Working with Databases 29
- Working with Object Stores 31
  - Making Object Stores 32

Defining Primary Keys 34
Defining Indexes 36
Working with Data 37
Creating Data 38
Reading Data 42
Updating Data 45
Deleting Data 47
Getting All the Data 48
Working with Ranges and Indexes 51
Even More with IndexedDB 54
Storing Arrays 54
Counting Data 59
Inspecting IndexedDB with Dev Tools 59
Support and Recommended Usage 61

5. Working with Web SQL. 63
Dead Spec Walking 63
Basic Database Terms 64
Checking for Web SQL Support 64
Working with Databases 64
Working with Transactions 66
Inspecting Web SQL with Dev Tools 72
Support and Recommended Usage 73

6. Making It Easier with Libraries. 75
"Use the Library, Luke..." 75
Working with Lockr 75
Simplifying IndexedDB with Dexie 80
Working with localForage 88
More Options 90

7. Building a Sample Application. 91
Let's Build Something! 91
Our Sample Data 92
The Application 95
The Code 97
Wrap-up 103

Index. 105

Preface

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/cfjedimaster/DataStorageBook*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Client-Side Data Storage* by Raymond Camden (O'Reilly). Copyright 2016 Raymond Camden, 978-1-491-93511-8."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/client-side-data-storage*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at http://www.oreilly.com.

Find us on Facebook: http://facebook.com/oreilly

Follow us on Twitter: http://twitter.com/oreillymedia

Watch us on YouTube: http://www.youtube.com/oreillymedia

Acknowledgments

As always—I thank my wife. You inspire me. You keep me sane. You make me smile from ear to ear. I love you.

CHAPTER 1

A Gentle Introduction to Client-Side Data Storage

Browsers have—slowly and with a lot of growing pains—evolved over the past decade to become powerful workhorses. Enhanced layout controls, 3D graphics and gaming, and even music are now within the realm of possibility of the little old browser. One more exciting, although relatively less flashy (no pun intended), feature is client-side data storage. But what do we mean by this?

The "typical" process by which you navigate the Web hasn't changed since time began: your browser asks for a URL, a web server sends stuff back, you then ask for more stuff, and the web server sends more stuff to you.

You can certainly get more complex and add JavaScript and AJAX to the mix. But even in a fancy web 2.0 application, your browser may be requesting information from the server again and again and again. The reason for this is that—for all intents and purposes—the browser is an amnesiac. Everything it knows it has to learn from the server.

While this is true in general, it overlooks a powerful alternative—storing data on the browser itself. This enables the browser to skip asking the server for information and to simply retrieve it locally from the user's machine. It even affords the chance to manipulate that data for whatever purposes may make sense. That data could then be sent back to the server for updating later.

In summary, this gives the browser:

- Immediate access to data. Even with AJAX being typically much quicker for fetching data, by having the data local to the machine itself access will be even quicker.

- Less network traffic. Instead of constantly fetching data from the server, data can be fetched once and stored for as long as it makes sense. This leads to...
- Less strain on your server. If your server is constantly responding to requests and fetching stuff from a database server, you could overtax your server. By reducing the amount of calls you make, your server does less work.
- Finally, with data stored locally, the possibility of a fully offline application becomes much more feasible.

Of course, it isn't all sunshine and roses. Moving data to the browser can also have the following negatives:

- There isn't any support for synchronization. Imagine you've copied data from a server to a browser. How do you handle keeping that data in sync? What happens if there are conflicts? None of the core technologies covered in this book support any concept of handling sync. You can, however, find libraries like PouchDB, that have this built in.
- Storage limits are fuzzy. As developers, we hate fuzzy. We want to know exactly how much of a resource we can use. Unfortunately, for many of the technologies this book will cover, the limits—as well as what happens when you break them—are a bit vague.
- Finally, while the technologies covered here are powerful, they are not a replacement for a full database server. Database servers are extremely fine-tuned to the job of handling huge amounts of data and providing a way to search them. The solutions we'll cover are certainly capable of storing data, but at the same time, they aren't like an embedded Oracle server. (Although that may be a good thing.)

In this book, we'll discuss various client-side storage techniques. For each one we'll also cover, plainly and honestly, how well they are supported in the wild. You'll see examples of the APIs as well as demos you can use to help you learn how to use the APIs yourself. Finally, we'll look at a few libraries that aim to make client-side storage even easier. Buckle up—this is going to be a fun, and sometimes rough, tour through some of the more useful aspects of the Web!

CHAPTER 2

Working with Cookies

Cookies? Seriously?

I feel almost ashamed to be speaking about cookies in a modern web development book in 2015, but they are the oldest, and most stable, form of client-side storage available to developers today. They are certainly not the best method, and I'd almost never recommend using them, but they are an option and you may be forced to use (or modify) code that makes use of them at some point in the future.

Cookies were introduced in 1994 in a beta version of Netscape. They worked by using header values sent along with HTTP requests and responses. As you may know, whenever your browser requests a resource, a set of headers will be sent along with the request. Those headers include various types of data, including information about the browser and what form of data it wants. On the flip side, the server will also send headers back. Basically, every time you see a web page rendered in your browser, there was a set of headers that were also sent that you don't see. (You certainly *can* see them using browser tools. They aren't hidden as in "impossible to see," just hidden from view by default.)

Cookies are sent using HTTP headers, specifically the "Cookie" HTTP header, and are sent by the browser to the server and sent to the browser from the server. Right away you should see a problem with that. If one of the benefits of using client-side storage is that we don't have to send data over the wire, doesn't sending cookies back and forth negate that benefit? Absolutely. This is one more reason why I typically won't recommend using cookies.

By default, the browser makes no limit on the number of cookies it can have. In the past, there was a limit of 20 cookies per domain, but modern browsers seem to have removed that limit. (As an aside, I once set over 400 cookies in a Chrome browser and it worked just fine. However, my web server started throwing errors when

requests were made. So in this case, it was an issue of the web server having a limit, not the browser itself. But please don't use 400 cookies.) Research, or in other words, Googling, seems to indicate that a limit of 50 cookies per domain is safe at a total of 4 KB. That isn't a lot of space for cookie values, which hinders their real-world usage.

Cookies are unique per domain. This means that a cookie value set on *foo.com* will not be available on *goo.com*. This is good since you wouldn't want some other site to interfere with your usage on your own site. Cookies can be made to be unique in subdomains as well. So, for example, maybe *app.foo.com* is a unique subdomain of the Foo website. You can create cookies that are readable only on *app.foo.com* as well as cookies that would be available to *www.foo.com* and *app.foo.com*.

To make things even more complex, you can also create cookies that are valid only for a particular path. So, instead of *app.foo.com*, you may want to create cookies available just to *foo.com/app*.

Finally, you can create cookies that work only on the secure (https) version of your site. Obviously which of these options you use will depend on whatever your application is doing and where you think cookie values need to be present.

Along with where cookies are available, you can also specify how long they are available. Your options are:

- Cookies that last for the current session (basically until the browser is closed)
- Cookies that will last forever
- Cookies that live for a certain amount of time
- Cookies that expire after a particular time

Working with Cookies

Cookies do not have an API. To work with them, you simply access the `document.cookie` object in your code. So for example, to create a cookie, you could do this:

```
document.cookie = "nameOfCookie=value";
```

In the preceding example, I created a cookie called `nameOfCookie` and defined it as value. You use a "*name=value*" format to define the name and value all at once. Here is a real example:

```
document.cookie = "name=Raymond";
```

In the previous example, I simply set a cookie called `name` to the value `Raymond`. Values must be URL-safe, which means if you are doing anything dynamic, you'll want to use a helper function like `encodeURIComponent`.

```
name = "Raymond Camden";
document.cookie = "name=" + encodeURIComponent(name);
```

So far, so good. But here's where things get a bit wonky. You may be wondering how you would set multiple cookies. If you literally set `document.cookie` multiple times, it just works. So, for example:

```
document.cookie = "name=Raymond";
document.cookie = "age=43";
```

In this code sample, we've actually created *two* cookies, not one. That just feels plain wrong to me, but you'll have to just roll with it.

So that's how you create a cookie with a value, but what about all the metadata I mentioned—like defining where a cookie is available and how long it lasts? The format for appending metadata in a cookie is to use a semicolon after the value. Here is an example:

```
document.cookie = "name=Raymond; expires=Fri, 31 Dec 9999 23:59:59 GMT";
```

This example specifies when the cookie expires. We can further expand it by specifying that it works only on a subdomain:

```
document.cookie = "name=Raymond; expires=Fri, 31 Dec 9999 23:59:59 GMT;
domain=app.foo.com";
```

You get the idea. When you don't specify metadata like this, cookies will default to the current domain, the current path (probably not what you want), and an expiration in the current session.

Reading Cookies

Reading cookies is somewhat easier—depending on how comfortable you feel about string parsing. There is no API to get "a" cookie. Instead, you can simply read `document.cookie`. Doing so will give you all the cookies set for a particular site. Here is the `document.cookie` value from CNN:

```
"_cb_ls=1;
_chartbeat2=Dlxk2YDHxyg1BXCry6.1426601000831.1439508384927.0000000000000001;
Akamai_AnalyticsMetrics_clientId=89E881222E0BD593DF2468758F328F689C36BAC1;
octowebstatid=16ppgnhrso5f2frjuvq5; ug=55cd27810eb00b0a3c6ac33c7d05339d; ugs=1;
__CG=u%3A2449373858398994400%2Cs%3A72001958%2Ct%3A1439508379253%2Cc%3A1%2Ck%3
Awww.cnn.com/19/19/54%2Cf%3A0%2Ci%3A0; __CT_Data=gpv=10;
__gads=ID=3d001e8bba3c7c6d:T=1426601001:S=ALNI_MYWNYv1SRt0tx7LQ2AzdSESOBygNA;
__vrf=1439508379290061VnNeHVWPiIjkcWMeUjRWpppwsPktE;
grvinsights=a5a942f8e7c604d573496053d63f590c; optimizelyBuckets=%7B%7D;
optimizelyEndUserId=oeu1426600996913r0.5135214943438768;
optimizelySegments=%7B%22170962340%22%3A%22false%22%2C%22171657961%22%3A%22
safari%22%2C%22172148679%22%3A%22none%22%2C%22172265329%22%3A%22direct%22%7D;
RT=sl=1&ss=1439508375405&tt=9387&obo=0&bcn=%2F%2F36f11e49.mpstat.us%2F&sh=1439
508384794%3D1%3A0%3A9387&dm=cnn.com&si=5be398d8-bb51-42ea-8128-6d4251e47ada;
```

```
 s_cc=true;s_fid=5324AC5D0F8323AB-3F26DEB602CBB276; s_ppv=13; s_sq=%5B
%5BB%5D%5D;s_vi=[CS]v1|2A841A13051D0B97-400001280000490C[CE]; tosAgreed=true"
```

If that seems like a mess, you're absolutely correct. Reading an individual cookie would mean parsing the string into components separated by a semicolon. Also note that you do not have access to any metadata. There is no way to get this information via the `document.cookie` value. It wouldn't be too difficult to parse the string, but you really don't need to do that. Toward the end of this chapter I'll show you a great little library that makes working with cookies easier.

Deleting Cookies

To delete a cookie, you can simply set a cookie with an expiration in the past:

```
document.cookie = "name=Raymond; expires=Thu, 01 Jan 1970 00:00:00 GMT";
```

Technically the value doesn't matter, but the name must match the name of the cookie you want to delete.

Demos

Now that you've seen the basics of working with cookies, let's look at a simple demo. As I mentioned, you probably don't want to build cookie parsing code yourself. Instead, you should use one of the many libraries out there. For our demo, we'll use a simple (and excellent) free library from the Mozilla Developer Network (MDN). You can find this code (along with even more information on cookies) at *https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie*. The library has methods for:

`getItem`
: Gets a cookie

`setItem`
: Sets a cookie (including expiration, path, domain, etc.)

`removeItem`
: Deletes a cookie

`hasItem`
: Checks if a cookie exists

`keys`
: Returns the names of all the cookies

The MDN code has been saved in a file called *cookies.js*. This is available along with the rest of the code for this book. Our first example, *test1.html*, will simply use a cookie to count how many times you've visited the site (Example 2-1).

As a quick aside, all examples in this book will assume (and require) that you run them from a local web server and not just by opening the file in your browser. If you don't have a local web server installed, consider a simple tool like httpster for a quick way to set up a development web server. (But promise me you'll install a proper web server later. You are a web developer, right?)

Example 2-1. test1.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cookie Test One</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script type="text/javascript" src="cookies.js"></script>
</head>

<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        //initial value
        var visited = 0;

        //Check to see if we have the cookie...
        if(docCookies.hasItem("visited")) {
            //and get it
            visited = docCookies.getItem("visited");
        }

        visited++;

        //update
        docCookies.setItem("visited", visited);

        $("#resultDiv").text("You have visited this site "+ visited +
             " times.");

    });
    </script>

</body>
</html>
```

Starting from the top, you can see a fairly typical `document.ready` block from the friendly jQuery library. I begin by setting an initial value for a variable called `visited`. If the cookie library says I have a cookie called `visited`, I then update the variable with the cookie's value. I then increment this value by one, store it back, and then display it in the browser. By default, this cookie will exist only for the current session, so let's improve it in *test2.html* (Example 2-2).

Example 2-2. test2.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cookie Test Two</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script type="text/javascript" src="cookies.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        //initial value
        var visited = 0;

        //Check to see if we have the cookie...
        if(docCookies.hasItem("visited2")) {
            //and get it
            visited = docCookies.getItem("visited2");
        }

        visited++;

        //update
        docCookies.setItem("visited2", visited, Infinity);

        $("#resultDiv").text("You have visited this site "+ visited +
            " times.");

    });
    </script>

</body>
</html>
```

In this version, we've made two changes. First, our cookie is now `visited2`. Normally I'd use the same name as before, but we want to differentiate between the two cookies while testing. The second change was to modify the `setItem` call. We've used an expiration value of `Infinity`. This creates a cookie that will last forever. Now that page will accurately reflect how many times it has been visited by a particular user regardless of whether the user is visiting during the same browser session.

So far our demos have been pretty simple, so let's take it up a notch. We can use cookies to remember when a user has last visited the site. Based on the user's last visit, we can either:

- Greet a new user who has never been here before.
- Provide a special message to someone who hasn't been here in a while—for example, mentioning cool new features.
- Simply welcome a user who has visited often.

Let's look at the code (Example 2-3) and then I'll walk you through how it works.

Example 2-3. test3.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Cookie Test Three</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script type="text/javascript" src="cookies.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        var $resultDiv = $("#resultDiv");

        //Is this a brand new user?
        var newUser = true;
        //How many days since last visit
        var daysSinceLastVisit;

        //Check to see if we have the cookie...
        if(docCookies.hasItem("lastVisit")) {
            newUser = false;
```

```
            //do some math to find out how long it has been
            var lastVisit = docCookies.getItem("lastVisit");
            var now = new Date();
            var lastVisitDate = new Date(lastVisit);
            //credit: http://stackoverflow.com/a/3224854/52160
            var timeDiff = Math.abs(now.getTime() - lastVisitDate.getTime());
            var daysSinceLastVisit = Math.ceil(timeDiff / (1000 * 3600 * 24));
        }

        //Set to now
        docCookies.setItem("lastVisit", new Date(), Infinity);

        if(newUser) {
            $resultDiv.text("Welcome to the site!");
        } else if(daysSinceLastVisit > 20) {
            $resultDiv.text("Welcome back to the site!");
        } else {
            $resultDiv.text("Welcome good user!");
        }

    });
    </script>

</body>
</html>
```

We begin by setting up two variables, `newUser` and `daysSinceLastVisit`. The first will simply be a Boolean we can check to determine if the user is brand new, while the latter will report on how many days it has been since the user's last visit.

If a cookie, `lastVisit`, exists, we get it and then create a `Date` variable out of the value. From that we can then use some simple math to figure out how many days it has been since the user's last visit.

Next, we set the cookie value for `lastVisit` to the current time.

That's the core of the logic; we then simply spit out a message based on one of the three states just mentioned. In this demo, the value `20` is used to determine that it has been too long since the user visited. Obviously this value is arbitrary.

Inspecting Cookies Within Developer Tools

Modern browsers provide excellent developer tools that make it easier to inspect and check your use of cookies. In Firefox, you can see cookies if you enable the Storage tab (it may not be visible by default). Once it's enabled, you can view your current cookies (see Figure 2-1). Firefox does not allow you to modify cookies, just view them.

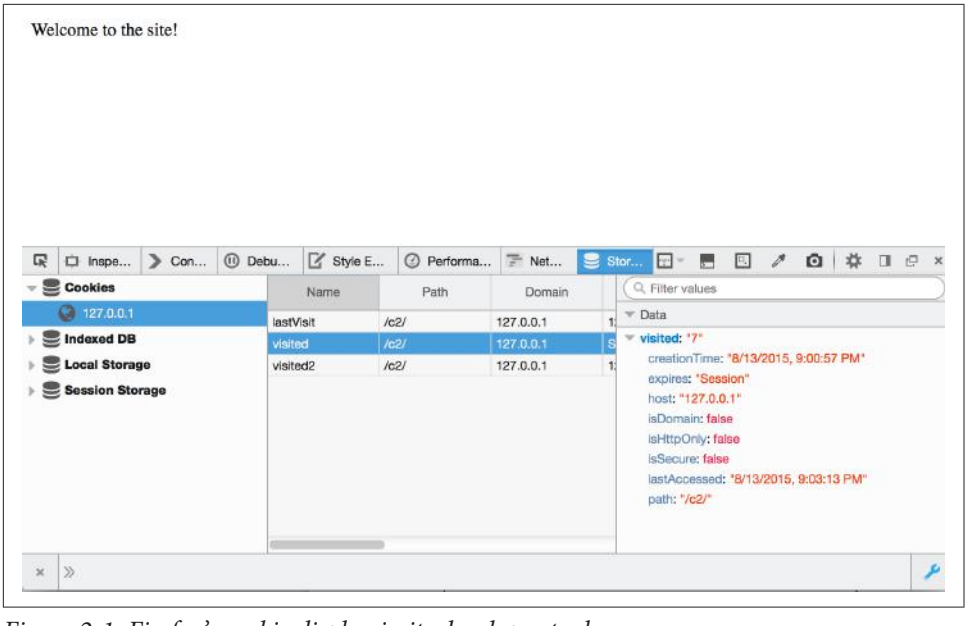

Figure 2-1. Firefox's cookie display in its developer tools

Chrome will show you a site's current cookies in the Resources tab (Figure 2-2). You can delete cookies here, but not edit them.

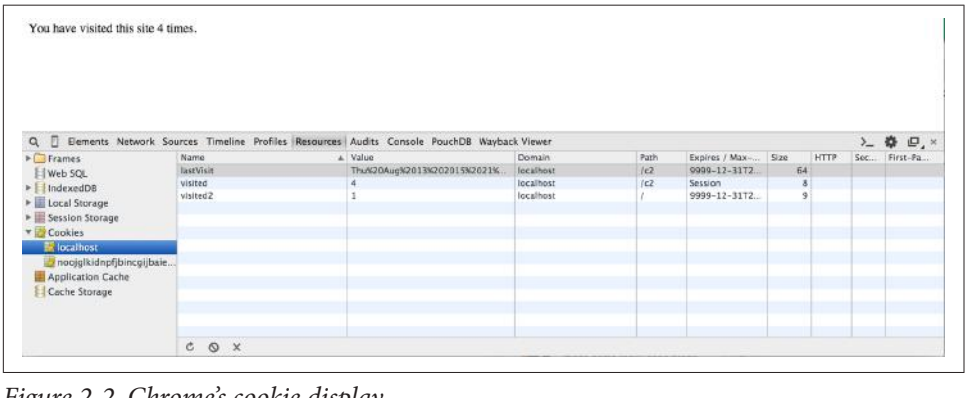

Figure 2-2. Chrome's cookie display

Support and Recommended Usage

CanIUse.com, the best resource for checking browser feature support, doesn't even report on cookies because support is and has been 100% for a long time. However, just because a browser supports the feature doesn't guarantee that it will work. Many people have developed a fear of cookies and blocked them.

As for recommended uses, as I said in the beginning, my recommendation is to *not* use cookies, but if you must, keep it simple. You can use them for user preferences and basic information (name, age, etc.). Here is a practical example. I use WordPress for my blog. Whenever I upload an image, WordPress asks if I want to include a link to the image when it's added to the blog entry. I nearly always change this particular form field to say that I want no link. A cookie could be used to remember this default so I don't have to constantly modify this when writing. That's a trivial example, but it is something I run into nearly every day. Finally, if there is something you think the server should know as well, then using cookies ensures the server will see the same values.

CHAPTER 3

Working with Web Storage

Web Storage, AKA Local Storage

Web storage is the formal name for a technology most of us call Local Storage. Out of all the client-side technologies we'll discuss in this book, the Web Storage API is probably the quickest to learn and simplest to pick up. It is an API focused on setting and retrieving simple values by keys—so, for example, storing the value `Ray` for the key `name`. Or storing the value `43` for the key `age`. Complex data (like arrays or objects) is not supported, but you can store it by encoding the values into JSON first. (And obviously you'll need to decode them on retrieval.)

Web storage comes in two flavors: Local Storage and Session Storage. Both use the exact same API, but while Local Storage will persist forever (or until the user clears it), Session Storage will go away as soon as the browser is closed. Since most people use the persistent version, most developers use (and talk about) Local Storage. The official specification for the Web Storage API can be found at *http://www.w3.org/TR/webstorage*.

Just like cookies (and every other technology covered in this book), web storage is unique to a particular domain. Unlike cookies, there is no magical way to make data stored at *www.foo.com* also available at *app.foo.com*. (There are fancy workarounds with iframes, but let's avoid that for now.) Basically this all means that using a web storage key called `name` is completely safe for both *foo.com* and *goo.com*—they won't conflict.

Your limits for web storage are a bit variable, but in general they range from 5 to 10 MB. Typically, this shouldn't be a problem unless you're storing large packets of data, which is (generally, not always) something that isn't recommended. If you go over the limit, Chrome, Firefox, and Safari will all give you an error you can handle in your

code. Internet Explorer 11 and (at the time of writing) Edge do nothing, unfortunately.

Working with Web Storage

The Web Storage API (and for this chapter, we'll use Local Storage for all our demos, but remember that the Session Storage version acts the *exact* same way) has four simple methods:

`localStorage.setItem`
: Sets a value for a particular key

`localStorage.getItem`
: Retrieves a value for a particular key

`localStorage.removeItem`
: Deletes a key and the value associated with it

`localStorage.clear`
: Removes all key/value pairs (but just for the specific domain making the request)

While Web Storage has an API, you can also treat the data like a simple JavaScript object. So, for example, this statement:

```
localStorage.setItem["something"] = 1
```

will write to Web Storage, and this:

```
console.log(localStorage["something"]);
```

would read from it. While this works, I generally recommend using the API methods for consistency's sake.

One thing you must be very careful about is in regards to what data you store in Web Storage. Web Storage supports only string data. This can be confusing at times. Imagine this code snippet:

```
var names = ["Ray", "Jeanne"];
localStorage.setItem("names", names);
```

This code will run just fine; however, it will store the string version of the array instead of the array itself. This means that if you then do `localStorage.getItem("names")`, you'll have `"Ray,Jeanne"`—a string, not the array you intended.

Luckily there is a pretty simple workaround: JSON encoding. By converting your complex data into JSON, and then decoding it back when you fetch the value, you can store complex data easily in Web Storage. Here's a modified version of the previous snippet that uses the JSON object available in modern browsers. (For older browsers you can find plenty of libraries that will add this support.)

```
var names = ["Ray", "Jeanne"];
localStorage.setItem("names", JSON.stringify(names));
```

Reading the value back into an array is pretty simple as well:

```
var storedNames = JSON.parse(localStorarge.getItem("names"));
```

In order for this to work, you'll need to remember what keys are storing what types of values, so be sure to keep track of this. You could use a naming system where all keys prefixed with `js` or `json` imply that the value is JSON-encoded.

Now that you've seen how simple the API is, let's look at a few demos.

Demos

For our first demo, we're going to do something simple, and a bit cheesy. We'll use Web Storage to track how many times you've visited the page (Example 3-1). In the previous chapter we told you to use a proper web server for testing, and we'll remind you of this one more time. Do *not* simply open this file by double-clicking on it. Run it in a local web server instead.

Example 3-1. test1.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Test One</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>
<body>

    <div id="resultDiv"></div>

    <script>
    $(document).ready(function() {

        if(window.localStorage) {
            var numHits = localStorage.getItem("numHits");
            if(!numHits) numHits=0;
            else numHits = parseInt(numHits, 10);
            numHits++;
            localStorage.setItem("numHits",numHits);
            $("#resultDiv").text("You have visited this site " +
                numHits +" times.");
        }
```

```
    });
    </script>

</body>
</html>
```

The example code has one simple `div` block that we'll use to render the number of times you've visited the site. The JavaScript code begins by checking that `window.localStorage` exists. While Web Storage has *very* good support (as you'll see toward the end of the chapter), it takes very little code to check and ensure it is supported. Next, we fetch a value from a key called `numHits`. If we get nothing back, we then default the value to `0`; otherwise, we use `parseInt` to turn the string value into a proper number. Remember, Web Storage stores everything as strings, even numbers.

Next we simply increment the value, store it back into Web Storage, and then render out the result to screen (see Figure 3-1). We could have been fancy and supported "1 time" versus "1 times," but that's overkill for our purposes.

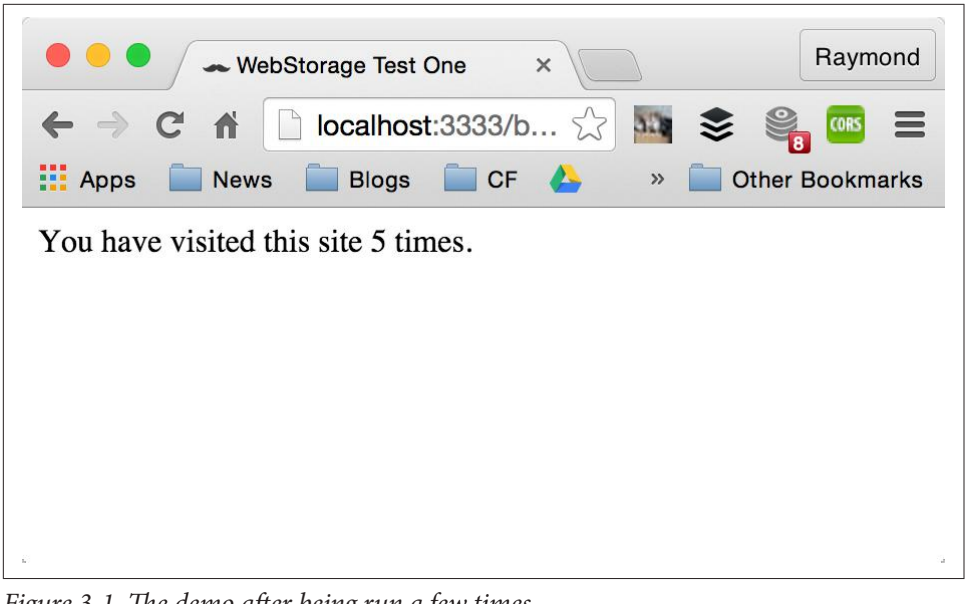

Figure 3-1. The demo after being run a few times

As we've said before, we won't be covering the Session Storage version of the API because it is no different, but included with the code for this book is a file called *test1_session.html*. It is a modified version of Example 3-1 that simply demonstrates the use of the `sessionStorage` object instead of `localStorage`.

Now let's kick it up a notch. Our next demo is actually something useful. Have you ever worked on a form and then accidentally closed the browser tab? Or perhaps the

form was on a site that required login information and your login expired before you could complete the form? In Example 3-2, we're going to use Web Storage to keep a copy of the form data around so it isn't lost. We also need to remove the form data when the form is actually completed.

Example 3-2. test2.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Test Two</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>
<body>


    <form id="myForm">

        <p>
        Your Name:
        <input type="text" id="name" name="name">
        </p>

        <p>
        Your Age:
        <input type="number" id="age" name="age">
        </p>

        <p>
        Your Email:
        <input type="email" id="email" name="email">
        </p>

        <p>
            <input type="submit">
        </p>

    </form>

    <script>
    $(document).ready(function() {

        if(window.localStorage) {

            //If I have data, fetch it and preset
            if(localStorage.getItem("personForm")) {
```

```
            var person =
            JSON.parse(localStorage.getItem("personForm"));
            $("#name").val(person.name);
            $("#age").val(person.age);
            $("#email").val(person.email);
            console.log("restored from storage");
        }

        //Listen for all <input> fields and their input event
        $("input").on("input", function(e) {
            var name = $("#name").val();
            var age = $("#age").val();
            var email = $("#email").val();
            var person = {"name":name, "age":age, "email":email};
            localStorage.setItem("personForm",
                JSON.stringify(person));
            console.log("stored the form...");
        });

        //form handler should clear storage
        $("#myForm").on("submit",function(e) {
            localStorage.removeItem("personForm");
            return true;
        });
    }

});
</script>

</body>
</html>
```

The top portion of the file is just the form we're going to persist. It has three input fields and a submit button. (Note it has no action value for the `<form>` tag, as we aren't really building a form processor here.)

In the JavaScript portion, we once again ensure the browser supports Web Storage and then get down to business.

The first thing we do is see if data exists for the form in Web Storage. We're calling the key `personForm` and if it exists, it is a JSON-encoded object. Once we get it and decode it, we can then update each of our three form fields with the existing data. Since they are simple text fields this isn't difficult, but obviously you could support select, checkbox, and radio fields with a bit more work.

Next we add an event listener to the input fields in the form. Every time the `input` event is fired on them, it means something has changed. We fetch the values, store them in a simple object, and then store a JSON-encoded version into Web Storage.

Finally, when the user submits the form (Figure 3-2), we don't need to keep a copy of it around anymore. (Although technically, there may be fields you wish to keep around if this is a form people use often.) We simply use `removeItem` to delete the key from storage.

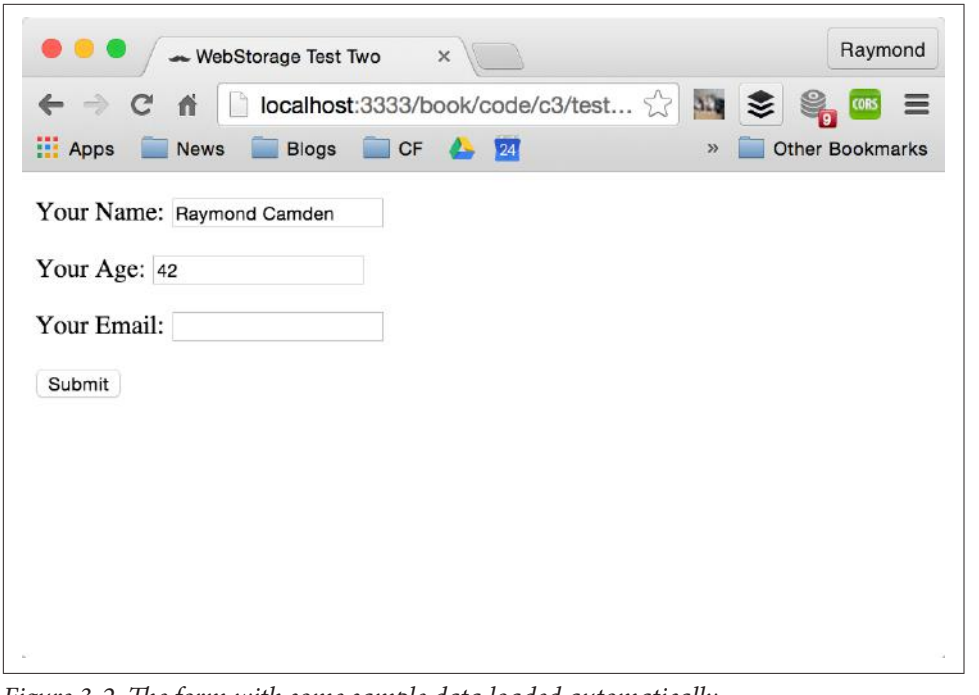

Figure 3-2. The form with some sample data loaded automatically

Listening for Storage Changes

The last feature we'll discuss is the `storage` event. This one is a bit odd and probably not something you'll need to worry about, but it definitely bears covering. The `storage` event is exactly what it sounds like—an event that is thrown when storage (either Local Storage or Session Storage) is modified. Let's look at a simple case (Example 3-3).

Example 3-3. test3.html

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>WebStorage Event Test</title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
```

```
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>
<body>

    <form id="myForm">

        <p>
        Test Value:
        <input type="text" id="test">
        </p>

    </form>

    <script>
    $(document).ready(function() {

        if(window.localStorage) {

            if(localStorage.getItem("testValue")) {
                $("#test").val(localStorage.getItem("testValue"));
            }

            //Listen for all <input> fields and their input event
            $("input").on("input", function(e) {
                var test = $("#test").val();
                localStorage.setItem("testValue", test);
                console.log("stored the test value.");
            });

            $(window).on("storage", function(e) {
                console.log("storage event fired");
                console.dir(e);
            });

        }

    });
    </script>

</body>
</html>
```

On the top of the page you see a simple form with one field, a text field with the ID `name`. In the JavaScript, you can see code that acts a bit like the previous demo. On load, we look for a previously existing value and set it to the field. There is then a generic input change handler to notice changes to fields and store them. Note the use of the `console.log` method to record the save.

Beneath this we have a new event listener. The `storage` event is fired on the window object, so we listen there. We'll talk a bit more about what's in this event in a moment. For now, though, go ahead and open the file and do some testing. Simply type something in and change it a few times. You may get something like Figure 3-3.

Figure 3-3. Where's the storage event?

Notice something odd there? There are multiple log messages about storing the value, but nothing about the `storage` event itself! What's going on here?

Well, it turns out that the `storage` event is fired only when another instance of the browser modifies storage. How can that happen? Simply open another tab and enter the same URL for the demo. Modify the value in *that* tab and then return to the *original* tab, and you'll finally see it. What's happening here is that the event is letting you know that other code has modified storage.

In Figure 3-4, note that the event contains two interesting values, `oldValue` and `newValue`. As you can guess, the event is reporting on what the original value was and what it changed to.

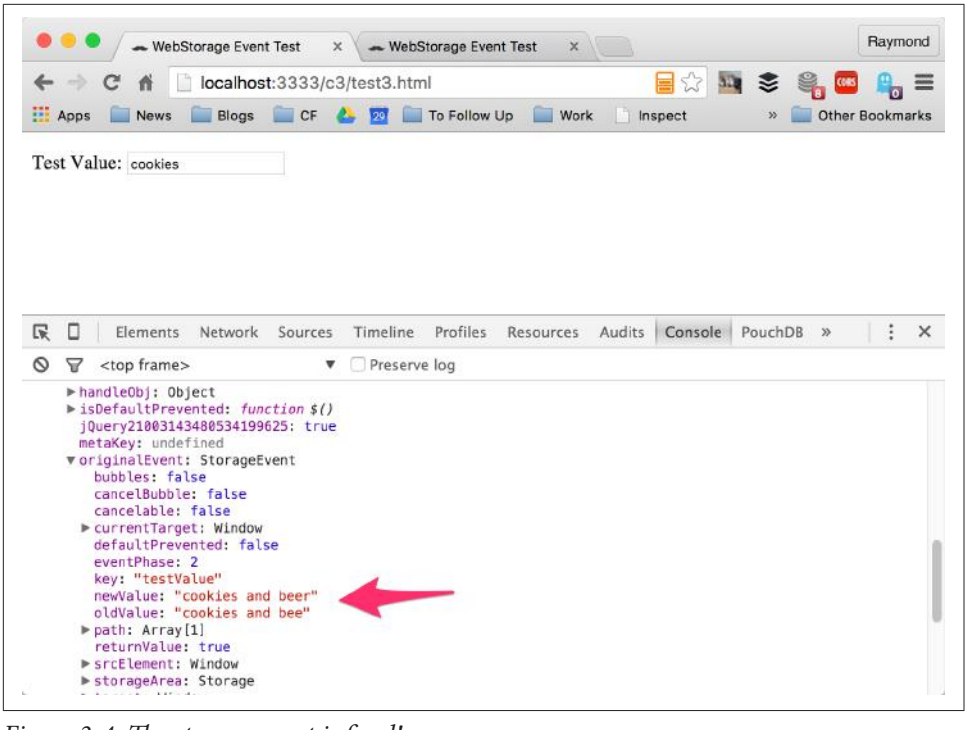

Figure 3-4. The storage event is fired!

The question now is, what do you do? Actually handling the change is up to your application. You may want to prompt the user, asking whether they want to accept the change or keep their current version. To be clear, it is already too late. The storage system changed. But you can take the value in the form and update storage. Of course, the *other* tab then would get the same warning. Prompting the user is probably a bad idea. A simpler solution may be to simply update the form field to represent the latest value. Example 3-4 demonstrates this approach. (Since we're changing only the storage event, the listing just contains that bit of code.)

Example 3-4. test4.html (snippet)

```
$(window).on("storage", function(e) {
    console.log("storage event fired");
    $("#test").val(e.originalEvent.newValue);
});
```

As you can see, we're simply changing the form field to the new value. We could have also used `localStorage.getItem("testValue")`, but since we already had the value in the event, it made sense to use it.

Inspecting Web Storage with Dev Tools

Both Firefox and Chrome provide really great support for working with Web Storage in their respective browser developer tools. In Figure 3-5, you can see Firefox's rendering of Local Storage data from the demos.

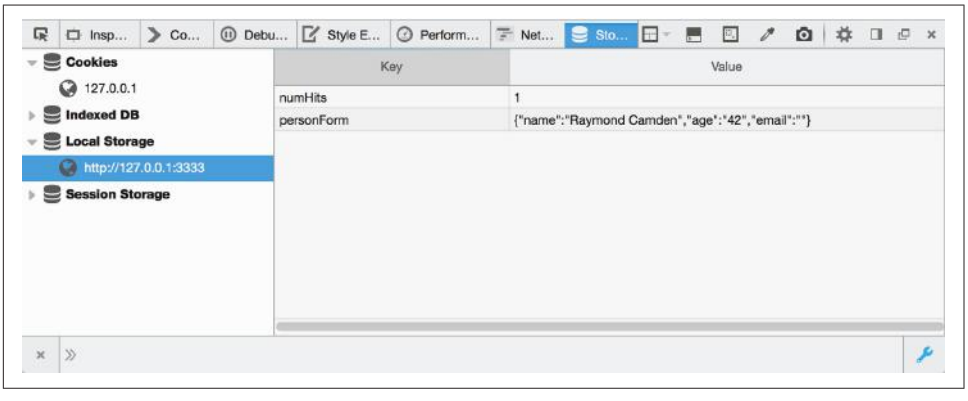

Figure 3-5. Firefox's Dev Tools view of Web Storage

It may not be immediately obvious, but you can click on a value for a detailed view. For the JSON string in `personForm`, Firefox recognizes it as JSON and helpfully displays it a bit more nicely (Figure 3-6).

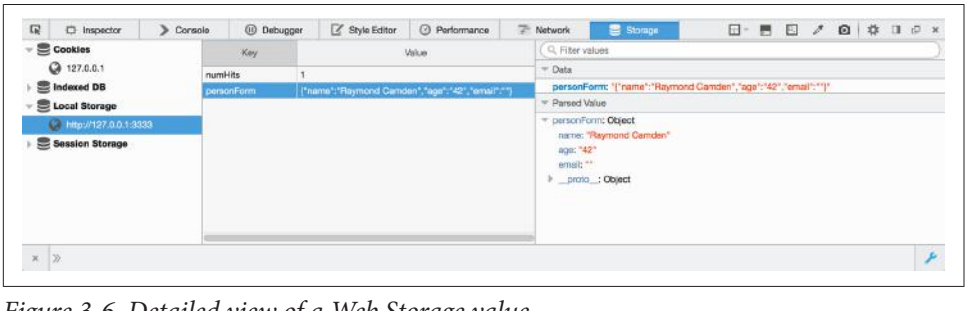

Figure 3-6. Detailed view of a Web Storage value

In Firefox, you cannot edit or delete Web Storage values. Don't forget, though, that you can use the Console tab to directly manipulate values.

In Chrome, you can find Web Storage values under the Resources tab (Figure 3-7).

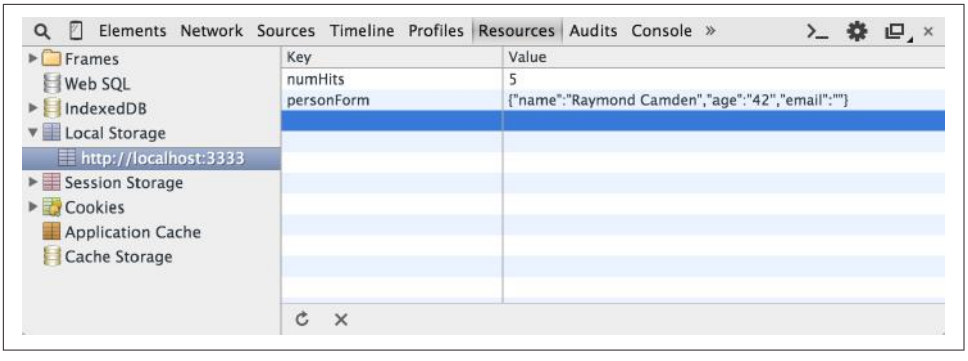

Figure 3-7. Chrome's Dev Tools for Web Storage

Unlike with Firefox, if you double-click on a value you can edit it by hand. You can also use the X icon at the bottom to remove an individual record.

Support and Recommended Usage

So, how well is Web Storage supported? Figure 3-8 shows the answer from CanIUse.com.

Figure 3-8. CanIUse.com

That's *really* good support, and as you saw in the demos, it is fairly easy to check for support and enhance pages with this feature. In both demos, if the browser didn't support Web Storage, nothing broke, which is how we should be building web pages in general!

As for recommended uses, I'd consider simple things like preferences to be a good example, as well as basic information like the user's name, age, and so on, and maybe a list of "favorited" items from your site. These are things that could be stored server-side too, of course, but as they are user-specific and not crucial to the site itself, they may be more appropriately stored on the client.

CHAPTER 4

Working with IndexedDB

Welcome to Deep Data

So far the options we've worked with for storing data on the client side have been relatively simple and relatively small in nature. Now it's time to dig deep and work with a large-scale storage system, IndexedDB. IndexedDB is a powerful storage system with a great deal of flexibility. You can store just about anything and everything you want to on the user's browser. However, with that great power and flexibility comes an API that isn't quite as friendly as Web Storage. You'll also find that IndexedDB does not quite yet have great support on mobile browsers, and even when it does, it can be poorly implemented. (iOS 8, in particular, has such bad support for IndexedDB that it is simply better that you pretend it doesn't exist.) However, in the future, IndexedDB will probably become the standard method of storing large amounts of data on the client side. For more information, and an exciting read (honest!), check out the specification at *http://www.w3.org/TR/IndexedDB/*.

Like every other client-side storage system described so far, IndexedDB is unique to a domain. Limits are usually poorly defined but tend to be extremely large when they exist. In general, there are no limits, but the browser will begin clearing out other IndexedDB instances if space begins to get low. Like most "persistent" systems, anything stored in the browser is inherently *not* persistent over eternity, but the benefits of storing data, even only semi-persistently, are worth the effort.

Key IndexedDB Terms

Before we get into the code, let's cover some important IndexedDB terms.

Databases

At the highest level of IndexedDB is the concept of a database. If you've ever worked with databases in server-side web applications, then you're already familiar with this concept. Basically, a database is where you put your data. As the developer of your site, you have the option of making any number of databases you want, but typically, you will create only one database for your site's needs. There's no hard and fast rule here, but in general, one database per site or web application makes the most sense.

Object stores

An object store is an individual bucket to hold data. If you've worked with traditional relational databases, then you can think of an object store as a table. Basically, if you have one database for your web application, then you will have one object store for each *type* of data you're storing. Given a website that persists documentation articles and user-generated notes, then you could imagine two object stores. Unlike with relational database tables, you do not have a rigid column structure that dictates how data is stored. So, for example, in a MySQL database table called "person," you could have two character columns for first and last name and a numeric column for age. In IndexedDB, what you can store can be more loose. I can store a person with a first and last name but an age value of `unknown` or even `too old to matter`. This can coexist with another person stored with a proper age. IndexedDB is much more flexible in letting you store data. That's both good and bad. Just because you can "mix it up" doesn't necessarily mean that you should!

Indexes

This is where the "Indexed" of "IndexedDB" comes in. An index is a way of retrieving data from your object store. You can always get all of the data from an object store, but many times you want to get data by a particular property. So, for example, if you are storing people, then you may want to fetch them later by their name, or their Social Security number, or perhaps their gender. By using indexes, you're telling the IndexedDB system to make it easier to fetch data by those properties later.

As we go on you'll learn a few other important IndexedDB terms, but these three cover the main ones you'll encounter throughout your development.

Checking for IndexedDB Support

Because IndexedDB still isn't widely supported, it is important that you check for its support before actually using it. The simplest way of doing so is with a check of the `window` object.

```
if("indexedDB" in window) {
}
```

You could write this as a function too, of course:

```
function idbOK() {
    return "indexedDB" in window;
}
```

Due to the serious issues with IndexedDB and iOS 8, you may wish to consider modifying the code to return `false` on those platforms. This StackOverflow answer demonstrates a simple regex text that could be used:

```
function idbOK() {
    return "indexedDB" in window &&
    !/iPad|iPhone|iPod/.test(navigator.platform);
}
```

Working with Databases

As I've stated, the database is the top-level container for your data. How many databases you have, what you name them, and so forth is completely up to you. When creating a database, you provide a name and a version, typically starting at 1. The version number is both arbitrary and important. You can only modify your database structure (and to be clear, this means the object stores and indexes, not the actual data itself) when you change versions. This means if you have a web app out in the wild and need to store some new *type* of data, then you'll need to increment your version to a new number.

Everything you do in IndexedDB is asynchronous, so opening a database means you'll need to respond to an event in order to begin working with it. The events you get from a database open operation are `success`, `error`, `upgradeneeded`, and `blocked`.

The first two are self-explanatory, but what do the others mean? `upgradeneeded` is used when your database is first accessed by a user or when the version number has changed. This is where you will set up the structure of your data. `blocked` is used when the database isn't available at all and cannot be used. Example 4-1 demonstrates a simple case of opening a database. We aren't actually doing anything with it—just attempting to open it.

Example 4-1. test_1_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb1",1);

    openRequest.onupgradeneeded = function(e) {
        console.log("running onupgradeneeded");
    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

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

You can see that the code begins by checking if IndexedDB is supported at all. If it is, the `indexedDB.open` method is used to open the database. The first argument is the name. Since IndexedDB is private to an individual site, you don't have to worry about your name conflicting with another database. The second argument is the version. Again, you can use any number here, but you should start with 1.

The result of this call is a request object that you can use to attach event listeners to. In the code here there is an event listener for all the events except `blocked`. The first time you run this code (assuming you're using an IndexedDB-capable browser), you would see the output shown in Figure 4-1 in the console.

```
10:43:27.006    running onupgradeneeded
10:43:27.008    running onsuccess
```

Figure 4-1. Notice the events being run

Since this was the first time you used the database, an `upgradeneeded` event is fired. This also represents the fact that the database itself was created. If you repeat this process, only the success event will be fired (see Figure 4-2).

```
10:45:18.190    running onsuccess
```

Figure 4-2. Since the database already existed and the version didn't change, only one event is fired

That's the basics of working with the database; now let's get deeper with object stores.

Working with Object Stores

As we said earlier, an IndexedDB object store is somewhat similar to an SQL database table. It should contain data of one "type"—for example, instances of "people" records or "notes" or something else. The idea is that you will have one object store for each type of data you need to persist.

Object stores can *only* be created during the `upgradeneeded` event. This is why the version number matters. Let's say you design your database to support two object stores. A few months down the road, you decide you need to store a third type of data. You will need to do two things: first, change the version, and second, write the code to add the new object store.

In pseudocode, you can think of this process like so:

```
I request to open the database
If the request fired an upgrade needed event, create object stores
If the request fired a success event, I'm ready to roll
```

Making Object Stores

To create an object store, you should first check to see if it exists already. Using a database variable (which you will get from the event handlers associated with opening the database), you can access the property `objectStoreNames`. This property is a `DOMStringList` that will let you inspect it for an existing value. If it doesn't exist, you can then create it using the method call `createObjectStore("name", options)`. Let's look at Example 4-2.

Example 4-2. test_2_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb2",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");
        if(!thisDB.objectStoreNames.contains("firstOS")) {
            console.log("makng a new object store");
            thisDB.createObjectStore("firstOS");
        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;
        console.dir(db.objectStoreNames);
    }
```

```
    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

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

After checking to ensure IndexedDB is supported, we open the database. (Note we are using a different name from the last example.) If `upgradeneeded` is fired, it means that either the user is visiting the page for the first time or had an earlier version of the database.

We fetch the database object itself by getting the result of the event's target object. The `objectStoreNames DOMStringList` value lets us use `contains` to see if the name of our object store exists. If it does not, then we create it. Note that we pass only the name of the object store. The `createObjectStore` method also lets us pass a second argument with options. This is how we'll define various configuration properties for the object store including indexes.

As before, the first time you run this, the `upgradeneeded` event will fire. This time it will actually do something (see Figure 4-3).

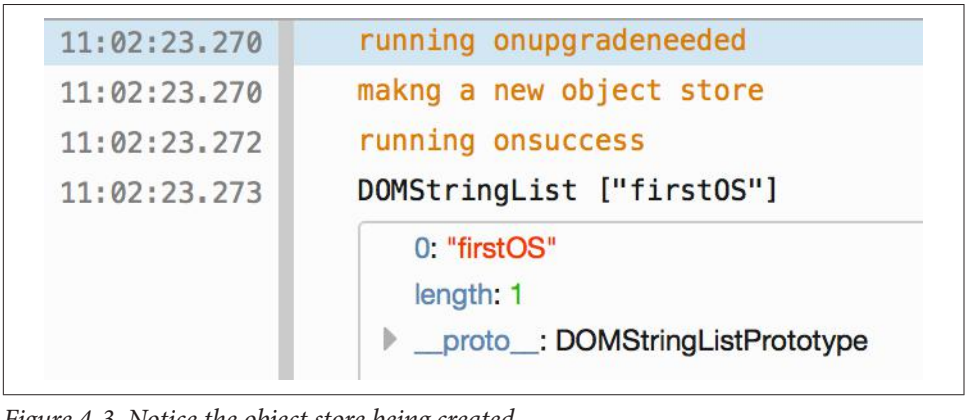

Figure 4-3. Notice the object store being created

On the next request, only the `success` handler is run, but our object store still exists (see Figure 4-4).

```
11:03:44.158    running onsuccess
11:03:44.159    DOMStringList ["firstOS"]
                  0: "firstOS"
                  length: 1
                  __proto__: DOMStringListPrototype
```

Figure 4-4. There's our lovely little object store!

Defining Primary Keys

In a few moments we'll begin discussing indexes, but before you begin defining different ways to fetch your data, you need to begin with a fundamental property: the primary key. In your object store, every piece of data must have a way to uniquely identify itself. For example, my name is Raymond Camden, and there are certainly other Raymond Camdens out there in the world, but I can be uniquely identified by my Social Security number. (OK, I know that applies only to Americans, but this wouldn't be the first time an American acts like the rest of the world behaves the same.) When you define object stores, you have the opportunity to define how data will be uniquely identified.

Practically, there are two main ways of doing this. One way is to define a *key path*, which is basically a property that will always exist and contain the unique information. So if I were defining a `people` object store, I could say the key path is `ssn`. Another option is to use a *key generator*, which basically means a way to generate a unique value. Here are a few examples.

```
somedb.createObjectStore("people", {keyPath: "email"});
```

This example creates an object store called `people` where it is assumed that each piece of data will contain a property called `email` that is unique.

```
somedb.createObjectStore("notes",     {autoIncrement:true});
```

This example creates an object store called `notes`. The primary key will be assigned automatically as an autoincrementing number.

```
somedb.createObjectStore("logs", {keyPath: "id", autoIncrement:true});
```

This example creates an object stored called `notes`. This time, the autoincrementing value will be used and stored as a property called `id`.

So which one is right? It depends. If you are working with data that has a property in it that should be unique, then you would want to use the `keyPath` option to enforce

this uniqueness. If you are working with data where nothing in the data itself is unique, then using an autoincrementing value will make sense. Example 4-3 demonstrates.

Example 4-3. test_2_2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb3",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");

        if(!thisDB.objectStoreNames.contains("people")) {
            thisDB.createObjectStore("people",
                {keyPath: "email"});
        }

        if(!thisDB.objectStoreNames.contains("notes")) {
            thisDB.createObjectStore("notes",
                {autoIncrement:true});
        }

        if(!thisDB.objectStoreNames.contains("logs")) {
            thisDB.createObjectStore("logs",
                {keyPath:"id", autoIncrement:true});
        }

    }

    openRequest.onsuccess = function(e) {
```

```
        console.log("running onsuccess");
        db = e.target.result;
        console.dir(db.objectStoreNames);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

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

The important part of this example is the `upgradeneeded` event. Three object stores are created, each of which demonstrates various ways of defining the primary key for the object stores. As we aren't actually storing data yet, there isn't much to see here.

Defining Indexes

After figuring out a primary key for your data, next you'll need to decide on your indexes. As we said earlier, indexes define how you plan on fetching data from your object store. This is highly dependent on your data and application needs. Indexes must be made when you create your object stores and can also be used to define a unique constraint on your data. (This is different from the primary key.)

To create an index, you use an instance of an object store variable:

```
objectStore.createIndex("name of index", "path", options);
```

The first argument is the name of the index, while the second refers to the property on the data you wish to index. Most of the time you'll use the same value for both. The final argument is a set of options that defines how the index operates. There are only two options: one for uniqueness, and one used specifically for data that maps to an array. You'll see an example of this later on. Here are two examples:

```
objectStore.createIndex("gender", "gender", {unique:false});
objectStore.createIndex("ssn", "ssn", {unique:true});
```

The first index is on gender and, as you can imagine, allows you to fetch data based on a person's gender. The second index is based on a Social Security number and is also unique.

Let's look at an example. Example 4-4 is a slightly different version of the previous one, so we'll share just the `upgradeneeded` event to keep it a bit more focused.

Example 4-4. Portion of test_2_3.html

```
openRequest.onupgradeneeded = function(e) {
    var thisDB = e.target.result;
    console.log("running onupgradeneeded");

    if(!thisDB.objectStoreNames.contains("people")) {
        var peopleOS = thisDB.createObjectStore("people",
            {keyPath: "email"});

        peopleOS.createIndex("gender", "gender", {unique:false});
        peopleOS.createIndex("ssn", "ssn", {unique:true});

    }

    if(!thisDB.objectStoreNames.contains("notes")) {
        var notesOS = thisDB.createObjectStore("notes",
            {autoIncrement:true});
        notesOS.createIndex("title","title", {unique:false});
    }

    if(!thisDB.objectStoreNames.contains("logs")) {
        thisDB.createObjectStore("logs",
            {keyPath:"id", autoIncrement:true});
    }

}
```

In this updated example, the first and second object stores have indexes. In order to create them, we now use the result of `createObjectStore` so we can run the `createIndex` method on them. The third and final object store does not have indexes, and that's totally fine. One thing to remember is that an index is going to be updated every time you add, edit, or delete data. More indexes mean more work for IndexedDB.

Working with Data

Finally, now that we've talked about the setup and initialization of an IndexedDB database, wouldn't it be nice to actually—I don't know—store data? First and foremost, all data operations with IndexedDB will be done in a *transaction*. You can think of a transaction as a safe wrapper around an operation. If something goes wrong in a transaction, any changes would be rolled back. Transactions add a level of security to your operations that ensure data integrity. What this means for you as a developer is that the simple act of creating, reading, updating, and deleting (CRUD) data will be slightly complex—especially when compared to the ease of use of Web Storage. Transactions in IndexedDB will be specific to one or more object stores, basically using whatever store you need to operate on. They can also be read-only or read and

write. This signifies whether you are changing the database or simply reading from it. Let's begin with creating data.

Creating Data

To create data, you simply call the `add` method of an object store object. At the simplest level, it could look like this:

```
someObjectStore.add(data);
```

If your object store requires you to pass in the primary key at creation, then you would pass that as the second argument:

```
someObjectStore.add(data, somekey);
```

The cool thing is that "data" can be anything you want—a string, a number, an object with strings and numbers, and so on. Like most operations, adding data is asynchronous so you'll need to listen for an event to check the status of the addition.

Let's look at an example. Before we begin, we'll look at the demo in the browser. The demo has two simple forms: one for "Add Person" and one for "Add Note," as seen in Figure 4-5.

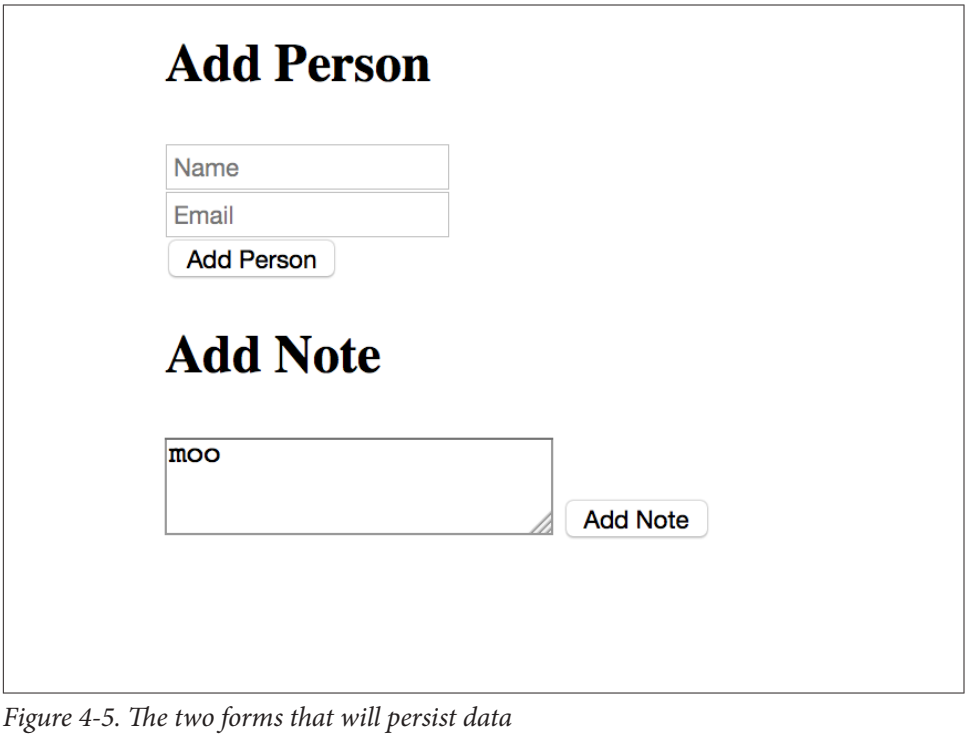

Figure 4-5. The two forms that will persist data

The first form asks for a name and email address, while the second simply asks for a string. The demo doesn't include any type of form validation, but that could be added in a real application. In both forms, you can simply type in data and hit the relevant button, and then the console is used to report the outcome (see Figure 4-6).

```
13:52:32.787    running onsuccess
13:52:45.454    About to add Raymond Camden/raymondcamden@gmail.com
13:52:45.456    Woot! Did it
13:52:50.729    About to add moo
13:52:50.745    Woot! Did it
```

Figure 4-6. The console reports on the success of the data entry

We aren't actually displaying the data, but for now, this is sufficient to test adding data to IndexedDB. Now let's look at the code (Example 4-5).

Example 4-5. test_3_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>

</head>

<body>

<h2>Add Person</h2>
<input type="text" id="name" placeholder="Name"><br/>
<input type="email" id="email" placeholder="Email"><br/>
<button id="addPerson">Add Person</button>

<h2>Add Note</h2>
<textarea id="note"></textarea>
<button id="addNote">Add Note</button>


<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {
```

```
    //No support? Go in the corner and pout.
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb5",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");

        if(!thisDB.objectStoreNames.contains("people")) {
            var peopleOS = thisDB.createObjectStore("people",
                {keyPath: "email"});

        }

        if(!thisDB.objectStoreNames.contains("notes")) {
            var notesOS = thisDB.createObjectStore("notes",
                {autoIncrement:true});
        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;

        //Start listening for button clicks
        $("#addPerson").on("click", addPerson);
        $("#addNote").on("click", addNote);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

function addPerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();

    console.log("About to add "+name+"/"+email);

    //Get a transaction
    //default for OS list is all, default for type is read
    var transaction = db.transaction(["people"],"readwrite");
    //Ask for the objectStore
    var store = transaction.objectStore("people");

    //Define a person
    var person = {
```

```
        name:name,
        email:email,
        created:new Date().getTime()
    }

    //Perform the add
    var request = store.add(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //some type of error handler
    }

    request.onsuccess = function(e) {
        console.log("Woot! Did it");
    }
}

function addNote(e) {
    var note = $("#note").val();

    console.log("About to add "+note);

    //Get a transaction
    //default for OS list is all, default for type is read
    var transaction = db.transaction(["notes"],"readwrite");
    //Ask for the objectStore
    var store = transaction.objectStore("notes");

    //Define a note
    var note = {
        text:note,
        created:new Date().getTime()
    }

    //Perform the add
    var request = store.add(note);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //some type of error handler
    }

    request.onsuccess = function(e) {
        console.log("Woot! Did it");
    }
}
</script>
</body>
</html>
```

This is a rather large demo, so let's break it down bit by bit. Begin by looking at the `upgradeneeded` event handler. This defines two object stores, one called `people` and one called `note`. For `people` we've defined the key path `email` as the primary key, and for `notes` we're using an autoincrementing value. This decision here was arbitrary: for the demo we've decided `people` will be unique by email address, and `notes` will simply have an assigned primary key.

Notice that we don't actually begin handling the form submissions until the `onsuccess` handler for the database runs. This makes sense, as we can't start adding data until the database is ready to be used. But also note we copy a variable `db` to the global scope. This gives us a handler on the database object so we can add data later.

Now turn your attention to the `addPerson` function. This is run when the first form is submitted. After getting the values (and again, some validation could be added here), we begin the process of working with the IndexedDB database. First, a transaction is created. We define the transaction by specifying what object store we care about and what type of transaction we need:

```
var transaction = db.transaction(["people"],"readwrite");
```

From the transaction we then ask for the object store.

```
var store = transaction.objectStore("people");
```

Now comes the fun part. We need to define what we're storing. IndexedDB lets you store pretty much anything you want to. So what I store here is completely up to my particular application needs. In my case I decided to use the values from the form as well as a timestamp for when the person was created. To be clear, this was arbitrary. It just shows that the *form* of your data is up to you.

```
var person = {
    name:name,
    email:email,
    created:new Date().getTime()
}
```

Now comes the persistence. The actual storage request is rather simple:

```
var request = store.add(person);
```

But because it is asynchronous, we need to listen for the results. In our case we listen for the error and success events and simply use the console to report on them. The `addNote` function works the same way—the only difference being the object store that is worked with and the actual data being saved.

Reading Data

Reading data will also be asynchronous and also requires a transaction. Outside of that it is rather simple: `someObjectStore.get(primaryKey)`. Our next demo builds

upon the last, as you can see in Figure 4-7. We've added two new forms to let you fetch data based on the primary key.

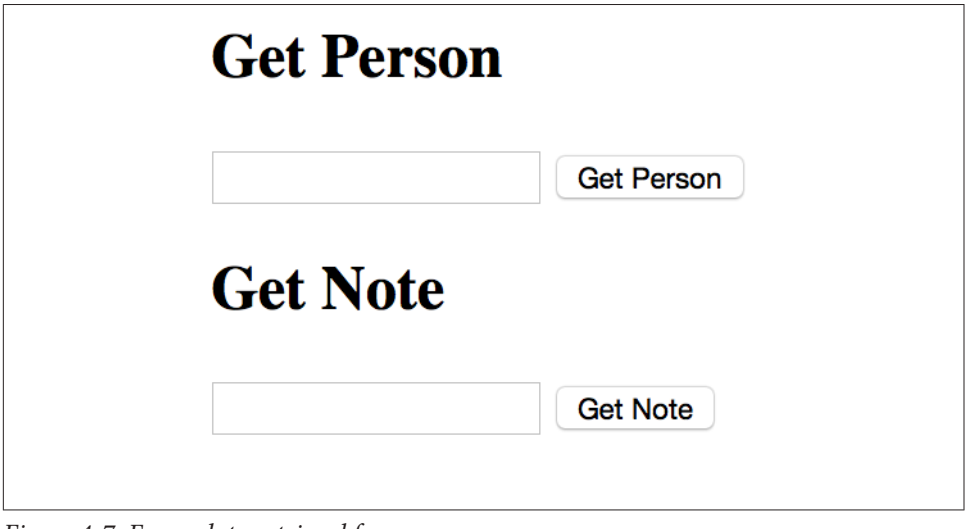

Figure 4-7. Fancy data retrieval forms

Since the code for this demo is so similar to the last one, we'll just focus on the event handlers for the new forms (Example 4-6).

Example 4-6. Portion of test_3_2.html

```
function getPerson(e) {
    var key = $("#getemail").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");

    var request = store.get(key);

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}

function getNote(e) {
```

```
    var key = $("#getnote").val();
    if(key === "") return;

    var transaction = db.transaction(["notes"],"readonly");
    var store = transaction.objectStore("notes");

    var request = store.get(Number(key));

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}
```

Let's begin with `getPerson`. After getting the value representing the primary key you want to load, we once again create a transaction. Note that this time it is a `readonly` transaction. Then we simply fetch the data like so:

```
    var request = store.get(key);
```

In the success handler we dump the result to the console. It looks like Figure 4-8.

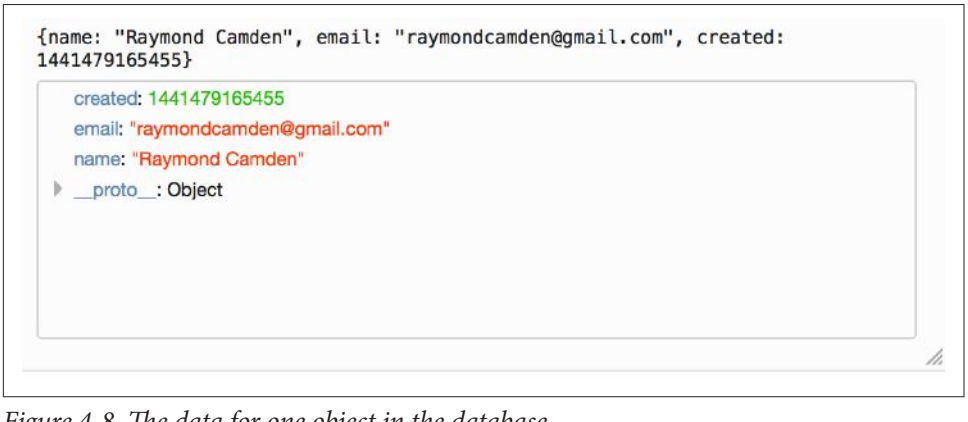

Figure 4-8. The data for one object in the database

If you try to fetch an object that does not exist, the `success` handler will still run, but the result will be undefined. In order for this demo to work for you, be sure to create at least one record and make note of the email address you used. (Later in this chapter we'll look at how to inspect IndexedDB with dev tools and see what data is there.)

Updating Data

You can probably guess what I'm going to say here. Once again you'll need to get a transaction, and once you do, you'll use the `put` method on an object store variable returned from a transaction to store your data. It can be as simple as `someobjectStore.put(data)`, but you can also use the second argument to specify a primary key.

The next demo is a bit more complex. It now asks you for the email address of an existing person (and again, remember to actually enter data in the previous demos). When you enter the email address of a person that exists, it will fill in a form so that you can update the data (see Figure 4-9).

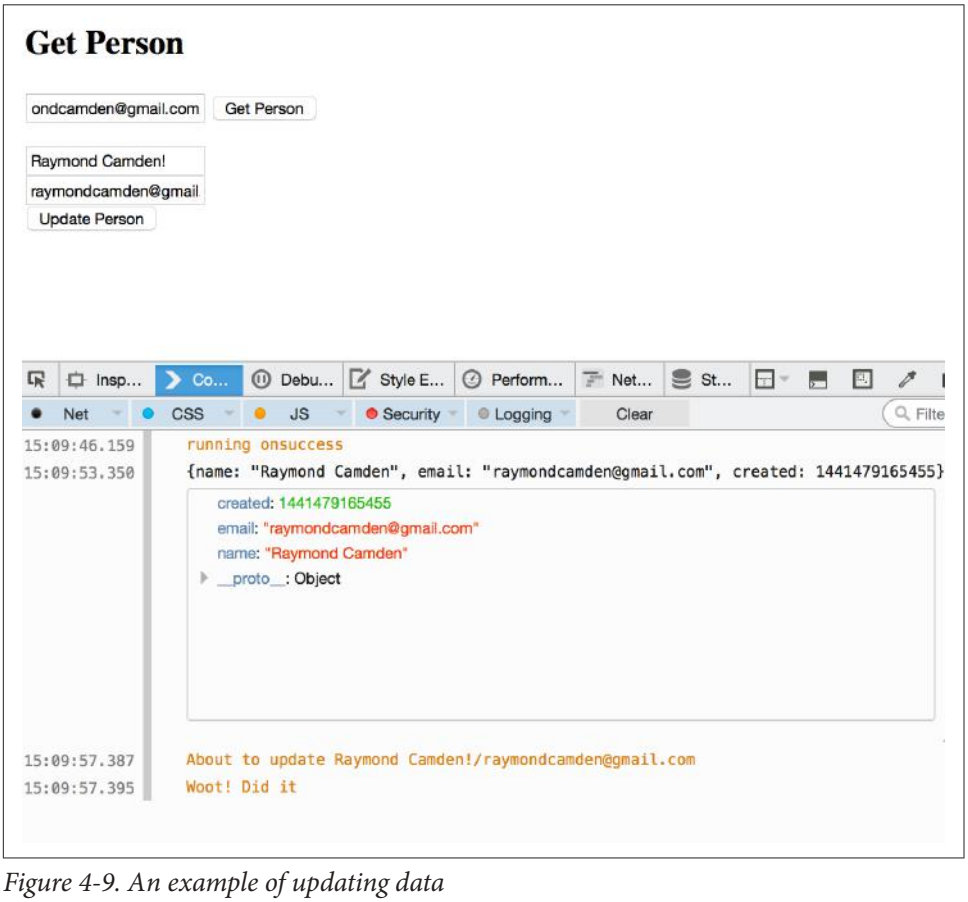

Figure 4-9. An example of updating data

The code that loads the person works the same as the previous demo. Example 4-7 demonstrates the important parts of this particular example.

Example 4-7. Portion of test_3_3.html

```
function getPerson(e) {
    var key = $("#getemail").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");

    var request = store.get(key);

    request.onsuccess = function(e) {
        var result = e.target.result;
        console.dir(result);
        $("#name").val(result.name);
        $("#email").val(result.email);
        $("#created").val(result.created);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}

function updatePerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();
    var created = $("#created").val();

    console.log("About to update "+name+"/"+email);

    //Get a transaction
    //default for OS list is all, default for type is read
    var transaction = db.transaction(["people"],"readwrite");
    //Ask for the objectStore
    var store = transaction.objectStore("people");

    var person = {
        name:name,
        email:email,
        created:created
    }

    //Perform the update
    var request = store.put(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //some type of error handler
    }
```

```
    request.onsuccess = function(e) {
        console.log("Woot! Did it");
    }
}
```

The `getPerson` code is similar to the previous example. Now we actually do something with the result: update the form. `updatePerson` simply takes the form values and persists it using the `put` method just described. Again, there are multiple places here where validation could be added to make things more stable, but you get the idea.

Deleting Data

Now for the final piece of the CRUD puzzle—deleting data. Once again, it will be in a transaction, and once again, it will be asynchronous. The method is simple: `someObjectStore.delete(primarykey)`. Our final demo is likewise simple; it will prompt you for the email address of a person and then delete that person (Figure 4-10).

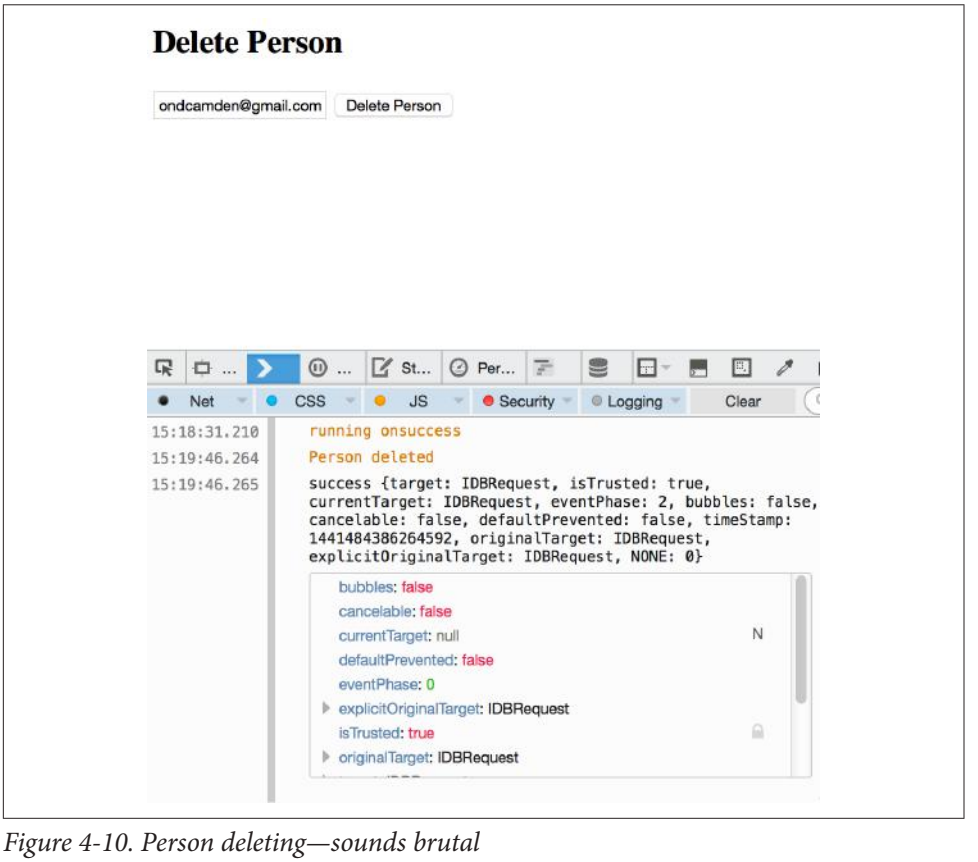

Figure 4-10. Person deleting—sounds brutal

Example 4-8 demonstrates the code that runs when the Delete Person button is clicked.

Example 4-8. Portion of test_3_4.html

```
function deletePerson(e) {
    var key = $("#email").val();
    if(key === "") return;

    var transaction = db.transaction(["people"],"readwrite");
    var store = transaction.objectStore("people");

    var request = store.delete(key);

    request.onsuccess = function(e) {
        console.log("Person deleted");
        console.dir(e);
    }

    request.onerror = function(e) {
        console.log("Error");
        console.dir(e);
    }

}
```

Note that a `delete` operation will fire the `success` handler even if the person doesn't exist. If you wanted to handle that with an error, you would need to get the person first, see if the result was defined, and then perform the `delete`. A transaction around the entire process would ensure data integrity, much like transactions in traditional relational databases.

Getting All the Data

Now that you've seen basic CRUD in action, let's discuss how you can fetch all (and some) of the data in your database. To iterate over the data in an object store, IndexedDB makes use of something called a *cursor*. You can think of a cursor as a happy little beaver who runs into your object store to return one piece of data at a time. Every time it gets a piece of data it brings it back to you, and you ask it to get the next piece. Cursors can move in either direction (so can beavers) and can also be restricted to a "range" of data (not so much for beavers; they are free spirits).

Cursors, just like the CRUD operations, will work within transactions. As before, you'll get a transaction, get an object store from the transaction, and then open a cursor upon that store. Here is an abstract example:

```
var transaction = db.transaction(["test"], "readonly");
var objectStore = transaction.objectStore("test");
```

```
var cursor = objectStore.openCursor();

cursor.onsuccess = function(e) {
    var res = e.target.result;
    if(res) {
        //stuff
        res.continue();
    }
}
```

Notice the `success` handler for the cursor. The event result contains the data that the beavercursor currently has. It also has a `continue` method. That's how you tell the cursor to go fetch the next object. If the result was undefined, that means you were at the end of the cursor.

Our new demo, shown in Figure 4-11, now includes a way to list all the people in the database (as well as to add, in case you deleted everyone in the previous example).

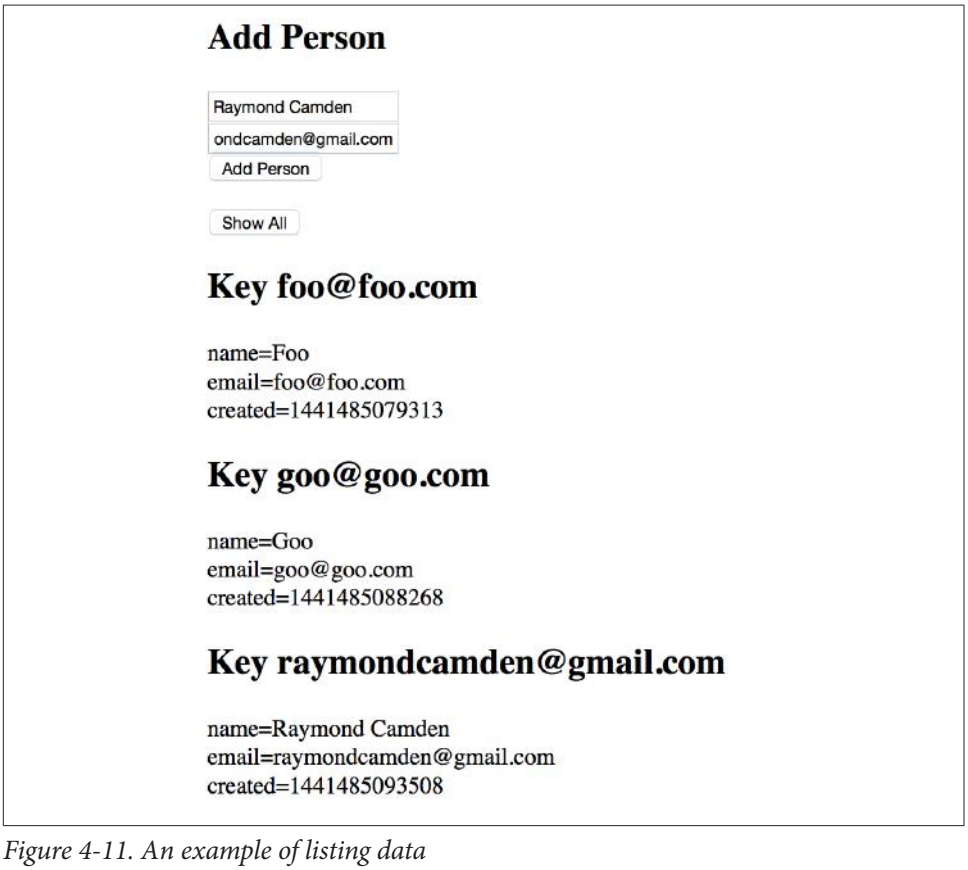

Figure 4-11. An example of listing data

Since the "Add" code isn't new, let's focus on the listing code (Example 4-9).

Example 4-9. Portion of test_4_1.html

```
function getPeople(e) {

    var s = "";

    var transaction = db.transaction(["people"], "readonly");
    var people = transaction.objectStore("people");
    var cursor = people.openCursor();

    cursor.onsuccess = function(e) {
        var cursor = e.target.result;
        if(cursor) {
            s += "<h2>Key "+cursor.key+"</h2><p>";
            for(var field in cursor.value) {
                s+= field+"="+cursor.value[field]+"<br/>";
            }
            s+="</p>";
            cursor.continue();
        }
    }

    transaction.oncomplete = function() {
        $("#results").html(s);
    }
}
```

As expected, you begin with a transaction, then move on to a store, and finally you open the cursor. The success handler is run every time an object is fetched. In order to update the web page, we're going to use a variable, `s`, that contains HTML representing the data itself. Note that it would be nicer to use a template language like Handlebars here. The cursor object contains a `key` property that represents the primary key for this item. The cursor object also contains a `value` property that represents the data. We can iterate over each key in the object and add it to the string. In a "real" application you wouldn't do this; you would know your data contains certain properties and output them directly. This code is just simple and generic.

The last part is crucial. How do you know when the cursor is complete? When you are no longer fetching data, the transaction object will fire a complete event. We can use that to take the string variable and inject it into the DOM.

Working with Ranges and Indexes

The cursor example you saw previously is useful for printing *all* the data, but typically you will want to work only with a subset of your data. This is where indexes come in. Indexes are based on a property of your data. Within that data, you can request a range of data.

So imagine an object store of people with an index on `name`. You could request a range of data based on names that begin with *B* and upward. (*C*, *D*, and so on.) You could instead request a range that begins at the "lowest" value for a name and goes up to *T*. Finally, you could request a range between *R* and *S*.

And to make things even more complex, for all of the preceding examples you can switch between an inclusive and exclusive mode. What does that mean? Imagine a range between *B* and *E*. An inclusive range will include *B* and *E* itself, giving you names like Barry and Elric. An exclusive range will give you values between *B* and *E* but not including names starting with those letters. So the first result may be Corwin. (And yes, numerical ranges work too.)

Finally, you can also create a "range" of one value, so, for example, just names that begin with *R* (like Raymond).

Working with ranges is only slightly different than cursors. Instead of opening a cursor on an object store, you open it on an index instead. As an example:

```
//make an IDBKeyRange
range = IDBKeyRange.upperBound("Camden");
cursor = someIndex.openCursor(range);
//or
cursor = someIndex.openCursor(range, "prev");
```

Notice that the range uses an upper bound of `"Camden"`, which means the name must be "lower" than Camden in a string comparison. So, for example, Cameron would not be lower, but Cade would be.

Ranges are created from an IDBKeyRange API. Methods include `upperBound`, `lowerBound`, `bound` (which means both), and `only`. Ranges are inclusive automatically, but if you pass `false` into the second (or in the case of `bound`, third) argument, you can specify exclusive. By default the direction of the cursor is `"forward"`, but in the last example you can see how to specify a backward traversal.

All of that is rather complex, so let's look at an example. The demo has been extended so that you can now search against people names, as shown in Figure 4-12. It lets you specify a search that starts at a letter, ends at a letter, or works between them both.

Add Person

Jay

jay@foo.com

Add Person

Search People

Starting with: B

Ending with:

Search

Key Jay

name=Jay
email=jay@foo.com
created=1441486226022

Key Ray

name=Ray
email=raymondcamden@gmail.com
created=1441486194446

Figure 4-12. A person search form

Before you try this code yourself, note that it is using a new IndexedDB database. Be sure to enter some values in the Add Person form on top so you have data to actually search. Since adding people isn't new, let's focus on search in Example 4-10.

Example 4-10. Portion of test_4_2.html

```
function searchPeople(e) {

    var lower = $("#lower").val();
    var upper = $("#upper").val();

    if(lower == "" && upper == "") return;

    var range;
    if(lower != "" && upper != "") {
        range = IDBKeyRange.bound(lower, upper);
    } else if(lower == "") {
        range = IDBKeyRange.upperBound(upper);
    } else {
        range = IDBKeyRange.lowerBound(lower);
    }

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");
    var index = store.index("name");

    var s = "";

    index.openCursor(range).onsuccess = function(e) {
        var cursor = e.target.result;
        if(cursor) {
            s += "<h2>Key "+cursor.key+"</h2><p>";
            for(var field in cursor.value) {
                s+= field+"="+cursor.value[field]+"<br/>";
            }
            s+="</p>";
            cursor.continue();
        }
    }

    transaction.oncomplete = function() {
        //no results?
        if(s === "") s = "<p>No results.</p>";
        $("#results").html(s);
    }


}
```

The search function begins by reading the values from the search form and doing a tiny bit of validation. At this point, things get tricky. Remember that we can search

from a letter, to a letter, or between letters. That means we need one of three types of ranges. That's what the next code block does. Based on your input, it figures out the right type of range to use. Once past that, the transaction is opened, the object store is fetched, and then the name index is retrieved.

Now when the cursor is fetched, the range is passed to it as an argument. Outside of that, the cursor object is treated the same as before in Example 4-9.

You may be wondering, what about more complex search—for example, people with a name beginning with x that are gender y and have an age between 10 and 30? Unfortunately, complex search is not something IndexedDB is good at. It isn't going to replace the power of a proper SQL database engine like MySQL. This is something to keep in mind when building your applications.

Even More with IndexedDB

We're not quite done with IndexedDB yet. Let's look at two interesting tricks you can do with the feature.

Storing Arrays

We mentioned earlier that nearly anything can be stored in IndexedDB, even array data. So, for example, this is completely fine:

```
var person = {
    name:"Ray",
    age:43,
    background:{
        born:1973,
        bornIn:"Virginia"
    },
    hobbies:["comics","movies","bike riding"]
}
someStore.add(person);
```

That's cool and all, and just plain works, but it brings up an interesting question: What if you wanted to fetch people based on their hobbies? Well, this is where the `multiEntry` option comes into play. When defining an index on a property that is array based, simply use this option and set it to `true`.

```
objectStore.createIndex("hobbies", "hobbies", {unique:false, multiEntry:true});
```

This tells IndexedDB to properly store each item in the array in the index so you can fetch someone based on one particular value. Let's look at a demo (Figure 4-13).

Add Person

Ray

ondcamden@gmail.com

books,movies

Add Person

Search Hobbies

books

Search

Key books

name=Ray
email=raymondcamden@gmail.com
hobbies=books,movies
created=1441487275789

Key books

name=Spock
email=spock@gmail.com
hobbies=books,cookies,beer
created=1441487258756

Figure 4-13. Our people have hobbies now

In Figure 4-13, you can see now that the Add Person form has been updated to include a hobby field. When testing, you should enter hobbies in a comma-separated list with no spaces between them—for example, `cookies,beer,movies`. Do not do `cookies, beer, movies`. (And again, in a released application you could handle spaces by removing them in code.) Now the search is hobby based. By entering the name of a hobby, you can find people who specified that as a hobby. Let's look at the code, and since this is somewhat different than earlier examples, we'll share the complete listing (Example 4-11).

Example 4-11. test_5_1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src=
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<h2>Add Person</h2>
<input type="text" id="name" placeholder="Name"><br/>
<input type="email" id="email" placeholder="Email"><br/>
<input type="text" id="hobbies" placeholder="Hobbies"><br/>
<button id="addPerson">Add Person</button>
<p/>

<h2>Search Hobbies</h2>
<input type="text" id="hobby"> <button id="search">Search</button>

<div id="results"></div>

<script>
function idbOK() {
    return "indexedDB" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!idbOK()) return;

    var openRequest = indexedDB.open("ora_idb7",1);

    openRequest.onupgradeneeded = function(e) {
        var thisDB = e.target.result;
        console.log("running onupgradeneeded");
```

```
        if(!thisDB.objectStoreNames.contains("people")) {
            var peopleOS = thisDB.createObjectStore("people",
                {keyPath: "email"});

                peopleOS.createIndex("name", "name",
                {unique:false});
                peopleOS.createIndex("hobbies", "hobbies",
                {unique:false, multiEntry: true});

        }

    }

    openRequest.onsuccess = function(e) {
        console.log("running onsuccess");
        db = e.target.result;

        //Start listening for button clicks
        $("#addPerson").on("click", addPerson);
        $("#search").on("click", searchPeople);
    }

    openRequest.onerror = function(e) {
        console.log("onerror!");
        console.dir(e);
    }

});

function addPerson(e) {
    var name = $("#name").val();
    var email = $("#email").val();
    var hobbies = $("#hobbies").val();

    if(hobbies != "") hobbies = hobbies.split(",");

    console.log("About to add "+name+"/"+email);

    //Get a transaction
    //default for OS list is all, default for type is read
    var transaction = db.transaction(["people"],"readwrite");
    //Ask for the objectStore
    var store = transaction.objectStore("people");

    //Define a person
    var person = {
        name:name,
        email:email,
        hobbies:hobbies,
        created:new Date().getTime()
    }
```

```
    //Perform the add
    var request = store.add(person);

    request.onerror = function(e) {
        console.log("Error",e.target.error.name);
        //some type of error handler
    }

    request.onsuccess = function(e) {
        console.log("Woot! Did it");
    }
}

function searchPeople(e) {

    var hobby = $("#hobby").val();

    if(hobby == "") return;

    var range = IDBKeyRange.only(hobby);

    var transaction = db.transaction(["people"],"readonly");
    var store = transaction.objectStore("people");
    var index = store.index("hobbies");

    var s = "";

    index.openCursor(range).onsuccess = function(e) {
        var cursor = e.target.result;
        if(cursor) {
            s += "<h2>Key "+cursor.key+"</h2><p>";
            for(var field in cursor.value) {
                s+= field+"="+cursor.value[field]+"<br/>";
            }
            s+="</p>";
            cursor.continue();
        }
    }

    transaction.oncomplete = function() {
        //no results?
        if(s === "") s = "<p>No results.</p>";
        $("#results").html(s);
    }

}

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

That's quite a bit of code, but the changes really are somewhat minimal. First, make note of the new index on people:

```
peopleOS.createIndex("hobbies", "hobbies",
{unique:false, multiEntry: true});
```

As we said before, `multiEntry` being `true` is the magic flag to make this all work. Now scroll down to the `addPerson` logic. To store the array, we simply convert the string value from the form into a JavaScript array:

```
if(hobbies != "") hobbies = hobbies.split(",");
```

Finally, the search needs to find exact matches, so instead of a range to and from something it uses the `only` method.

```
var range = IDBKeyRange.only(hobby);
```

Counting Data

For our final demo, we're going to show how to count the data in an object store. You may have thought you'd need to iterate over the entire table using a cursor. However, there is a much simpler way of counting data: using the `count` method. The `count` method of an object store does exactly what you think it does—it asynchronously returns the number of objects in the store. Here is an example.

```
db.transaction(["note"],"readonly").objectStore("note").count().onsuccess =
function(event) {
    console.log('total is '+event.target.result);
}
```

Notice that we're chaining the various method calls together in one slick line so we can look cool to our coworkers. That is totally unnecessary. The actual count value is available in the event result value. You can find an example of this in the code that ships with the book (*test_5_2.html*).

Inspecting IndexedDB with Dev Tools

As with Web Storage, both Firefox and Chrome provide nice tools to let you work with IndexedDB. In Figure 4-14 you can see an example of Firefox's support.

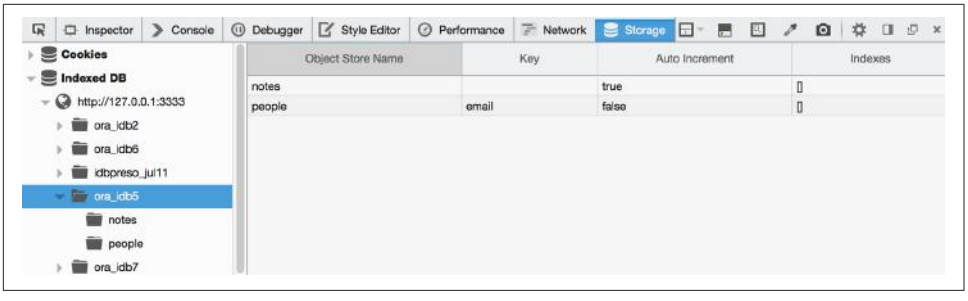

Figure 4-14. Firefox Dev Tools support for IndexedDB

Along with a high-level view of the databases and object stores, you can select a store for a detailed value list (see Figure 4-15).

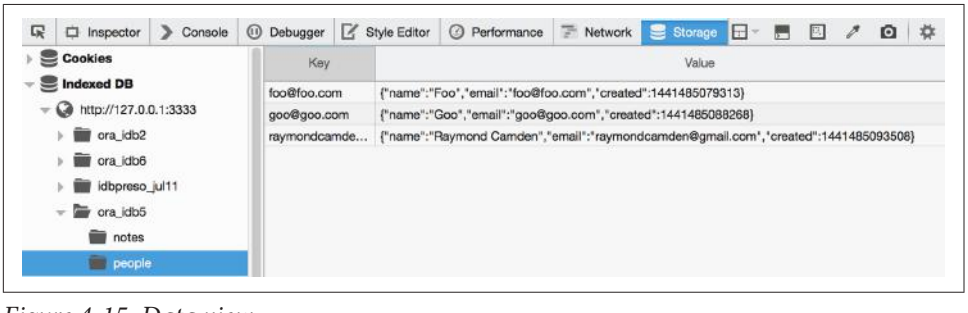

Figure 4-15. Data view

Figure 4-16 shows how Chrome renders it. Note the crossed-out circle at the bottom. It lets you quickly delete data as well.

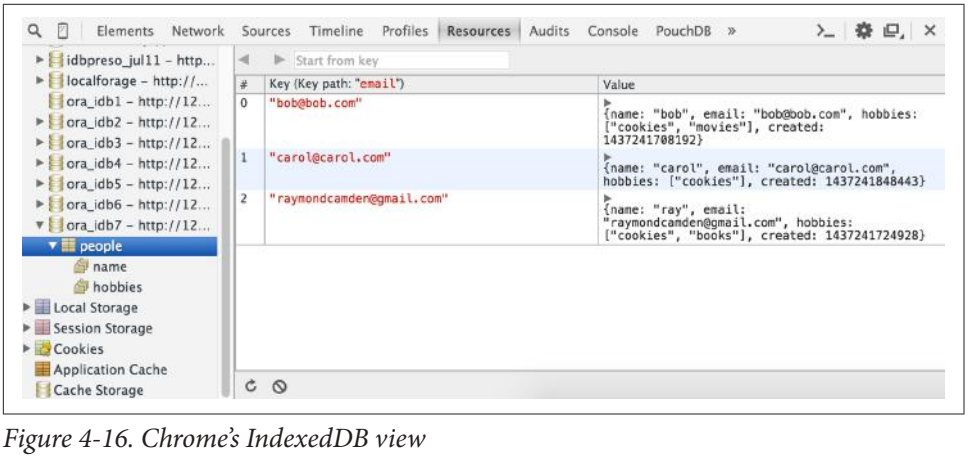

Figure 4-16. Chrome's IndexedDB view

Support and Recommended Usage

So, how well is IndexedDB supported? Figure 4-17 shows the report from CanIUse.com.

Figure 4-17. IndexedDB support table from CanIUse.com

That's...OK but not stellar. It is growing, however, and even iOS will support it (properly) soon.

As for recommended uses, I'd consider anything that the user can create to be a good candidate for IndexedDB. You could use it to copy nonprivate intranet information locally for faster retrieval and offline support as well. Game assets, like small music files and game data, could be copied here too.

CHAPTER 5

Working with Web SQL

Dead Spec Walking

Before we say anything at all about Web SQL, we regret to inform you that this spec is dead. Or dying. Or at least sentenced to death. Web SQL as a feature was a rather interesting one. It gave you access to miniature databases within the browser. For web developers who did server-side work, this was especially nice because they may have had some familiarity with SQL already. However, for reasons that are not important to this book, the specification has been EOLed (End of Life) and will (possibly) not be available in the future. That means—in theory—you shouldn't even be reading this chapter.

However...

Web SQL has *very* good support on mobile browsers, and was available before IndexedDB and much better supported than IndexedDB. It is entirely possible that as a developer, you will run into web apps making use of Web SQL. While we don't recommend starting new projects with Web SQL, we hope this chapter will give you enough knowledge about the ins and outs of Web SQL so that if you have to help support an existing implementation you'll know what to do.

As before, this particular client-side data storage technique will be tied to a particular domain. Storage limits are pretty varied and can range from 5 MB to 50 to more. Before we dive in, let's cover some basic terms.

Basic Database Terms

For those of you with experience working with traditional relational database servers, you can just go ahead and skip this section.

Databases
: A database is the top-level container where you would store your data. As with IndexedDB, you can have as many of these as you would like, but typically you'll probably stick to one database per site.

Tables
: This is where data for a particular type of information is stored. Unlike object stores in IndexedDB, tables are very strict about what gets stored. If you have defined a "people" table as having name, age, and gender columns, you can only store values in those columns. Each column also has a particular type of expected data, and you must match that criteria to store your data.

Rows
: A row is simply an individual unit of data for a table. Given a table called people, one row would be a person.

Checking for Web SQL Support

The simplest way to check for Web SQL support is by checking for the `openDatabase` API of the `window` object:

```
if("openDatabase" in window) {
}
```

Which can be nicely turned into a function like so:

```
function webSQLOK() {
    return "openDatabase" in window;
}
```

Working with Databases

Much like IndexedDB, databases in Web SQL have a name and a version. Unlike in IndexedDB, though, the version number will not provide an event for you to perform changes; instead, it acts as a validation. If the user had an earlier version of the database, you can perform an update manually to handle changes. (We'll show an easier way around that, however.) Next you provide a "friendly name" for the database, which as far as I can tell is never referenced again. You also are required to provide an initial size for the database. This is an estimated size, and frankly, most examples I've seen use the same value (5 MB) and don't bother actually trying to figure out what

size they really think they need. Your code will begin by opening the database, which is a synchronous API.

```
db = window.openDatabase("name","1","nice name",5*1024*1024);
```

Note that I've taken the result of the `window.openDatabase` call and stored it. This lets you perform operations on the database later on. Let's consider a simple example that just opens up a database and uses the console to display the object itself (Example 5-1).

Example 5-1. test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function websqlOK() {
    return "openDatabase" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    console.dir(db);

});

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

While there isn't much going on here, this shows the basic "how to get started"-type code for working with Web SQL. Figure 5-1 shows what Chrome displays in the console for the database object.

```
▼Database
    version: "1"
  ▼__proto__: Database
    ▶changeVersion: function changeVersion()
    ▶constructor: function Database()
    ▶readTransaction: function readTransaction()
    ▶transaction: function transaction()
     version: (...)
    ▶get version: function ()
    ▶__proto__: Object
>
```

Figure 5-1. Chrome's dump of the database object

Working with Transactions

Once you've gotten a database object, you can then begin doing, well, everything—and unlike in IndexedDB, working with data in Web SQL is rather simple (assuming you know SQL, of course). Once again, a transaction will be used, and once again, you get to specify either a read-only or read/write transaction, but after that, the code remains the same no matter what you do. All that changes is the SQL. As a basic example, here is how you open a read-only transaction (this assumes you've created a Web SQL variable called `db`):

```
db.readTransaction(function to do stuff, error handler, success handler);
```

In real code, this could look like this:

```
db.readTransaction(function(tx) {
    tx.executeSql("select * from foo");
}, function(e) {
    console.log("Db error ",e);
}, function() {
    console.log("Done");
});
```

The first argument to the `readTransaction` call is a function that is provided a transaction object. On that object, you can run `executeSql`, which, as you might guess, is where you perform SQL queries. The second argument is the error handler and the last is the success handler.

So far, so good. This is where things get a tiny bit confusing. First, here is how the API works in general:

```
    tx.executeSql("sql statement", "array of values", "success handler",
    "error hander");
```

Disregard the second argument for now; we'll come back to it. The thing you'll want to note most is that the order of handlers (success and then error) is the *opposite* of the transaction call. This is very easy to mess up, so be careful when working with these handlers.

Let's enhance the initial demo to do some setup work. Before you can store data in a database, you'll need a table. Luckily, it isn't difficult to create tables in SQL. Example 5-2 shows how.

Example 5-2. test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<script>
function websqlOK() {
    return "openDatabase" in window;
}

var db;

$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    db.transaction(function(tx) {
        tx.executeSql("create table if not exists notes(id INTEGER PRIMARY "+
        "KEY AUTOINCREMENT, title TEXT, body TEXT, updated DATE)");
    },dbError,function(tx) {
        ready();
    });

});

function dbError(e) {
    console.log("Error", e);
}

function ready() {
```

```
    console.log("Ready to do stuff!");
}
</script>
</body>
</html>
```

In this version, after the database is opened we create a read/write transaction (using `db.transaction`). We then use SQL to create a table. The cool thing about this SQL is that it gracefully handles *not* doing anything if the table already exists. As we stated earlier, Web SQL does include the concept of versioning and does offer a way to perform tasks when versions change, but this style of setup is far simpler and most likely sufficient for your needs. You could execute multiple different SQL statements there to set up as many tables as you need.

Obviously all of this is simple if you know SQL. If you don't, there are various books and tutorials that can help you. The SQL used in Example 5-2 creates a table called `notes`. It has an `id` column that will be the primary key for data, a `title` and `body` column that contain text, and an `updated` column storing a date value.

Now let's kick it up a notch with a real, but simple, demo in Example 5-3.

Example 5-3. test3.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
</head>

<body>

<h2>Add a Note</h2>
<form>
Title: <input type="text" id="title"><br/>
Body:<br/>
<textarea id="body"></textarea><br/>
<button id="addNote">Add Note</button>
</form>

<p/>

<table id="notes" border="1"><tbody></tbody></table>

<script>
function websqlOK() {
    return "openDatabase" in window;
}

var db;
```

```
$(document).ready(function() {

    //No support? Go in the corner and pout.
    if(!websqlOK()) return;

    db = window.openDatabase("db1", "1", "Database 1", 5*1024*1024);

    db.transaction(function(tx) {
        tx.executeSql("create table if not exists notes(id INTEGER PRIMARY "+
        "KEY AUTOINCREMENT, title TEXT, body TEXT, updated DATE)");
    },dbError,function(tx) {
        ready();
    });

});

function dbError(e) {
    console.log("Error", e);
}

var $title, $body, $notesTable;

function ready() {
    $("#addNote").on("click", addNote);
    $title = $("#title");
    $body = $("#body");
    $notesTable = $("#notes tbody");
    renderNotes();
}

function addNote(e) {
    e.preventDefault();
    //no validation
    var title = $title.val();
    var body = $body.val();

    db.transaction(function(tx) {
        tx.executeSql("insert into notes(title,body,updated) values(" +
        "'" + title + "','" + body + "'," + (new Date().getTime()) +")");
    },dbError,function(tx) {
        $title.val("");
        $body.val("");
        renderNotes();
    });

}

function renderNotes() {
    db.readTransaction(function(tx) {
        tx.executeSql("select * from notes order by updated desc",[],
        function(tx, results) {
```

```
            var rowStr = "";
            for(var i=0;i<results.rows.length;i++) {
                var row = results.rows.item(i);
                //use row.col
                rowStr += "<tr><td>" + row.title + "</td>";
                rowStr += "<td>" + row.body + "</td>";
                var d = new Date();
                d.setTime(row.updated);
                rowStr += "<td>" + d.toDateString() + " " + d.toTimeString();
                rowStr += "</td></tr>";
            };
            $notesTable.empty();
            $notesTable.append(rowStr);
        });
    },dbError);

}
</script>
</body>
</html>
```

This new version includes a form and an empty table. The form will be used to let the user enter a note (a title and the body), while the table will be used to display existing data. That's it for the user interface—now let's dig into the code.

The `ready` function will be run after the initial table creation SQL is executed. Remember, this SQL safely runs multiple times, as it won't recreate the table after the first time. We add a simple click handler for the form, store some jQuery variables from the DOM, and immediately run the `renderNotes` function.

The `addNote` click handler simply fetches the form values and then creates a SQL statement to handle the insert. This SQL statement is rather brittle. We'll fix that in the next update. Once the SQL statement is executed, `renderNotes` is run again to update the display.

Within `renderNotes`, we again have a transaction, but note the change to a read-only transaction. You can see that we select all rows and order by the `updated` column so we always get the latest data first. Ignore that empty array argument for now. Once the SQL is executed, we can work with the results. The success handler for `executeSql` is passed the transaction object itself and the results. This `results` object is an instance of a `SQLResultSet`. It has a `rows` property, which has a `length` allowing us to loop over it. To get an individual row, the `item` method is called with the corresponding row number. That row object is just a set of key/value pairs representing the columns in the row. A string is used to construct a table (yes, yes, I know, tables are passé), which is then rendered out to the DOM. Figure 5-2 demonstrates how this looks. (And yes, it could be much better designed.)

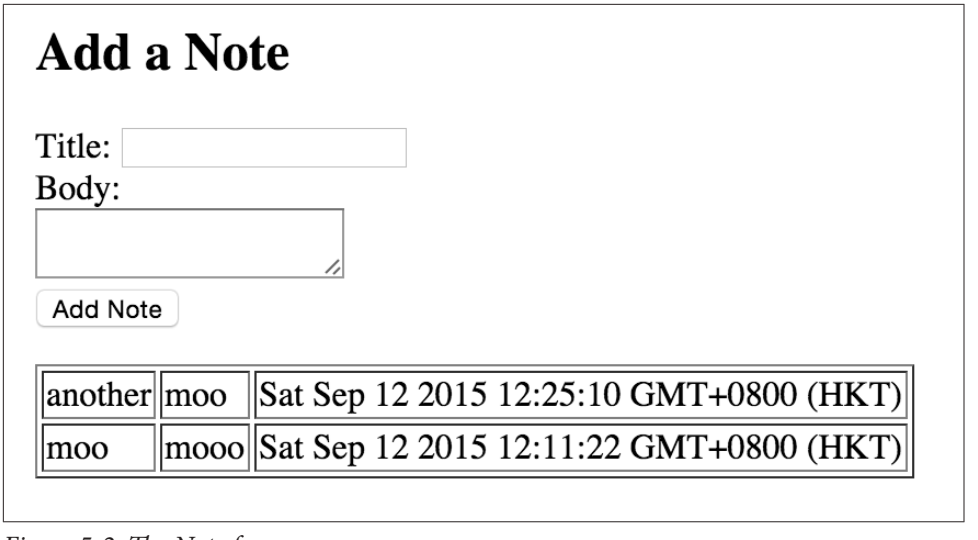

Figure 5-2. The Note form

So, now that you have a basic idea of how Web SQL works, let's focus on the insert statement from Example 5-3:

```
tx.executeSql("insert into notes(title,body,updated) values(" +
"'" + title + "','" + body + "'," + (new Date().getTime()) +")");
```

When executed, this generates a SQL statement that could look like so:

```
insert into notes(title, body, updated)
values('some title', 'some body', 1)
```

The issue with the code is that if the form values themselves included a single quote character, the SQL would break. Typically, allowing user input to drive dynamic SQL leads to something called a *SQL injection attack*. It's nasty, but luckily easily fixed. Remember that second argument that was an empty array? Instead of creating a dynamic SQL string with concatenation, you can use "tokens" within the SQL that represent variables. You can then use the array argument to supply those values. Example 5-4 demonstrates how simple this is to put into effect.

Example 5-4. Partial code from test4.html

```
function addNote(e) {
    e.preventDefault();
    //no validation
    var title = $title.val();
    var body = $body.val();

    db.transaction(function(tx) {
        tx.executeSql("insert into notes(title,body,updated) "+
```

```
        "values(?,?,?)", [title, body, new Date().getTime()]);
    },dbError,function(tx) {
        $title.val("");
        $body.val("");
        renderNotes();
    });

}
```

Notice how the SQL now is a simple string—no embedded variables. Where the variables were, there are now question marks. They will be replaced in the same order as the values included in the array in the next argument.

Inspecting Web SQL with Dev Tools

You can find pretty good support for Web SQL in Chrome's Dev Tools. Under the Resources tab, you'll see a section just for Web SQL along with any defined databases. Selecting one and expanding it then lets you select a table to view all the data (see Figure 5-3).

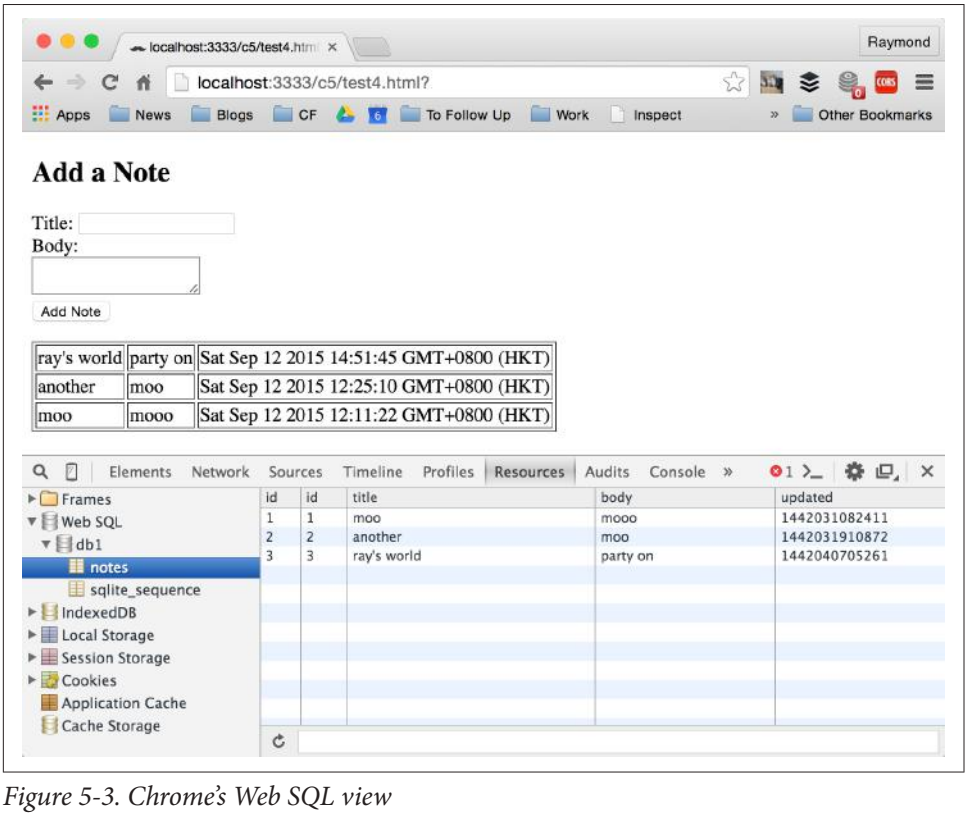

Figure 5-3. Chrome's Web SQL view

The empty text field at the bottom of the data display lets you enter the name of a column. Doing so will filter the view to just the primary key and that column. What isn't terribly obvious is that if you click on the database itself, you'll see a console where you can enter arbitrary SQL statements (see Figure 5-4).

Figure 5-4. Running SQL commands in Dev Tools

As we mentioned in the beginning of the chapter, you'll probably not be starting new projects with Web SQL, but if you have to debug an existing one, Chrome's Dev Tools support can be very useful.

Support and Recommended Usage

Let's look at the current state of Web SQL support in Figure 5-5.

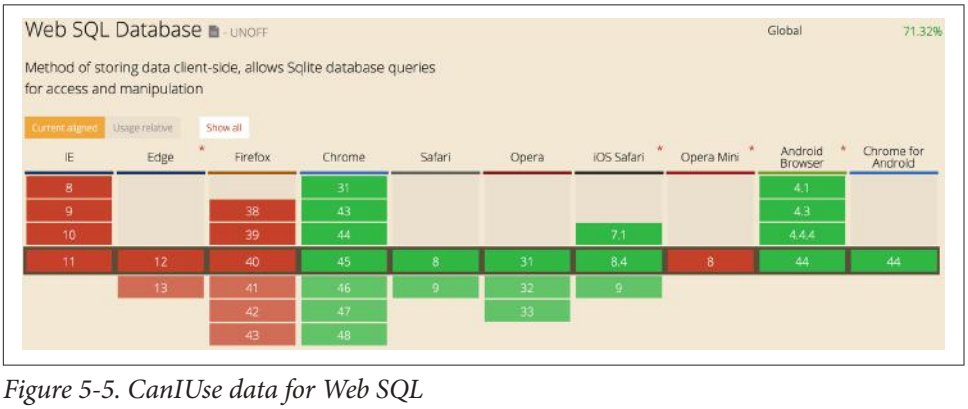

Figure 5-5. CanIUse data for Web SQL

As you can see, Chrome, Safari, and their respective mobile versions support this feature—not bad for a dead spec. But alas, it is truly dead (or on the way out), so the recommendation is to *not* use it if you can avoid it. If you are going to use it, then the same places where you would use IndexedDB would certainly apply here as well.

CHAPTER 6

Making It Easier with Libraries

"Use the Library, Luke..."

OK, so that may not be *exactly* how the quote goes, but consider this. Client-side storage is a useful feature of modern browsers. Because it is useful, friendly developers have created libraries to help make using client-side storage even easier. In some cases, these libraries make the APIs easier to use. In some cases, they add features that the native API doesn't even support. As you can imagine, there are quite a few of these libraries available to you, but in this chapter we'll look at three in particular: Lockr, Dexie, and localForage.

Working with Lockr

Our first library is Lockr, a wrapper for Web Storage (see Figure 6-1). Right away you may be wondering why in the heck anyone would need to make Web Storage simpler, but stick with me and you'll see why in a moment. Lockr provides a Redis-like API for Web Storage, but don't worry if you've never heard of Redis. It is a *very* small library (2.5 KB), and like everything we'll be covering in this chapter it is free and open source. The Lockr home page may be found here: *https://github.com/tsironis/lockr*.

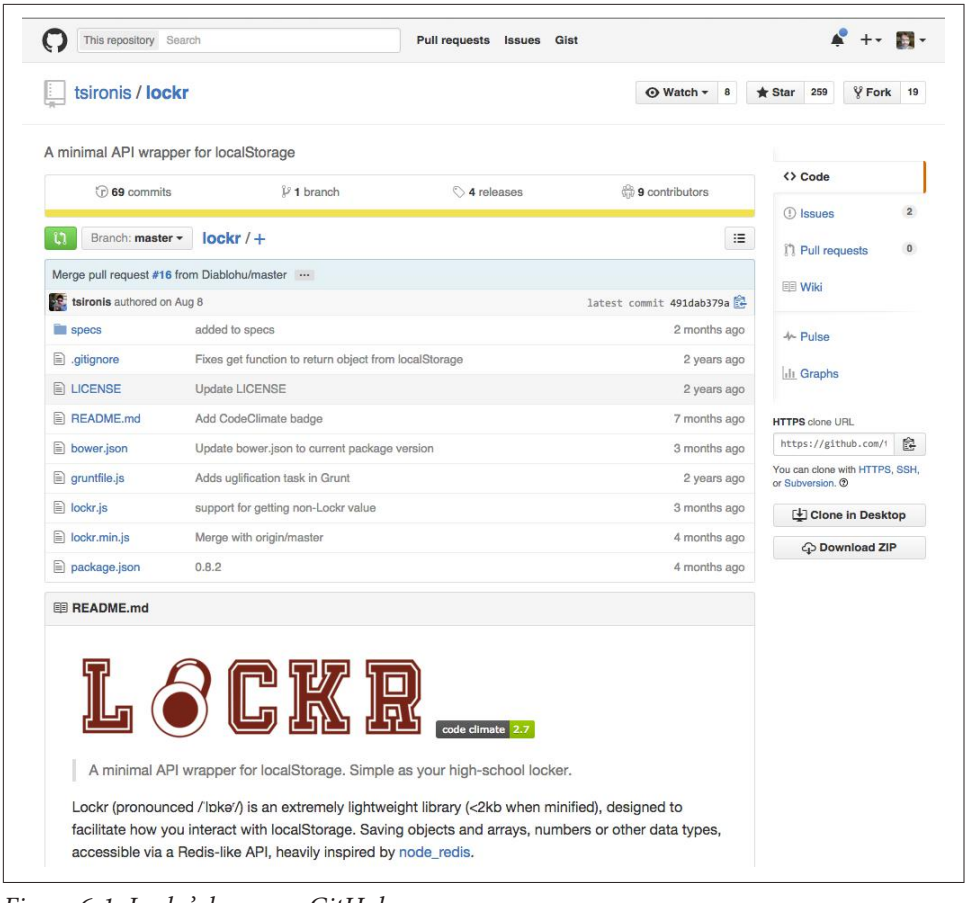

Figure 6-1. Lockr's home on GitHub

You can grab the source from GitHub or install via Bower: `bower install lockr`. Once you have the library downloaded, simply include it in your code like any other JavaScript library.

Using Lockr is relatively simple. So, for example, this will set a value:

```
Lockr.set("name", "Raymond");
```

And this will get a value:

```
Lockr.get("name");
```

So why bother? Well, let's consider two interesting examples. First, take a look at Example 6-1.

Example 6-1. lockr/test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {

    Lockr.set("name", "Ray");
    Lockr.set("age", 43);

    var name = Lockr.get("name");
    var age = Lockr.get("age");
    console.log(name, age + 1);

    //compare to localStorage
    localStorage.setItem("age_ls", 43);
    console.log(localStorage.getItem("age_ls")+1);

});

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

The code begins with two simple sets—a name and an age—and then fetches them and prints them to the console. Notice, though, that we add 1 to the `age` value. Immediately after this is a similar test using "regular" Web Storage. If you run this, you'll discover something interesting, as shown in Figure 6-2.

```
Ray 44                                  test1.html:20
431                                     test1.html:24
```

Figure 6-2. Comparing Lockr and basic Web Storage

Do you see it? When Lockr retrieved and then modified the numeric value, it worked correctly. But Web Storage treats everything as a string, so getting the age and "adding 1" ended up adding a 1 to the end of the value. OK, so that's not a terribly big deal, but how about Example 6-2?

Example 6-2. lockr/test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {

    Lockr.set("stuff", [1,2,3,4]);
    Lockr.set("person", {
        name:"Ray",
        age:43,
        hobbies:["stuff","more stuff"]
    });

    var stuff = Lockr.get("stuff");
    var person = Lockr.get("person");
    console.dir(stuff);
    console.dir(person);

});

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

In this example, we've taken two complex objects and stored them with Lockr. We did not have to serialize them to JSON on storage or retrieval, as Figure 6-3 demonstrates.

```
▼Array[4]                                          test2.html:24
    0: 1
    1: 2
    2: 3
    3: 4
    length: 4
  ▶ __proto__: Array[0]
▼Object                                            test2.html:25
    age: 43
  ▼hobbies: Array[2]
      0: "stuff"
      1: "more stuff"
      length: 2
    ▶ __proto__: Array[0]
    name: "Ray"
  ▶ __proto__: Object
```

Figure 6-3. Lockr deftly handles complex data

But wait—it gets better. You can also use Lockr's `get` API to return a default value when there isn't an existing value in Web Storage:

```
var coolness = Lockr.get("coolness", "Infinity!");
```

In this snippet, if there wasn't a value for `coolness` in Web Storage, then `"Infinity!"` will be returned. (You can find a full demo of this in *lockr/test3.html*.)

Lockr also supports a special type of value called a *hash*. Given an array of data, Lockr will let you add unique values to it. If you try to add a value that already exists, it will not be added again. So, for example, given an array that consists of three numbers, `[1, 8, 9]`, if you try to add another 9, Lockr will not append it to the array. If you try to add 4, however, Lockr will allow it, so the array will now be `[1, 8, 9, 4]`. Lockr does this via a `sadd` API, as demonstrated in Example 6-3.

Example 6-3. lockr/test4.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="lockr.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {
```

```
    Lockr.set("testS", []);

    Lockr.sadd("testS", 1);
    Lockr.sadd("testS", 2);
    Lockr.sadd("testS", 3);
    Lockr.sadd("testS", 2);
    Lockr.sadd("testS", 2);
    Lockr.sadd("testS", 1);

    console.log(Lockr.get("testS"));
    console.log(Lockr.smembers("testS"));

    Lockr.srem("testS", 3);

    console.log(Lockr.smembers("testS"));

    console.log(Lockr.sismember("testS", 3));

});

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

To initialize the value, I set an empty array to the `testS` key. I then used `sadd` to add a number of values. After all those numbers are added, though, the only items in the array are 1, 2, and 3. You can use `smembers` to return all the values as well.

Next, `srem` is used to remove a value. When the members are returned again, now only 1 and 2 remain. Finally, `sismember` will return `true` or `false` if a value exists in the hash.

All in all, Lockr is a rather nice little library. Even with Web Storage being easy to use, the data handling aspects of Lockr alone are enough to interest me. When you throw in its incredibly small size, the library becomes even more appealing.

Simplifying IndexedDB with Dexie

For our next library, we'll look at Dexie (shown in Figure 6-4), a *far* simpler wrapper for the somewhat complex IndexedDB API. As with all the libraries we'll discuss in this chapter, it is 100% free and open source. You can find out more and download the library at *http://www.dexie.org*.

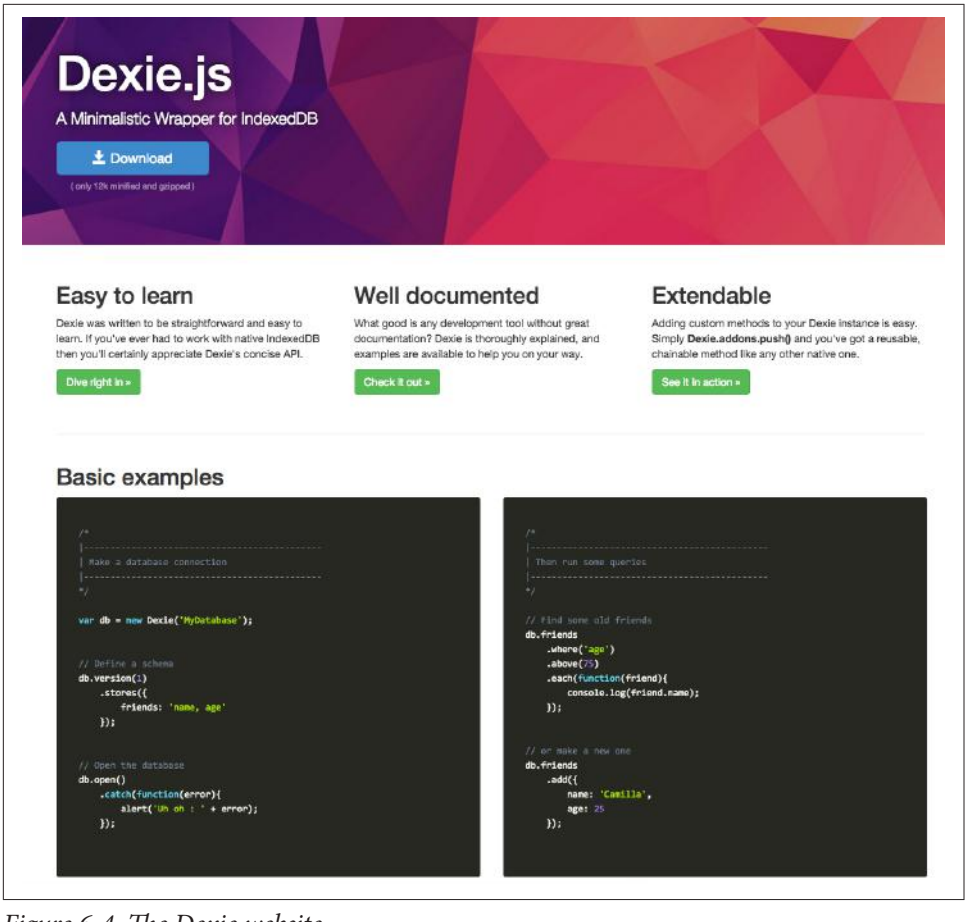

Figure 6-4. The Dexie website

With Dexie, you have not one but three different ways to install the library. You can use Bower (`bower install dexie`), use npm (`npm install dexie`), or just download the bits from GitHub.

Once you have the library loaded on your web page, you'll discover that working with Dexie is incredibly simple. For example, here is how you create a pointer to an IndexedDB database and initialize it with an object store called `notes`:

```
var db = new Dexie("name-here");
db.version(1).stores({
    notes:'text,created'
});
db.open();
```

You can probably guess that the string `"text,created"` refers to the expected properties of the data that will be stored, but Dexie goes a lot further and lets you pass sim-

ple tokens to these properties to define how they act. For example, this version will add a key called `id` that is automatically set.

```
var db = new Dexie("name-here");
db.version(1).stores({
    notes:'++id,text,created'
});
db.open();
```

In Example 6-4, you get a complete view of this in action.

Example 6-4. dexie/test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie1");
    db.version(1).stores({
        notes:"++id,text,created"
    });
    db.open();

    console.dir(db);

});

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

Altogether this doesn't do much, and the `console.dir` won't be terribly helpful because it's just an instance of a Dexie-wrapped database, but if you use your browser developer tools to look at your IndexedDB instances you'll see that a new one called `dexie1` has been created (Figure 6-5).

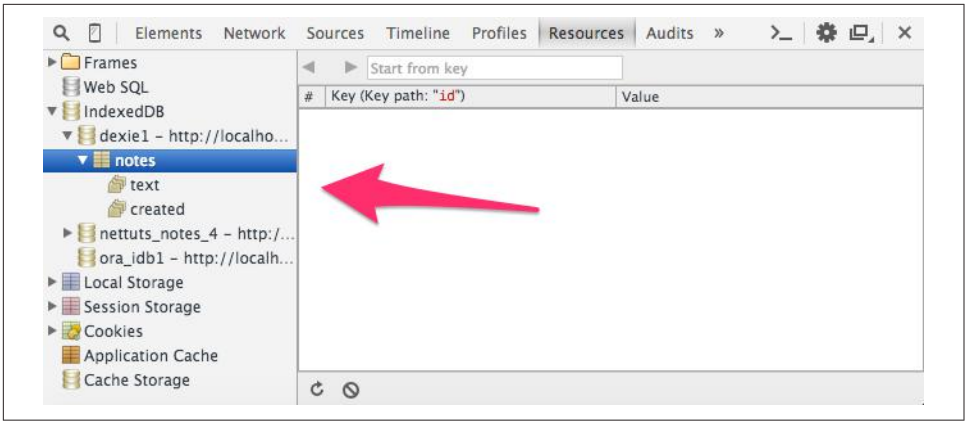

Figure 6-5. A new IndexedDB without the pain!

So far, so good, but let's look at a few basic CRUD examples. Here is how you could add data.

```
db.notes.add(
{ text:'foo', created:new Date().getTime() }
).then(function() {
 console.log('Note added.');
}).catch(function(err) {
});
```

If you are familiar with promises, then this syntax will look familiar. It's certainly simpler than the transaction API you use normally. (To be clear, Dexie is still using transactions behind the scenes, but you don't have to worry about them for simple operations. While we won't cover the topic in this simple introduction, if you want to do multiple CRUD operations in Dexie, it also has a transaction API you can use. And yes, it is still easier than the default IndexedDB API.)

How about reading? Yep—it is also simple:

```
db.notes.get(1).then(function(note) {
  console.dir(note);
});
```

Updating is slightly more complex—you pass in the key of the object you are modifying:

```
db.notes.put(
{ text:'foo', created:new Date().getTime(), key }
).then(function() {
 console.log('Note updated.');
}).catch(function(err) {
});
```

Even nicer, you can use `put` without a primary key and it will perform an insertion instead of an update. This lets you simply use `put` without switching between it and `add`.

And then finally, the `delete` operation:

```
db.notes.delete(1).then(function(note) {
  console.log("Removed");
});
```

Let's modify our previous example to add a bit of data to the database (Example 6-5).

Example 6-5. dexie/test2.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie1");
    db.version(1).stores({
        notes:"++id,text,created"
    });
    db.open();

    db.notes.add(
        { text:"foo",created:new Date().getTime() }
    ).then(function() {
         console.log("Note added.");
    }).catch(function(err) {
        console.dir(err);
    });

});

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

Now our template actually adds a bit of data. Typically, you would tie this to a form, but if you open it in your browser and check your developer tools, you'll see the freshly added data (Figure 6-6).

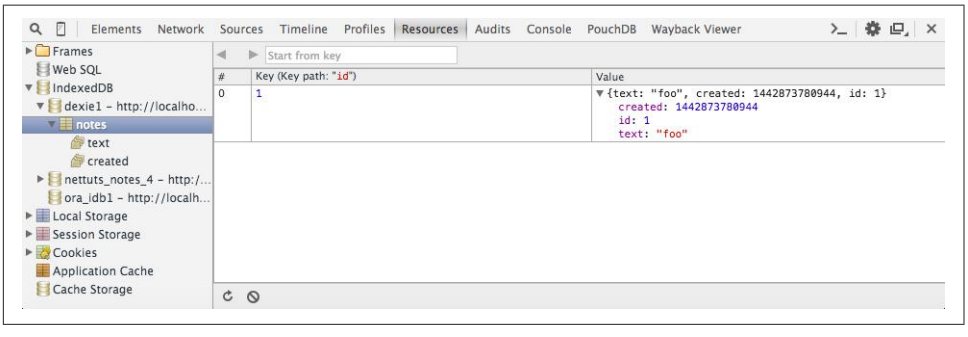

Figure 6-6. New data added by Dexie

The final piece of the puzzle is searching for data, and here is where Dexie really, *really* shines. Let's consider a simple example—finding data where some particular column (or property) has a value lower than a target:

```
db.something.where("column").below(value).each(
function(item) {
  console.log('runs for each match')
});
```

Or maybe you meant higher, not lower:

```
db.something.where("column").above(value).each(
function(item) {
  console.log('runs for each match')
});
```

Or perhaps between two values:

```
db.something.where("column").between(value1, value2).each(
function(item) {
  console.log('runs for each match')
});
```

You can find a simple demonstration of this API in Example 6-6.

Example 6-6. dexie/test3.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>
```

```
<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie3");
    db.version(1).stores({
        people:"email,name,age"
    });
    db.open();

    db.people.put({ email:"raymondcamden@gmail.com", name:"Raymond", age:43 });
    db.people.put({ email:"elric@google.com", name:"Elric", age:23 });
    db.people.put({ email:"zula@google.com", name:"Zula", age:12 });

    db.people.where("age").between(20,50).each(function(person) {
        console.log("age match",JSON.stringify(person));
    });

    db.people.where("name").anyOf(["Elric","Zula"]).each(function(person) {
        console.log("name match",JSON.stringify(person));
    });
});

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

If you run this, you may notice something interesting about the output (see Figure 6-7).

```
14:54:02.709    age match {"email":"elric@google.com","name":"Elric","age":23}
14:54:02.710    name match {"email":"elric@google.com","name":"Elric","age":23}
14:54:02.711    age match {"email":"raymondcamden@gmail.com","name":"Raymond","age":43}
14:54:02.711    name match {"email":"zula@google.com","name":"Zula","age":12}
```

Figure 6-7. Output from the Dexie search example

While the results are correct, don't forget that they are asynchronous results. That's why you see the results "intermingled" in the console. While it's not necessarily relevant to the book at hand, let's quickly demonstrate how you could handle a case like this. We mentioned earlier that Dexie supports transactions, and transactions themselves support knowing when they've completed. Example 6-7 shows a modified version of the previous template that uses a transaction to wait for the query operations to complete.

Example 6-7. dexie/test4.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="Dexie.min.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    var db = new Dexie("dexie3");
    db.version(1).stores({
        people:"email,name,age"
    });
    db.open();

    db.people.put({ email:"raymondcamden@gmail.com", name:"Raymond", age:43 });
    db.people.put({ email:"elric@google.com", name:"Elric", age:23 });
    db.people.put({ email:"zula@google.com", name:"Zula", age:12 });

    var ageResults, anyResults;
    db.transaction('r', db.people, function() {
        ageQuery = db.people.where("age").between(20,50).toArray().then(
        function(age) {
            ageResults = age;
        });
        anyQuery = db.people.where("name").anyOf(["Elric","Zula"]).toArray().then(
        function(any) {
            anyResults = any;
        });

    }).then(function() {
        console.log(JSON.stringify(ageResults));
        console.log(JSON.stringify(anyResults));
    });

});

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

Also note that we modified the second query to use the `toArray` helper. This is a utilty Dexie provides that can return the result of a query into a simple array, meaning you would not need to use the `each` method to iterate over each result.

Working with localForage

For our last and final library (but don't forget, there's more!), we'll look at localForage (shown in Figure 6-8), an open source project by Mozilla, the folks behind Firefox. localForage is an ambitious client-side storage wrapper that supports IndexedDB, Web SQL, *and* Local Storage, selecting the best mechanism it can on the fly to store data on the user's browser. You can find complete documentation, examples, and the downloads at the localForage website.

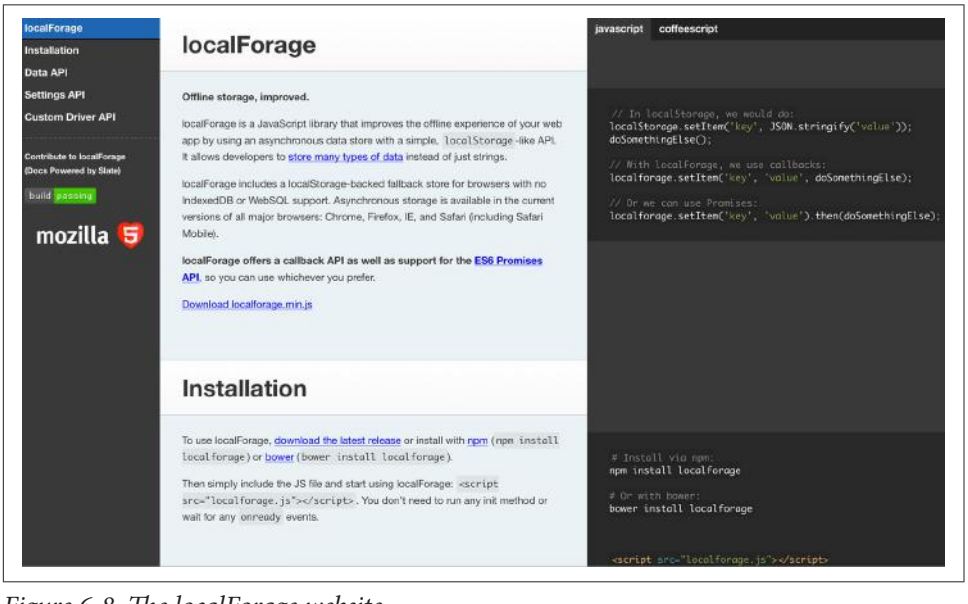

Figure 6-8. The localForage website

As with the other libraries in the chapter, you have multiple ways of installing localForage. You can use Bower (`bower install localforage`), use npm (`npm install localforage`), or just download the code from GitHub.

localForage's API is entirely asynchronous, but supports both the "old style" callback as well as "new and hot" Promise-based APIs. You can use whatever form you're most comfortable with.

As a simple example, here is how you would set a value with a callback that is executed when the value is persisted:

```
localforage.setItem("name", value, function(err, value) {
});
```

And here is the Promise-style version:

```
localforage.setItem("name", value).then(function(value) { });
```

Both do the exact same thing (OK, technically I'd need a `catch` in my second example) so you can use whatever form seems easier for you. Retrieving a value works the same way:

```
localforage.getItem("name", function(err, value) { });
localforage.getItem("name").then(function(value) { });
```

Example 6-8 gives a simple demonstration of these read and writes in action.

Example 6-8. localForage/test1.html

```
<!doctype html>
<html>
<head>
    <script type="text/javascript" src =
    "http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
    <script src="localforage.js"></script>
</head>

<body>

<script>

$(document).ready(function() {


    localforage.setItem("name", "Ray", function(err, value) {
        if(err) console.dir(err);
        console.log(value);
    });

    localforage.setItem("age", 43, function(err, value) {
        if(err) console.dir(err);
        console.log(value);

        localforage.getItem("age").then(function(value) {
            console.log("the value of age plus one is "+(value+1));
        });
    });

});

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

The first example simply sets a `name` value to `Ray`. The callback is fired when the data is successfully stored. Reporting the value back out to console is not very useful since

it (obviously) matches what we passed in. The second example stores a numeric value, and to ensure it is stored correctly, we fetch it and add 1 to the value when done.

localForage also supports APIs for removing data (`removeItem`), clearing storage (`clear`), counting the number of keys (`length`), and fetching all the keys (`keys`). You can also iterate over all the key/value pairs with a simple `iterate` call:

```
localforage.iterate(function(value, key, index) {

}, callback);
```

As we said earlier, localForage attempts to use the "best" storage method it can on the current browser. By default, localForage will first try IndexedDB, then Web SQL, and finally Local Storage. What's cool is that you can actually request a different priority. The `setDriver` API lets you specify which system you wish to use, or an array of systems you want to use in order. Here is an example that prefers Web SQL over IndexedDB:

```
localforage.setDriver(localforage.WEBSQL, localforage.INDEXEDDB);
```

Note that you do not need to specify Local Storage. If localForage cannot find your preferred storage system, it will fall back automatically.

localForage does not provide any query or search capabilities, so keep that in mind before adopting this particular library. This means localForage is better suited for simple storage needs, like large data sets you wish to retrieve via key rather than via some ad hoc search.

More Options

As we've said multiple times, we've covered only a few client-side storage libraries. Here are a few more you may want to consider checking out.

- PouchDB is an incredibly powerful option. In fact, it was so powerful I was worried it was a bit too much to cover in a short form here in the chapter. The developers behind this library have done a huge amount of work in the area of client-side storage and are well-known experts in the field. One of the biggest features of PouchDB that may entice you is that it supports data synchronization.
- lawnchair is an older library that also supports multiple storage methods via an adapter API.
- And finally, you can also peruse libraries via this helpful list created by Juho Vepsäläinen.

CHAPTER 7

Building a Sample Application

Let's Build Something!

Now that you've seen multiple types of client-side storage techniques as well as some libraries to help make using them easier, let's build a real, if simple, application that makes use of some of these techniques. Our application will be a tool for a company intranet ("Camden Incorporated"—coming to the NYSE soon) that lets you search for your coworkers. This could be built using a traditional application server model, but we've decided to build something fancy using modern web standards. To make the search near instantaneous, we'll use client-side storage to keep a copy of the employee database on the user's browser. This, of course, opens up all kinds of interesting issues.

First off, how do we handle synchronization? Companies aren't static. People join or leave companies all the time. How often that happens, of course, depends on the company itself, but obviously you have to consider some form of strategy for keeping the user's copy of data in sync with the real list on the server. Luckily, in our scenario we don't have to worry about user edits. The server side is always "truth," which means we can ignore changes on the client side when syncs happen. For our demo we're not going to worry about syncing at all, but in a real-world demo your application server could provide an API where the client says—and by "says" I mean via code, of course—"My copy of the data was last updated on October 10, 2015 at 8:55 AM." The server could then respond with a set of changes that have occurred since that date. Those changes could cover deletions (people who left the company), changes (people getting married and changing their name, or getting new titles), and additions (new hires). The client-side code would apply those changes and then make a note of the current time so that the next time it speaks to the server it can correctly receive the changes.

The next issue is a thorny one: privacy. The company database probably has a good deal of data about you that you don't want to share—like your salary. Remember that we are essentially sending private information to each employee, and while you may trust your employees, you still can't send information that could put their privacy at risk. A safe metric might be, "If it is on their business card, share it," but certainly you want to be *overly* cautious here. And to be clear, you cannot "filter" out the insecure data on the client side. If your app server is returning private data, anyone can clearly see it by opening up their browser developer tools. Anything the browser gets is open to inspection by the user. As a matter of habit I tend to browse the Web with my browser tools open, and I'll naturally look at Ajax calls and the data just for curiosity's sake. I'm a "good guy," but you have to assume that the "not-so-good guys and gals" are looking as well.

The last issue is performance. Given a "small" company of 10,000 people, how do you handle transferring that data to the browser in a performant matter? We've said our hypothetical situation here is a company intranet, so we're already kind of assuming desktop/LAN, but you'll want to be cognizant of the size of your data packets going to the client. We'll discuss a way to handle this later in the chapter.

OK, let's talk data!

Our Sample Data

To keep things as easy as possible, our "server" will be a simple JSON file of data. As we said earlier, we are not going to work with synchronization and creating updates, so a flat JSON file will serve our needs just fine. To make things even easier, we're going to use a cool, free web service to generate our data: the Random User Generator, shown in Figure 7-1.

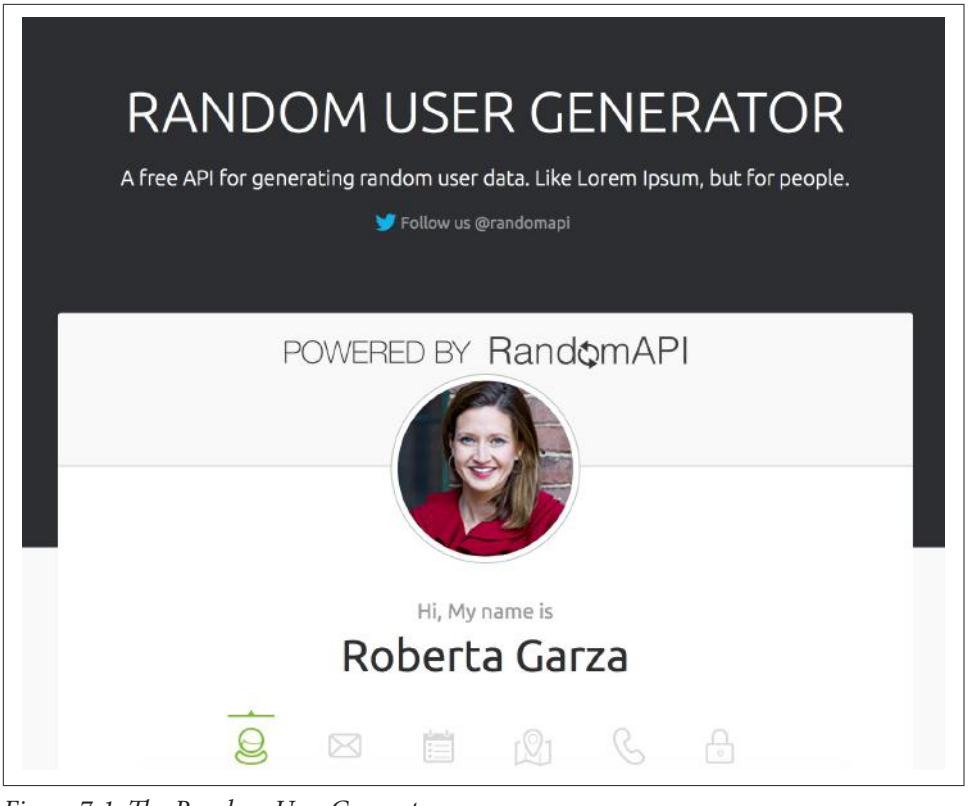

Figure 7-1. The Random User Generator

This site provides a free API that returns user information. The user information includes quite a bit of detail and can be useful for demos like the one we're building here. Example 7-1 is a sample of the output taken from their docs.

Example 7-1. Sample API result

```
{
  results: [{
    user: {
      gender: "female",
      name: {
        title: "ms",
        first: "manuela",
        last: "velasco"
      },
      location: {
        street: "1969 calle de alberto aguilera",
        city: "la coruña",
        state: "asturias",
        zip: "56298"
```

```
      },
      email: "manuela.velasco50@example.com",
      username: "heavybutterfly920",
      password: "enterprise",
      salt: ">egEn6YsO",
      md5: "2dd1894ea9d19bf5479992da95713a3a",
      sha1: "ba230bc400723f470b68e9609ab7d0e6cf123b59",
      sha256: "f4f52bf8c5ad7fc759d1d415e508aa0b7946d4ba",
      registered: "1303647245",
      dob: "415458547",
      phone: "994-131-106",
      cell: "626-695-164",
      DNI: "52434048-I",
      picture: {
        large: "http://api.randomuser.me/portraits/women/39.jpg",
        medium: "http://api.randomuser.me/portraits/med/women/39.jpg",
        thumbnail: "http://api.randomuser.me/portraits/thumb/women/39.jpg",
      },
      version: "0.6"
      nationality: "ES"
    },
    seed: "graywolf"
  }]
}
```


While the API is incredibly easy to use, we want a static set of data for our demo. If you sign up at RandomAPI, you get permission to use the random user API for up to 10,000 results. The RandomAPI site is—as you can imagine—a collection of APIs that provide random data. All in all, both sites are really darn useful and you can use them within your own applications as well. It is a great way to work with "sensible" random data while building your application.

For this demo, I signed up and requested 10,000 users. In the zip file of sample code from this book, you can find it in *c7/data/users.json*. Earlier in this chapter we discussed how you would want to be careful about what data you expose in your application. Since we're just taking the random user data as is, we *definitely* have information here that we would absolutely *not* want to share. Not only that, but our demo is only going to use about half of the user values present in the data, which means quite a bit of wasted data will be sent from the server to the frontend. These are all things you would want to be very cognizant of in a proper, production-ready application. But imagine for a moment that we've done that. We've streamlined our API down to the bare essentials required to meet the application's needs. What else can we do to make the data load quicker?

One simple method is GZip compression. This is a setting your web server can use to enable zip compression of assets before they are sent to the browser. The web server is intelligent enough to use this feature only when the browser sends a header saying it supports it, and since almost all modern browsers support it, this is an "easy win" to

help speed up your transfers. Apache, especially, makes it fairly trivial to enable. How much does it help?

Our *users.json* file is 13.5 MB. That isn't small. Poorly optimized graphics probably won't go over a megabyte each, so you're really looking at a big hit here to download that file. Figure 7-2 shows Chrome reporting on the size of the JSON file when it's requested via the browser. This is before any compression is added.

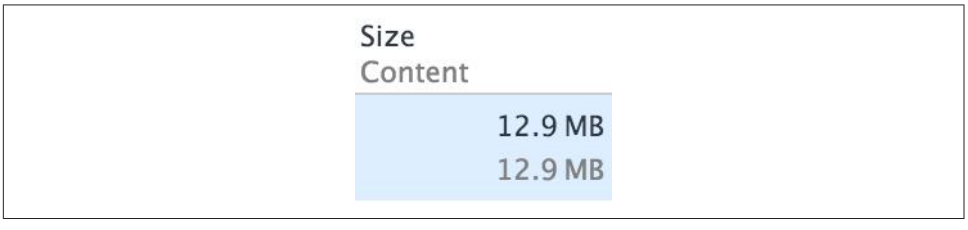

Figure 7-2. Chrome's report of the file request

And Figure 7-3 shows the size after compression is enabled in Apache.

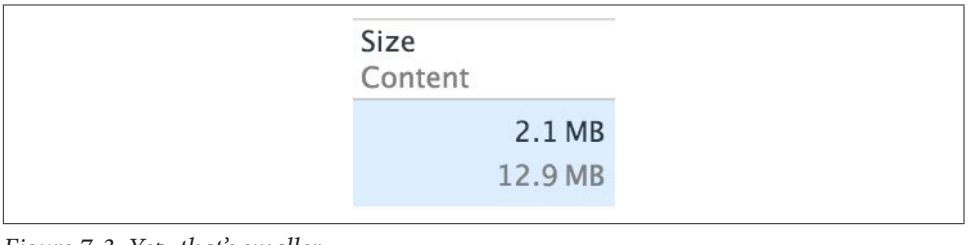

Figure 7-3. Yep, that's smaller

The difference is pretty staggering. Keep in mind that the browser still has to decompress that file on the client side, so you'll want to use this approach with caution. I still wouldn't recommend sending more than 10 MB of data over the wire. At least in our case this is an initial, "worst case" load, and later calls—again using an imaginary application server—would send only the changes.

Now that you've seen the data in play, let's look at the finished application.

The Application

When you first hit the application, it will fetch the initial data set (that large JSON file) and begin inserting it into a local data store. Since this can take a little while, a modal window is used to let the user know what's going on. For the application it is kept rather simple—just one message (Figure 7-4). You could enhance this messaging to report on whether or not the application is downloading the initial data or has moved on to inserting it for storage locally.

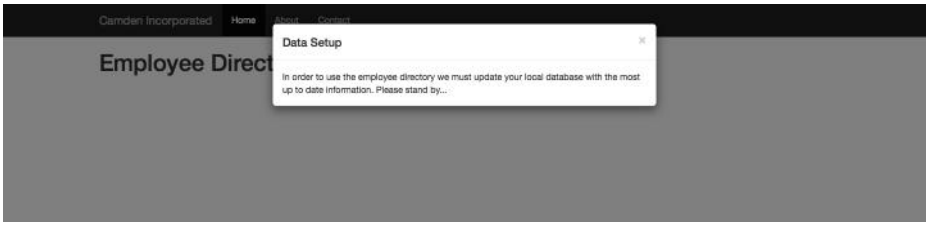

Figure 7-4. A message is displayed while the application is setting up

After everything is loaded, a basic form, shown in Figure 7-5, is presented to the user. In this application, you can search only by first and last name.

Figure 7-5. The search form

You can then begin searching. You can search on just the first name, last name, or both. Figure 7-6 shows some sample search results.

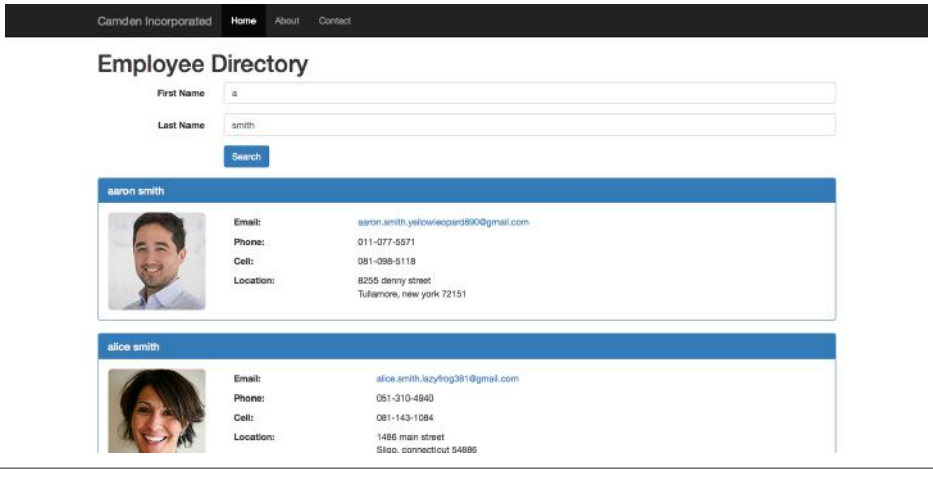

Figure 7-6. Search results

The application will also correctly let you know when nothing is found. All in all, this is a rather simple interface. You could add more filters (like business department or

managers) to further enhance the searching later. In case you're curious, those pictures all come from the Random User API itself.

Now that we've discussed our data and demonstrated how the application looks, let's begin looking at the code behind it.

The Code

The application will make use of Local Storage and IndexedDB. Local Storage will just be used as a way to remember if data has been loaded. IndexedDB will store the data itself. For Local Storage, even though the usage will be rather trivial, we'll use Lockr. For IndexedDB, we'll use Dexie to simplify both inserting data as well as searching.

Let's begin by discussing how we'll determine if data has been cached on the client side. Example 7-2 demonstrates the function written to determine if local data exists.

Example 7-2. haveData function

```
function haveData() {
    var def = $.Deferred();

    var lastFetch = Lockr.get("lastDataSync");

    if(lastFetch) def.resolve(true);
    else def.resolve(false);

    return def.promise();
}
```

There are a couple of interesting things going on here. On line one, a deferred object is created so that a promise can be returned. This allows us to use the function in an asynchronous matter. We're not actually using an asynchronous process, though. As we learned, Local Storage access is synchronous, but in the future we may update the process so that it becomes asynchronous. The code calling this function won't have to change.

For now, our code simply uses Lockr to check for a `lastDataSync` property. If this exists, then we have data. It's going to be a date value we set later, with the idea being that if you hook this code up to a "real" app server in the future, the date would be valuable information to use in determining what new data you need. Let's look at how this code is called (Example 7-3).

Example 7-3. Calling haveData

```
haveData().then(function(hasData) {

    if(!hasData) {
        console.log("I need to setup the db.");
        setupData().then(appReady);
    } else {
        appReady();
    }

});
```

The code calling `haveData()` uses the `then` method of the returned promise to set up what is going to run when the asynchronous process is done. Yes, it really isn't asynchronous, but as we've said, the caller doesn't need to worry about that. If there is no data, then a call to set up the data will be fired; otherwise, the code will run a function signifying that the search application is ready to go. Let's look at the data setup function.

First, Dexie requires us to create an IndexedDB database and define the object store that will store data (Example 7-4). This is done earlier in the code within the `$(document).ready` block.

Example 7-4. IndexedDB setup

```
myDb = new Dexie("employee_database");
myDb.version(1).stores({
    employees:"++id,&email,name.first,name.last"
});
myDb.open();
```

As demonstrated in the previous chapter, Dexie goes a *long* way to simplifying IndexedDB usage. You can see the database being created as well as the `employees` object store. The `employees` store has an autoincrementing number `id`, a unique index on `email`, and indexes on `name.first` and `name.last`. These indexes were created based on how we plan on searching for employees. Now let's move to the function run to set up data (Example 7-5).

Example 7-5. setupData function

```
function setupData() {
    var def = $.Deferred();

    //setup modal options
    $("#setUpModal").modal({
```

```
    keyboard: false
});

//now show it
$("#setupModal").modal("show");

//now, fetch the remote data
$.get("data/users.json", function(data) {
    console.log("Loaded JSON, have "+data.results.length+" records.");
    console.dir(data.results[0].user);

    myDb.transaction("rw", myDb.employees, function() {

        data.results.forEach(function(rawEmp) {

            /*
            We aren't copying the data as is, we modify it a bit.
            Specifically the raw data has some dupes on email/username
            so I make an email based on the compation of both
            */
            var emp = {
                cell:rawEmp.user.cell,
                dob:rawEmp.user.dob,
                email:rawEmp.user.email.split("@")[0]+"."
                + rawEmp.user.username + "@gmail.com",
                gender:rawEmp.user.gender,
                location:rawEmp.user.location,
                name:rawEmp.user.name,
                phone:rawEmp.user.phone,
                picture:rawEmp.user.picture
            };

            myDb.employees.add(emp);
        });

    }).then(function() {

        //hide the modal
        $("#setupModal").modal("hide");

        //store that we synced
        Lockr.set("lastDataSync", new Date());

        def.resolve();

    }).catch(function(err) {
        console.log("error in transaction", err);
    });

}, "json");

return def.promise();
```

```
}
```

This one, as you can imagine, is the big one. As with the `haveData` function, jQuery `Deferred`s are used to handle the asynchronous nature of the setup. Ignoring the UI items (Bootstrap makes this so easy), the real meat begins with an Ajax call to the JSON file. Once the file is retrieved, a transaction is opened via Dexie. For each user in the JSON data, we need to create a new object that will be stored. In an ideal world, our data would match what we want to store exactly, but since we're working with data from the Random User API, we need to manipulate it a bit. If you are setting up an app server to feed your code data like this, you will want to try to make it match as best you can. Note that the email address is modified a bit, as the Random User data did not have unique email addresses. This is most likely just a bug on their side, and it was quicker to work around it in the JavaScript. This object is added and—that's it. When the transaction finishes, the UI is updated again and the earlier deferred is resolved. Note that the final step is to update Local Storage, again via Lockr, with the current time.

Now let's turn to search. After the data is loaded, or it was determined that the data already existed, `appReady` is executed as shown in Example 7-6.

Example 7-6. appReady function

```
function appReady() {
    console.log('appReady fired, lets do this');
    //show the search form
    $("#searchFormDiv").show();
    $("#searchForm").on("submit", doSearch);
}
```

There isn't much here. Basically the form is displayed and an event handler for running that search is registered. Let's look at that in Example 7-7.

Example 7-7. doSearch function

```
function doSearch(e) {
    e.preventDefault();
    var fName = $.trim($firstNameField.val());
    var lName = $.trim($lastNameField.val());

    $results.empty();
    console.log('search for -'+fName+'- -'+lName);

    var fnEmps = [];
    var lnEmps = [];
    myDb.transaction('r', myDb.employees, function() {
```

```
    if(fName !== '') {
        myDb.employees.where("name.first").startsWithIgnoreCase(fName)
        .each(function(emp) {
            fnEmps.push(emp);
        });
    }

    if(lName !== '') {
        myDb.employees.where("name.last").startsWithIgnoreCase(lName)
        .each(function(emp) {
            lnEmps.push(emp);
        });
    }

}).then(function() {
    console.log('done');
    var results = [];

    //just a first name
    if(fName !== '' && lName === '') {
        console.log('first');
        fnEmps.forEach(function(emp) { results.push(emp); });
    //just a last name
    } else if(lName !== '' && fName === '') {
        lnEmps.forEach(function(emp) { results.push(emp); });
    //both
    } else {

        //only return items where ob exists in both
        //to make it simpler, we'll make an index of
        //email values in lnEmps so we can check them
        //quicker while going over fnEmps
        var lnEmails = [];
        lnEmps.forEach(function(emp) { lnEmails.push(emp.email); });

        results = fnEmps.filter(function(emp) {
            return lnEmails.indexOf(emp.email) >= 0;
        });
    }

    //Begin rendering the results.
    if(results.length) {
        results.forEach(function(r) {
            console.log(r.name.first+' '+r.name.last);
            var result = resultTemplate(r);
            $results.append(result);
        });
    } else {
        $results.html("Sorry, nothing matched your search.");
    }

}).catch(function(err) {
```

```
        console.log('error', err);
    });

}
```

This is another large one, so let's take it bit by bit. First, the current fields in the search form are retrieved and trimmed. Once we have those fields, the search can begin. Unfortunately, we can't search for both values in one call, but we can do them both in one transaction. So a transaction is opened and then a search, again using the nice functions Dexie provides, against the first and last name indexes. For each search, the results are placed in an array.

When the transaction is done, that means both searches (or one if only one search field was used) are finished. We then have to merge the results. If you didn't use both fields, this is a simple matter: the result array for the field you searched for is copied to a results array.

If you used both fields, it is a bit more complex. We want to return results that exist in both arrays. The `lnEmps` array is looped over to create a simpler array of just email addresses. This then lets us loop over the `fnEmps` array and accept only those values where a corresponding email address in the last name result exists as well.

Finally, the results are ready. But how to display them? To make it simpler to dynamically write out content in the template, we'll use Handlebars as a client-side templating language. Handlebars lets us define a template with variable tokens. We can load this template, automatically replace the tokens, and then render out HTML. The template for handling results is defined in *index.html* (see Example 7-8).

Example 7-8. Result template

```
<script id="result-template" type="text/x-handlebars-template">
<div class="panel panel-primary">
    <div class="panel-heading">
        <h3 class="panel-title">{{name.first}}{{name.last}}</h3>
    </div>
    <div class="panel-body">
        <div class="row">
            <div class="col-md-2">
                <img src="{{picture.medium}}" class="img-rounded">
            </div>
            <div class="col-md-10">
                <table style="width:100%">
                    <tr>
                        <td><b>Email:</b></td>
                        <td><a href="mailto:{{email}}">{{email}}</a></td>
                    </tr>
                    <tr>
                        <td><b>Phone:</b></td>
```

```
                    <td>{{phone}}</td>
                </tr>
                <tr>
                    <td><b>Cell:</b></td>
                    <td>{{cell}}</td>
                </tr>
                <tr valign="top">
                    <td><b>Location:</b></td>
                    <td>{{location.street}}<br/>
                    {{location.city}}, {{location.state}} {{location.zip}}</td>
                </tr>
            </table>
        </div>
    </div>
  </div>
</div>
</script>
```

In the preceding listing, you can see each token as a value wrapped in double curly braces, {{ and }}. Handlebars can process these tokens and replace them with the actual result data from our search. What's nice about a client-side templating language is that it makes it much easier to generate dynamic output from JavaScript.

Wrap-up

You can find the complete code for the demo in the zip file you downloaded, and I strongly encourage you to play with it yourself to see what changes you could make. More search filters could be added, and if you're really motivated you could set up an application server to start working on a "send only what's changed" API. Good luck!

Index

A
appReady function, 100
arrays, 54-59

B
browser tools (see developer tools)

C
CanIUse.com, 11
Chrome developer tools
    cookies, 11
    IndexDB, 59
    Web SQL, 72-73
    Web Storage, 23
compression, 94-95
cookies, 3-12
    basics, 3-4
    deleting, 6
    demos, 6-10
    developer tools, 10-11
    reading, 5-6
    recommendations for use/non-use, 12
    support, 11
    working with, 4-6
CRUD, 37, 48, 83-85
cursors, 48-50

D
databases, 28, 29-31, 64
developer tools
    cookies, 10-11
    IndexDB, 59
    Web SQL, 72-73
    Web Storage, 23
Dexie, 80-87, 97-103
    CRUD examples, 83-85
    searching for data, 85-86
doSearch function, 100-102

F
Firefox developer tools
    cookies, 10
    IndexDB, 59
    Web Storage, 23-23
form data, 17-19

G
GZip compression, 94-95

H
Handlebars, 50, 102-103
hash values, 79
headers, 3
HTTP headers, 3
httpster, 7

I
IndexDB, 27-61, 97-103
    add method, 38
    blocked events, 29
    continue method, 49
    createIndex method, 37
    createObjectStore, 32, 37
    data
        counting, 59
        creating, 38-42

deleting, 47-48
reading, 42-44
updating, 45-47
data retrieval
cursors and, 48-50
ranges and indexes, 51-54
databases and, 28, 29-31
developer tools for, 59
Dexie for (see Dexie)
error events, 29
IDBKeyRange, 51
and iOS8, 29
object stores, 28, 31-37
only method, 59
put method, 45
recommended uses, 61
someObjectStore.delete, 47
storing arrays, 54-59
success events, 29
support, 29, 61
terminology, 28
upgradeneeded events, 29, 31, 33-36, 42
indexes, 28, 36, 37, 51
iOS8, and IndexDB, 27, 29

J

JSON encoding, 14

K

key generators, 34
key paths, 34

L

lastDataSync property, 97
lawnchair, 90
libraries, 75-90
Dexie, 80-87
list of, 90
localForage, 88-90
Lockr, 75-80
PouchDB, 90, 90
Local Storage, 13, 97-103
(see also Web Storage)
localForage, 88-90
Lockr, 75-80, 97-103
hash values, 79
versus basic Web Storage, 77

M

MDN (Mozilla Developer Network) code, 6
Mozilla localForage (see localForage)

N

Netscape, 3

O

object stores, 28, 31-37
counting data in, 59
creating, 32-36
primary keys, defining, 34-36
version numbers and, 31

P

performance issues, 92, 94
PouchDB, 90
primary keys, 34-36, 50
privacy issues, 92

R

Random User Generator, 92
RandomAPI, 94
ranges, 51-54
Redis, 75

S

sample application, 91-103
Session Storage, 13, 16
setupData function, 98-100
SQL injection attack, 71
storage event, 19-22
string data, 14
synchronization issues, 91

T

tables, 64
transactions, 66-72

V

version numbers, 31

W

Web SQL, 63-73
databases, 64-66
developer tools for, 72-73

reasons for learning, 63
rows, 64
SQL injection attack, 71
support, 64, 73
tables, 64, 67-68
terminology, 64
transactions, 66-72
Web Storage, 13-25
clear method, 14
converting data into JSON, 14
demos, 15-19
event listeners, 18
form data, 17-19
getItem method, 14
limits, 13
Lockr for (see Lockr)
methods, 14
recommended uses, 25
removeItem method, 14
setItem method, 14
storage event, 19-22
support, 24-24
supported data, 14
web storage

About the Author

Raymond Camden is a developer advocate for IBM. His work focuses on the Mobile-First platform, Bluemix, hybrid mobile development, Node.js, HTML5, and web standards in general. He's a published author and presents at conferences and user groups on a variety of topics. Raymond can be reached at his blog (*www.raymondcamden.com*), @raymondcamden on Twitter, or via email at *raymondcamden@gmail.com*.

Colophon

The animal on the cover of *Client-Side Data Storage* is the unstriped ground squirrel (*Xerus rutilus*). The unstriped ground squirrel is native to the arid savanna and shrubland found in the Horn of Africa. As ground squirrels, they make their homes in subterranean burrows.

Unstriped ground squirrels have brown fur, with darker backs and lighter fronts. As their name would suggest, they lack the white-striped backs common in other African ground squirrel species. Unstriped ground squirrels can grow up to one pound in weight and 25 centimeters in length, with their tails growing an additional 25 centimeters.

Unstriped ground squirrels have an omnivorous diet, consisting of leaves, fruit, seeds, and insects. They spend most of their time foraging for food, and only return to their burrows to sleep. The main predators for unstriped ground squirrels are birds of prey, leopards, jackals, and snakes.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Lydekker's Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.