5 Mobile applications: client storage and offline execution

This chapter covers

- Storing data on the client side with the Web Storage API
- Managing a full client-side IndexedDB database
- Enabling applications to work offline with the Application Cache API

HTML5 is finally providing the web with a solution to the problem of working offline. Although a plethora of solutions for saving web pages for later use in an offline environment already exist, until now there's been no solution for using web applications in such a manner. By allowing web applications to store data locally on the client, HTML5 now enables web applications to work without a constant connection to a central server.

When might this be useful? Think of a sales representative in the field being able to use his firm's customer relationship management application on the go, even in areas with poor network coverage, such as a remote location or an underground train. With the new capabilities provided by HTML5, that rep can still use the application in such areas, viewing data that has already been downloaded to the device, and even being able to enter new data, which is stored temporarily on the device and synchronized back to the central server when the network is available

again. Also, think of an HTML5 game like the ones you will build in chapters 6 and 7. Rather than storing game saves and state data on a server, you can increase performance and reduce latency and load by saving the data locally. One feature in particular—the application cache manifest—gives you the ability to create a game that can be run completely offline.

In this chapter, we're going to show you how to put these features and concepts into practice by building a simple mobile web application called My Tasks. This application, which will be fully functional when the user is offline, will create, update, and delete tasks that are stored locally in the browser. In addition, My Tasks will allow the user to change settings for the application's display.

Why build the sample My Tasks application?

While working through this chapter's sample application, you'll learn how to

- Store data on the client side using the Web Storage API
- Store data on the client side using the IndexedDB database
- Use the application cache manifest file to build web applications that will function while offline

Let's get started by taking a closer look at the sample application.

5.1 My Tasks: application overview, prerequisites, and first steps

My Tasks is a simple task management application for mobile devices. All data will be stored on the client side, and the application will be fully functional offline. In building it, you'll take advantage of the following HTML5 features:

- *Storage*—Allows the app to save small amounts of data to the user's local storage. My Tasks will use this feature to store user settings like name and preferred color scheme.
- *Indexed database (aka IndexedDB)*—Enables the application to create a database of key/value records. My Tasks will use IndexedDB to store task data, allowing users to easily view, add, update, and delete task items. The application will use the now-defunct Web SQL to provide a fallback for devices that don't yet support IndexedDB.
- *Application cache manifest*—Enables the application to be used offline. The cache manifest ensures that the user's browser keeps a copy of needed files for offline use. Upon reconnection to the web, the browser can look for updates and allow the user to reload the application and apply the updates.

As you can see in figure 5.1, the application is split into three distinct views—Task List, Add Task, and Settings.

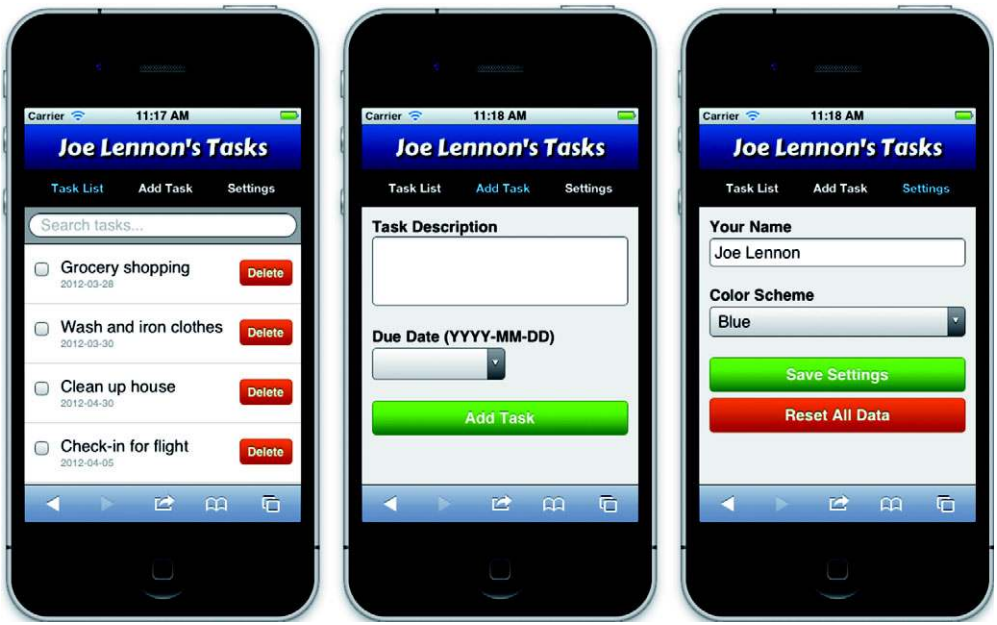

Figure 5.1 The three main views of the My Tasks application: Task List, Add Task, and Settings. To select a view, the application includes a navigation bar near the top.

Task List displays a list of existing tasks, each with a check box to mark the task as completed and a Delete button to remove it. Task List also features a search box, which allows you to filter the task list by description. Add Task contains a form to add a new task to the database. Settings contains a form to customize the application and to reset all locally stored data (deleting all Storage data and IndexedDB/Web SQL data). The navigation bar at the top of the screen lets you easily switch among the three views.

All three views are contained in a single HTML page, and you will use `location.hash` to switch among them, ensuring the application is highly responsive and fast.

We'll walk you through seven major steps to build the application:

- Step 1: Create the basic structure of the application: the HTML page with the application's three views and the JavaScript code to navigate among them.
- Step 2: Implement the data management of the Settings view using the Web Storage API.
- Step 3: Connect to the database and create a storage area for tasks.
- Step 4: Enable data entry and search of the Task List view using the IndexedDB API.
- Step 5: Allow users to add, update, and delete tasks.
- Step 6: Create a cache manifest file to allow the application to work offline.
- Step 7: Implement automatic updating of the application.

NOTE The application should be run from a web server rather than the local filesystem. Otherwise, you won't be able to use it on a mobile device and offline support won't work. Also note that the application has been tested on iOS, Android, and BlackBerry Torch mobile devices, as well as on Opera Mobile. It's also fully functional in the Chrome, Firefox, Safari, and Opera desktop browsers.

If you're looking for a quick and easy way to set up a web server for this chapter's application, we suggest you try Python's built-in server, http.server. You can get this server module by downloading and installing the latest version of the Python programming language from http://python.org/download/. Once you have it installed, you can start the server by changing your current directory to the directory of your web app and then invoking the web server with the following command:

```
python -m http.server
```

Python's web server will start running on port 8000. If you don't like the default 8000 port, you can specify another port by adding the desired port number at the end of the `python` command:

```
python -m http.server 8080
```

In this section, you'll define the application's HTML structure, use CSS to define visibility for each view, and write the JavaScript to implement navigation between the views. For the development of the My Tasks basic structure, the process consists of four steps:

- Step 1: Define the top-level HTML structure.
- Step 2: Write HTML code for the navigation bar.
- Step 3: Create views with `<section>` elements.
- Step 4a: Enable navigation between views by using CSS to define section visibility rules.
- Step 4b: Enable navigation between views by using JavaScript to initiate view changes.

Prerequisites

Before you create the application, you need to handle a few prerequisites:

- Create a new directory on your web server. When the chapter tells you to create or edit a file, save it to this directory.
- You won't be creating the CSS style sheet. Instead copy the CSS style sheet for chapter 5 from the code package at the book's website: www.manning.com/crowther2; then save the style sheet to the directory mentioned in the first prerequisite.

Note that all files for this chapter and the book are available at the Manning website: www.manning.com/crowther2.

Enough chatter about what you're going to build, let's get building!

5.1.1 Defining the HTML document structure

In this section, the index.html file will define a very basic <head> and <body> framework for the application. The index.html file will contain a title and font for the application, as well as a <script> element to tell the application where the JavaScript file is located. Near the end of index.html, a <body> element will be added to hold the HTML markup coming in subsequent sections.

STEP 1: DEFINE THE TOP-LEVEL HTML STRUCTURE

Create a file named index.html and include the contents of the following listing. This code defines the basic layout of the page and loads external CSS and JavaScript files.

Listing 5.1 index.html—Application HTML structure

```
<!DOCTYPE html>
<html lang="en" class="blue">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
        initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
    <title>My Tasks</title>
    <link rel="stylesheet"
        href="http://fonts.googleapis.com/css?family=Carter+One">
    <link rel="stylesheet" href="style.css">
    <script src="app.js"></script>
</head>
<body class="list">
</body>
</html>
```

A class attribute on the root element defines the color scheme. Later in the chapter, the application will use this attribute to allow the user to change the color scheme.

Load a custom font using the Google Font API.

A class attribute on the <body> element will direct the browser, via a CSS rule, to display one of three views: Task List, Add Task, or Settings. The class attribute also directs the browser, via another CSS rule, to highlight the corresponding button on the navigation bar.

STEP 2: WRITE HTML CODE FOR THE NAVIGATION BAR

This code comprises a <nav> element with three list items, one for each view in the application: Task List, Add Task, and Settings. Add the navigation bar's HTML code in the next listing within the <body> of your HTML document.

Listing 5.2 index.html—Adding a navigation bar

```
<header>
    <h1><span id="user_name">My</span> Tasks</h1>
    <nav>
        <ul>
            <li><a href="#list" class="list">Task List</a></li>
            <li><a href="#add" class="add">Add Task</a></li>
            <li><a href="#settings" class="settings">Settings</a></li>
        </ul>
    </nav>
</header>
```

Create a list with three links, each pointing to a hash reference for the view in question.

In the Settings view, the user has the option to replace the "My" with any other string of characters. The markup surrounding "My" will make finding and changing the title easy.

STEP 3: CREATE VIEWS WITH <SECTION> ELEMENTS

The final part of the HTML page uses `<section>` elements to define the application's three views. The first view, Task List, contains a search form and a results list, which will be generated by a JavaScript function. The next view, Add Task, contains a form that allows the user to create a new task and due date. The last view, Settings, contains a form to set the name and color scheme preference for the application. A class attribute bound to each `<section>` element will allow the forthcoming CSS and JavaScript code to control the view's visibility. Insert the code in the following listing directly after the code from listing 5.2.

Listing 5.3 index.html—Main application views

```
<section class="list">
    <form name="search">
        <input type="search" name="query" placeholder="Search tasks...">
    </form>
    <ul id="task_list"></ul>
</section>
<section class="add">
    <form name="add">
        <label>
            Task Description
            <textarea name="desc"></textarea>
        </label>
        <label>
            Due Date (MM/DD/YYYY)
            <input type="date" name="due_date">
        </label>
        <input type="submit" value="Add Task">
    </form>
</section>
<section class="settings">
    <form name="settings">
        <label>
            Your Name
            <input type="text" name="name">
        </label>
        <label>
            Color Scheme
            <select name="color_scheme">
                <option>Blue</option>
                <option>Red</option>
                <option>Green</option>
            </select>
        </label>
        <input type="submit" value="Save Settings">
        <input type="reset" value="Reset All Data">
    </form>
</section>
```

This form allows the user to search the Task List by task description.

Place the results of the search in an empty unordered list with the ID "task_list."

Put a form in the Add Task section that allows users to add a new task to the list. The form contains a task description <textarea> and a due date <input>.

In the Settings section, create a settings form that allows users to set their name and choose a color scheme for the application (red, blue, or green).

Use an <input> element to implement a button that resets user settings and removes all tasks.

5.1.2 Controlling visibility of views using CSS

Now that you have the three views implemented in one HTML file, you need the ability to switch among the different views. You will do this by turning off the visibility of the previous view and turning on the visibility of the next view. (You won't need to make these changes to the CSS file, because you should have already copied the Manning-supplied CSS file to your server's directory. See "Prerequisites.")

STEP 4A: ENABLE NAVIGATION BETWEEN VIEWS BY USING CSS TO DEFINE <SECTION> VISIBILITY RULES

In order to have only one view visible at a time, the application's CSS file defines rules to control the visibility of each view's `<section>` element:

```
section {
    display: none;
}
```

The first rule declares that a `section` element should be invisible wherever a `section` element is defined.

In order to make a specific view visible, the application defines some counteracting rules:

```
body.list section.list,
body.add section.add,
body.settings section.settings {
    display: block;
}
```

These rules declare that a `<section>` element should be visible when a `<body>` element and its embedded `<section>` element have a `class` attribute in common (either `list`, `add`, or `settings`). In this situation, the `<section>` element would also match the first rule, but the more specific rule will override the first rule.

To see how this works, consider what happens when the user wants to switch views. When the user taps the Add Task button on the navigation bar, the application changes the `<body>`'s `class` attribute to `add`. Because the `<body>`'s `class` attribute now matches the `<section>` with a `class` attribute of `add`, the `section.add` element becomes visible, and all other `<section>`s are rendered invisible.

The CSS rules only get you part of the way toward implementing the navigation of the views. Although the CSS rules declare the conditions for switching views, the rules can't initiate the view switching. As mentioned earlier, a user's tap of a button on the navigation bar initiates the view switch by changing the `class` attribute of the `<body>` element. The next section describes how to implement this attribute change and link it to the view buttons.

5.1.3 Implementing navigation with JavaScript

In this section, you'll use JavaScript to modify the `class` attribute of the `<body>` element. Each time the `class` value is changed to a different value, one or more CSS rules will be activated to change the application's view. The user will initiate these changes by tapping one of three buttons: Add Task, Settings, or Task List. Each button

is implemented as a link with an anchor name of `#add`, `#settings`, or `#list`. So when a link is selected, it will change the `location.hash` property to one of the three anchor names. The browser will detect the change in `location.hash` and then invoke an event handler defined by the application. The event handler will respond by using the value of the `location.hash` property to set the value of the `<body>` element's `class` attribute. If the attribute value is different from the previous one, the application will switch to the new view.

STEP 4B: ENABLE NAVIGATION BETWEEN VIEWS USING JAVASCRIPT TO INITIATE VIEW CHANGES

Let's start off by defining methods to switch between views in the application. The code in the next listing creates a new object constructor, `Tasks`, containing two functions, `nudge` and `jump`. When the page has loaded, a new `Tasks` object is created, which forms the basis for your application. Take the code in the following listing and insert it into a new file, app.js. Store this file in the same directory as index.html.

Listing 5.4 app.js—Foundation JavaScript code for the application

```
(function() {
    var Tasks = function() {
        var nudge = function() {
            setTimeout(function(){ window.scrollTo(0,0); }, 1000);
        }
        var jump = function() {
            switch(location.hash) {
                case '#add':
                    document.body.className = 'add';
                    break;
                case '#settings':
                    document.body.className = 'settings';
                    break;
                default:
                    document.body.className = 'list';
            }
            nudge();
        }
        jump();
        window.addEventListener('hashchange', jump, false);
        window.addEventListener('orientationchange', nudge, false);
    }

    window.addEventListener('load', function() {
        new Tasks();
    }, false);
})();
```

The nudge function hides the browser toolbar on iOS devices to gain extra space for the application.

When a user wants to change the view, they click a button on the navigation bar. This action changes the value of location.hash and raises a hashchange event. You want to call jump when a hashchange is detected.

The jump function takes the value of location.hash and uses it to define the current view. Notice how the Tasks constructor calls jump after its definition. Because the user may have bookmarked a view other than the application's home view of Task List, the Tasks constructor uses jump to check the value of location.hash for a non-default view.

On mobile devices, when the screen orientation changes, call the nudge function to hide the browser toolbar, if possible.

After the page loads, create a new instance of the Tasks object to start the application.

TRY IT OUT

If you run the application in any HTML5-compatible web browser, you should be able to navigate between the different views of the application and see the current view highlighted in the navigation bar. This is illustrated in figure 5.2.

If you are trying to run this app on your desktop browser with the Python web server, start the My Tasks app by entering localhost:8000 into your browser's address

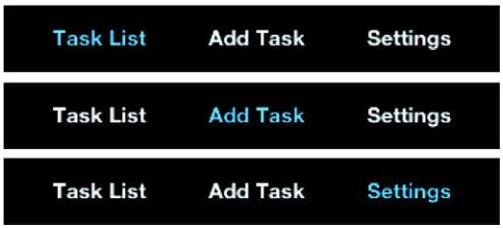

Figure 5.2 The application highlights the current view by displaying a navigation button with a darker background and blue text.

box. (If you configured the web server with a different port number, use that number instead of 8000.)

With the basics out of the way, let's move on to implementing the Settings view using the Web Storage API in HTML5.

5.2 Managing data with the Web Storage API

Among other features, the Settings view allows users to choose a name and color scheme for the application. Traditionally, web applications would have implemented this either by storing the user's settings in a remote database on the server side or by storing the preferences in a cookie, which often gets deleted when the user clears their browsing history.

Web Storage API	4.0	3.5	8.0	10.5	4.0

Fortunately, we have better options with HTML5: the Web Storage specification. It defines two `window` attributes for storing data locally on the client: `localStorage` and `sessionStorage`. The `localStorage` attribute allows you to store data that will persist on the client machine between sessions. The data can be overwritten or erased only by the application itself or by the user performing a manual clear down of the local storage area. The API of the `sessionStorage` attribute is identical to that of the `localStorage` attribute, but `sessionStorage` won't persist data between browser sessions, so if the user closes the browser, the data is immediately erased.

TIP You can try `sessionStorage` in this section by replacing any reference to `localStorage` with `sessionStorage` in the listing to come.

In this section, you'll learn

- How to read data from `localStorage`
- How to write data to `localStorage`
- How to delete some or all data from `localStorage`

To implement the management of the application's settings using the Web Storage API and to integrate the setting functions with the UI, you'll need to follow these four steps:

- Step 1: Read application settings from `localStorage`.
- Step 2: Save application settings to `localStorage`.
- Step 3: Clear all settings and data from `localStorage`.
- Step 4: Connect the UI to `localStorage` functions.

NOTE You need to complete all the steps before you can run and test the code in this section.

5.2.1 Reading data from localStorage

When the application starts, it will need to read the user's name and chosen color scheme from some client-based data store, then apply them to the UI. You'll use `localStorage` as a repository for this information and store each piece of data as a key/value pair. Retrieving items from `localStorage` is done by calling its Storage API method `getItem` with the value's key.

STEP 1: READING APPLICATION SETTINGS FROM LOCALSTORAGE

Core API

For the purpose of retrieving application settings from `localStorage`, the application will need a `loadSettings` function. This function reads the user's name and color scheme from `localStorage` using the Web Storage API method `getItem` and then adjusts the navigation bar's header to include the user's name, and changes the document element's `class` attribute to assign the selected color scheme.

Open the app.js file you created earlier in the chapter, and add the code from the next listing to the `Tasks` constructor function (just below the line where you attach a handler to the `orientationchange` event).

Listing 5.5 app.js—Reading data from `localStorage`

```
var localStorageAvailable = ('localStorage' in window);

var loadSettings = function() {
    if(localStorageAvailable) {
        var name = localStorage.getItem('name'),
            colorScheme = localStorage.getItem('colorScheme'),
            nameDisplay = document.getElementById('user_name'),
            nameField = document.forms.settings.name,
            doc = document.documentElement,
            colorSchemeField = document.forms.settings.color_scheme;
        if(name) {
            nameDisplay.innerHTML = name+"'s";
            nameField.value = name;
        } else {
            nameDisplay.innerHTML = 'My';
            nameField.value = '';
        }
        if(colorScheme) {
            doc.className = colorScheme.toLowerCase();
            colorSchemeField.value = colorScheme;
        } else {
            doc.className = 'blue';
            colorSchemeField.value = 'Blue';
```

Use the Storage API method getItem to retrieve data from localStorage. If the data does not exist, getItem will return a null value instead.

Before you start to access localStorage, query the window object for a localStorage attribute. The variable localStorageAvailable will be true if the browser supports the localStorage attribute.

```
        }
    }
}
```

At this point you're probably wondering how your application is going to read data from `localStorage` when you haven't actually saved anything in the first place. Fear not! You're going to solve that problem next by creating a function that will save the user's selected settings to `localStorage`.

5.2.2 Saving data to localStorage

Core API

Saving the user's settings is relatively easy. Save data in `localStorage` by using its Web Storage API method `setItem`, passing two arguments: a key and value.

STEP 2: SAVE NAME AND COLOR SCHEME TO LOCALSTORAGE

In order to save the user's name and chosen color scheme, you'll implement a new function, `saveSettings`. It will store the user's preferences and change the `location.hash` to `#list`, the Task List view. Add the code from the next listing directly after the `loadSettings` function from the previous listing.

Listing 5.6 app.js—Saving data to `localStorage`

```
var saveSettings = function(e) {
    e.preventDefault();
    if(localStorageAvailable) {
        var name = document.forms.settings.name.value;
        if(name.length > 0) {
            var colorScheme = document.forms.settings.color_scheme.value;

            localStorage.setItem('name', name);
            localStorage.setItem('colorScheme', colorScheme);
            loadSettings();
            alert('Settings saved successfully', 'Settings saved');
            location.hash = '#list';
        } else {
            alert('Please enter your name', 'Settings error');
        }
    } else {
        alert('Browser does not support localStorage', 'Settings error');
    }
}
```

Use the setItem method to store data in localStorage. If an item with this name already exists, it will be overwritten without warning.

When the data has been stored, call loadSettings to update the application with the new settings.

Setting location.hash to #list will trigger a redirect to the Task List view.

You've now seen how to read and write data using the Web Storage API. Next, we'll show you how to remove data.

5.2.3 Deleting data from localStorage

Core API

In the Settings view of My Tasks, the user has an option to remove all items and settings from the application. So, you'll need to consider the two data-removal methods in the Storage API. The first, `removeItem`, is useful when you need to delete a single item from `localStorage`. The method requires one argument, the key to identify and remove the value from `localStorage`. Because the application needs to reset all settings

and data in the application, you won't use `removeItem`. Instead, you'll want the second method, `clear`, which removes all items from `localStorage`.

STEP 3: CLEAR ALL SETTINGS AND DATA FROM LOCALSTORAGE

You'll need a function, `resetSettings`, to erase all the settings data in the application. Before `resetSettings` erases the data, you should ask the user to confirm this action. After erasing the data, load the default user settings into the application and change the `location.hash` to `#list`, the Task List view.

Add the following code immediately after the code from the previous listing.

Listing 5.7 app.js—Clearing data from `localStorage`

```
var resetSettings = function(e) {
    e.preventDefault();
    if(confirm('This will erase all data. Are you sure?', 'Reset data')) {
        if(localStorageAvailable) {
            localStorage.clear();
        }
        loadSettings();
        alert('Application data has been reset', 'Reset successful');
        location.hash = '#list';
    }
}
```

When the data has been removed, call loadSettings to restore application defaults.

Before clear down of localStorage, the application will prompt the user to confirm deletion of user settings.

Change location.hash to trigger a redirect to the Task List view.

At this point, all of the functions for interacting with `localStorage` have been created, and all that's left is to connect the UI to these functions.

STEP 4: CONNECT THE UI TO THE LOCALSTORAGE FUNCTIONS

The final piece of the puzzle for our sample application is to add event handlers to the Settings view so that data is saved and reset when the buttons are pressed. Aside from connecting the storage methods to the buttons, you'll need to call `loadSettings` so that data is read from `localStorage` each time the application page loads. The code you need to add (again, add it below the code from the previous listing) is in the following listing.

Listing 5.8 app.js—Connecting the UI to the `localStorage` functions

```
loadSettings();
document.forms.settings.addEventListener('submit', saveSettings, false);
document.forms.settings.addEventListener('reset', resetSettings, false);
```

Attach event handlers to the submit and reset events of the Settings form.

TRY IT OUT!

If you now launch the application in a compatible browser, you should be able to navigate to the Settings view and change the name and color scheme from the default settings. Figure 5.3 shows this happening on a BlackBerry Torch 9860 smartphone.

If you were to press the Reset All Data button, the application would return to its default color and name.

Because you're using `localStorage`, these name and color settings will persist between browser sessions (unless the user specifically clears down their `localStorage`

Figure 5.3 The user fills out the Settings form and presses the Save Settings button. When the data has been saved to `localStorage`, the settings are reloaded, and a message is displayed to the user. When the user dismisses this message, they are taken back to the Task List view (which is empty for now).

area via the browser preferences screen). Try refreshing the page, restarting your browser, and even restarting your computer; the data should persist. Pretty neat.

In the next section, we'll show you how to take things even further with client-side data storage using the IndexedDB API. We'll do so by having you add real meat to your sample application by implementing the ability to add, edit, delete, view, and search tasks.

5.3 Managing data using IndexedDB

IndexedDB provides an API for a transactional database that is stored on the client side. The Web Storage API stores and retrieves values using keys; IndexedDB supports more advanced functionality, including in-order retrieval of keys, support for duplicate values, and efficient value searching using indexes.

IndexedDB	11.0	4.0	10.0	N/A	N/A

In the cases where the application detects no browser support for IndexedDB, you'll use Web SQL as a fallback.

FYI: More about Web SQL

IndexedDB was added to HTML5 quite late in the specification process. As a result, browser support for it has been much slower than with other parts of the specification. Prior to IndexedDB, HTML5 included a client-side database specification known as Web SQL, which defined an API for a full relational database that would live in the browser. Although Web SQL is no longer part of HTML5, many browser vendors had already provided decent support for it, particularly mobile browsers.

Using the IndexedDB API can be notoriously complex at first glance, particularly if you don't have experience writing asynchronous JavaScript code that uses callback functions. But this section will slowly guide you in the use of IndexedDB as you add task management features to My Tasks.

In this section, you'll learn

- How to create and connect to an IndexedDB database
- How to load existing data from an IndexedDB database
- How to perform queries on an IndexedDB database using IndexedDB's key ranges
- How to store new data in an IndexedDB database
- How to delete single data items from an IndexedDB database
- How to clear an entire data store from an IndexedDB database

As you learn how to use the database services of IndexedDB and Web SQL, you'll also implement the UI for the Add Task and Task List views. Overall, building out the UI and application features happens in eight steps:

- Step 1: Detect IndexedDB or Web SQL.
- Step 2: Connect to the database and create an object store.
- Step 3: Develop the UI for the Task List view.
- Step 4: Implement a search engine for the database and display search results.
- Step 5: Implement the search interface for the Task List view.
- Step 6: Add new tasks from the Add Task view to the database.
- Step 7: Update and delete tasks from the Task List view.
- Step 8: Drop the database to clear all tasks.

5.3.1 Detecting database support on a browser

Before you can create a database, you need to detect what database system is running within a browser. Currently two systems can be found: IndexedDB and Web SQL. Detection of the database system is done by assigning a variable to a logical expression of alternating `or` operators (`||`) and vendor-prefixed `IndexDB` object identifiers. Because IndexedDB isn't a standard feature, you must use the vendor prefixes to access the database system object on the various browsers.

If a database object is found, the application saves the found database object to a variable for later use; otherwise, the application assigns a `false` value to the variable. You also need to find and save the database key range. We'll discuss the key range later in the section.

STEP 1: DETECT INDEXEDDB OR WEB SQL

Now, to add feature detection to the sample application, add the code from the following listing to the app.js file. This code should be added immediately after the code you inserted in the previous section.

Listing 5.9 app.js—Feature detection for database-related objects

```
var indexedDB = window.indexedDB || window.webkitIndexedDB
        || window.mozIndexedDB || window.msIndexedDB || false,

IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange
        || window.mozIDBKeyRange || window.msIDBKeyRange || false,

     webSQLSupport = ('openDatabase' in window);
```

Web SQL object is not implemented as a member of window. To detect if the browser supports Web SQL, check for the existence of openDatabase as a member of window.

5.3.2 Creating or connecting to an IndexedDB database, creating an object store and index

Core API

To create or connect to an IndexedDB database, the application needs to invoke the `IndexedDB` method `open`. If no database exists when the `open` method is called, a new database will be created, and a connection object will be created. Once `indexedDB.open` successfully creates a connection, the `onsuccess` and/or `upgradeNeeded` event handler will be called, and the connection object will be accessible through the event object passed to the event handler.[1] With this connection object, the application can create an object store or index for the application.

Before looking at how an application would create object stores and indexes, let's discuss how data is stored in an IndexedDB database. All data in an IndexedDB database is stored inside an object store. Each database can contain many object stores, which can be roughly thought of as equivalent to tables in a relational database management system (RDBMS). In turn, each object store comprises a collection of zero or more objects, the equivalent of rows in a RDBMS. Figure 5.4 illustrates the structure of an IndexedDB database.

Now that you have a better idea of how objects are stored in the IndexedDB database, let's get back to creating object stores and indexes.

Core API

Object stores can only be created while the application is handling an `upgradNeeded` event. This event can occur in two situations: when a new database is created and when a database's version number is increased. Once the application has entered the `upgradeNeeded` event handler, the object store is created by calling the

[1] If a new database is created, events `upgradeNeeded` and `onsuccess` will be fired, but `upgradeNeeded` will be handled before `onsuccess`.

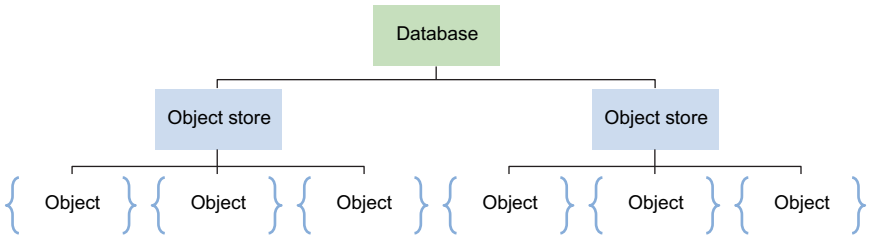

Figure 5.4 Hierarchical structure of an IndexedDB database. Each database can have many object stores, which themselves can contain many objects. The object is the structure for a data record, equivalent to a row in a relational database.

`createObjectStore` method with two arguments: a name and keypath for the new object store. The keypath defines what property within each object will serve as the key for retrieving the object from its store.

Core API

Once the object store is created, you can create one or more indexes for it. Creating an index allows the application to retrieve an object with a key different than the one defined in the object store. To create a new index, use the object store's method `createIndex` and pass it three arguments: the name of the new index, the name of the object property that will serve as the key, and an `options` object.

The `options` object has two properties that serve as flag parameters. The first flag, `unique`, allows the application to specify whether or not a key can be shared. The second flag, `multiEntry`, allows the application to specify how to handle array-based keys: Either enter an object under several different keys listed in an array, or enter an object using the entire array as a key. You won't need to use the second flag in the My Tasks application (for more detail about `multiEntry`, see appendix B or www.w3.org/TR/IndexedDB/#dfn-multientry).

Let's look at the database-creation process and apply it to our application.

STEP 2: CONNECT TO THE DATABASE AND CREATE AN OBJECT STORE

You will need to create an object store, "tasks", for all the tasks the user will want to keep track of. Remember to first create the database connection, because you'll need this to create the object store and the index. You'll use the index to access the object store by the task's description. This will be useful when you implement the application's search engine that allows the user to filter their task list by a task's description.

You'll also add a call to the `loadTasks` function here. It's not related to object store or index creation, but it will be useful later when the application is in the startup phase and needs to load the existing `task` objects into the Task List view. You'll implement `loadTasks` later in this section.

The following listing might seem like a lot of code, but it's doing quite a bit for us: opening a database connection, creating an object store, and providing a Web SQL fallback for browsers that don't support IndexedDB. Add the code from this listing to app.js, just below the code you added from listing 5.9.

Listing 5.10 app.js—Connecting to and configuring the database

```
var db;

var openDB = function() {
    if(indexedDB) {
        var request = indexedDB.open('tasks', 1),
            upgradeNeeded = ('onupgradeneeded' in request);
        request.onsuccess = function(e) {
            db = e.target.result;
            if(!upgradeNeeded && db.version != '1') {
                var setVersionRequest = db.setVersion('1');
                setVersionRequest.onsuccess = function(e) {
                    var objectStore = db.createObjectStore('tasks', {
                        keyPath: 'id'
                    });
                    objectStore.createIndex('desc', 'descUpper', {
                        unique: false
                    });
                    loadTasks();
                }
            } else {
                loadTasks();
            }
        }
        if(upgradeNeeded) {
            request.onupgradeneeded = function(e) {
                db = e.target.result;
                var objectStore = db.createObjectStore('tasks', {
                    keyPath: 'id'
                });
                objectStore.createIndex('desc', 'descUpper', {
                    unique: false
                });
            }
        }
    } else if(webSQLSupport) {
        db = openDatabase('tasks','1.0','Tasks database',(5*1024*1024));
        db.transaction(function(tx) {
            var sql = 'CREATE TABLE IF NOT EXISTS tasks ('+
                        'id INTEGER PRIMARY KEY ASC,'+
                        'desc TEXT,'+
                        'due DATETIME,'+
                        'complete BOOLEAN'+
                      ')';
            tx.executeSql(sql, [], loadTasks);
        });
    }
}

openDB();
```

Use db to store the database connection.

The open method is asynchronous; while the request is in progress, open immediately returns an IDBRequest. If no database exists, create one, and then create a connection to the database.

If upgradeNeeded is a member of the request object, then the browser supports upgradeNeeded event.

If the event upgradeNeeded doesn't exist, then the browser supports the deprecated setVersion method.

If db.version is not equal to 1, then no object store exists and it must be created. Object stores can only be created during a version-change transaction. So, increase the version number of the current database by calling db.setVersion with a version argument set to '1'.

Use createIndex to create another index for the objectStore. This index will be used later to implement the application's search feature.

This event handler will be called when the database is created for the first time.

Allocate 5 MB (5 * 1024 * 1024) for the tasks database.

Use the executeSql method of the transaction object, tx, to create a tasks table if it doesn't already exist. A [ ] means no optional argument array being passed. loadTasks is the callback function.

Now that you can open a connection to the database and create an object store, let's look at how users will interact with the tasks database by developing the UI for the

Task List view. Building this interface will generate a list of user features to guide your later development of database management functions.

5.3.3 Developing a dynamic list with HTML and JavaScript

Your Task List view will require a list of to-do items that can change as the user adds and deletes tasks. Building a web page with a varying list requires the use of JavaScript to generate new HTML markup for each list item and its UI controls. In addition, you'll need to insert those new list items by making modifications to the DOM. If a user needs to delete an item, the application will regenerate the entire list rather than try to remove an individual list item from the DOM. Although this isn't the most efficient way to handle list management, it's fast to implement and allows you to get on to more interesting tasks like learning about the HTML5 IndexedDB API!

STEP 3: DEVELOP THE UI FOR THE TASK LIST VIEW

The Task List view is a dynamic part of the application's webpage that updates itself in response to user actions. Here's a list of those actions and how to implement them:

- *Adding a task to the list*—The application needs to define a function, `showTask`, to generate the HTML markup for each added task and then insert the markup into the view's DOM.
- *Checking off and deleting tasks*—You'll also use `showTask` to add check boxes and Delete buttons to each added task. `showTask` will also define and bind an event handler for each check box and delete button.

Figure 5.5 illustrates how the buttons and check boxes will appear.

The code in listing 5.11 implements the `showTask` and `createEmptyItem` functions. `CreateEmptyItem` is a helper function to handle the boundary conditions where the user has no task items to display in the to-do list. This can occur in two situations:

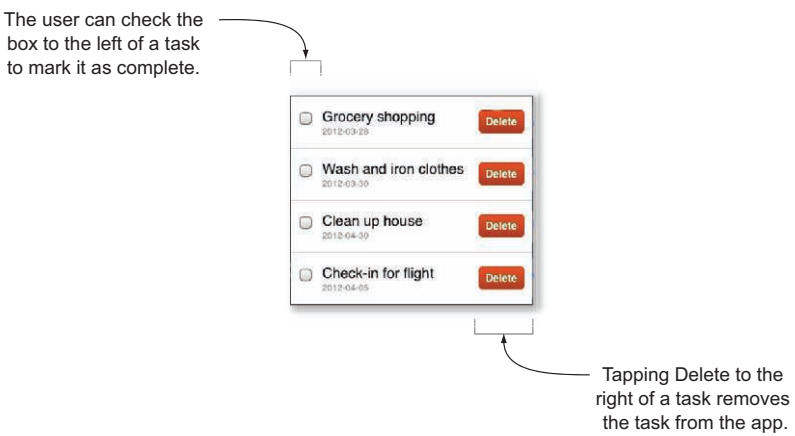

Figure 5.5 Each task item has two components that allow the user to update the task list. Checking the box on the left-hand side will mark the task as complete, whereas pressing the red Delete button on the right-hand side will remove the task.

when no task items exist in the database and when a search of the task list yields no matches. In order to handle these cases, `createEmptyItem` will create an "empty item," actually a message that says either "No tasks to display. Add one?" or "No tasks match your query."

Add the following code to your application, just after the code from the previous listing.

Listing 5.11 app.js—Generating the markup for task items

```
var createEmptyItem = function(query, taskList) {
    var emptyItem = document.createElement('li');
    if(query.length > 0) {
        emptyItem.innerHTML = '<div class="item_title">'+
            'No tasks match your query <strong>'+query+'</strong>.'+
            '</div>';
    } else {
        emptyItem.innerHTML = '<div class="item_title">'+
           'No tasks to display. <a href="#add">Add one</a>?'+
           '</div>';
    }
    taskList.appendChild(emptyItem);
}

var showTask = function(task, list) {
    var newItem = document.createElement('li'),
        checked = (task.complete == 1) ? ' checked="checked"' : '';

    newItem.innerHTML =
        '<div class="item_complete">'+
            '<input type="checkbox" name="item_complete" '+
                'id="chk_'+task.id+'"'+checked+'>'+
        '</div>'+
        '<div class="item_delete">'+
            '<a href="#" id="del_'+task.id+'">Delete</a>'+
        '</div>'+
        '<div class="item_title">'+task.desc+'</div>'+
        '<div class="item_due">'+task.due+'</div>';
    list.appendChild(newItem);

    var markAsComplete = function(e) {
        e.preventDefault();
        var updatedTask = {
            id: task.id,
            desc: task.desc,
            descUpper: task.desc.toUpperCase(),
            due: task.due,
            complete: e.target.checked
        };
        updateTask(updatedTask);
    }

    var remove = function(e) {
        e.preventDefault();
        if(confirm('Deleting task. Are you sure?', 'Delete')) {
```

If a query doesn't exist, the search will return zero results.

The showTask function creates and displays a task list item containing a title, due date, check box, and Delete button.

The markAsComplete event handler is executed when the user marks or unmarks the check box.

The remove event handler is executed when the user clicks the Delete button for a task item.

```
            deleteTask(task.id);
        }
    }

    document.getElementById('chk_'+task.id).onchange =
       markAsComplete;
    document.getElementById('del_'+task.id).onclick = remove;
}
```

This code attaches event handlers to the task item's check box and remove button.

5.3.4 Searching an IndexedDB database

Core API

Now that the UI for the Task List view is complete, you need to search the IndexedDB database to extract a list of `task` objects for display in the Task View list. To do this, IndexedDB requires the creation of a transaction to define an array of object stores to scan and the type of transaction to execute. The transaction type defines how the database will be accessed. IndexedDB provides two options: read-only and read-write. In the case of implementing a search for the My Tasks application, the transaction would need to be defined with tasks as the object store to search and a transaction type of `'readonly'`. The application could use the read/write option, but the search performance would be slower.

Once the transaction is defined, you then need to extract the index from the object store. The index will enable the application to filter the object store based on some property of the object. In your application, the index's key is based on the task's description property. Using this index and a string describing some portion of the task description, you'll create a database cursor using the IndexedDB API method `openCursor`. The application will then use this cursor's `continue` method to iterate over the database and find all of the tasks containing a portion of the task description.

Core API

Using cursors to iterate through database records

Cursor is a generic term describing a control structure in a database that allows you to iterate through the records stored in it. Cursors typically enable you to filter out records based on certain characteristics and to define the order in which the result set is returned. Using the cursor's `continue` method, you can then sequentially move through the record set returned by the cursor, retrieving the data for use in your applications. Cursors in IndexedDB allow you to traverse a result set that's defined by a key range, moving in a direction of either an increasing or decreasing order of keys.

STEP 4: IMPLEMENT A SEARCH ENGINE FOR THE DATABASE AND DISPLAY SEARCH RESULTS

In the application, the `loadTasks` function is responsible for retrieving and displaying tasks from the IndexedDB or Web SQL database. `loadTasks` will either retrieve a filtered set of tasks or all tasks and then pass them to the `showTask` function, which will render them onto the Task List view. Add the code from the next listing immediately after the code from the previous listing.

Listing 5.12 app.js—Searching the database and displaying the resulting tasks

```
var loadTasks = function(q) {
    var taskList = document.getElementById('task_list'),
        query = q || '';
    taskList.innerHTML = '';

    if(indexedDB) {
        var tx = db.transaction(['tasks'], 'readonly'),
            objectStore = tx.objectStore('tasks'), cursor, i = 0;
        if(query.length > 0) {
            var index = objectStore.index('desc'),
                upperQ = query.toUpperCase(),
                keyRange = IDBKeyRange.bound(upperQ, upperQ+'z');
            cursor = index.openCursor(keyRange);
        } else {
            cursor = objectStore.openCursor();
        }

        cursor.onsuccess = function(e) {
            var result = e.target.result;
            if(result == null) return;
            i++;
            showTask(result.value, taskList);
            result['continue']();
        }

        tx.oncomplete = function(e) {
            if(i == 0) { createEmptyItem(query, taskList); }
            }
        } else if(webSQLSupport) {
        db.transaction(function(tx) {
            var sql, args = [];
            if(query.length > 0) {
                sql = 'SELECT * FROM tasks WHERE desc LIKE ?';
                args[0] = query+'%';
            } else {
                sql = 'SELECT * FROM tasks';
            }
            var iterateRows = function(tx, results) {
                var i = 0, len = results.rows.length;
                for(;i<len;i++) {
                    showTask(results.rows.item(i), taskList);
                }
                if(len === 0) { createEmptyItem(query, taskList); }
            }
            tx.executeSql(sql, args, iterateRows);
        });
    }
}
```

Build a key range on the uppercase version of the task description. The 'z' appended to the second argument allows the application to search for a task description beginning with the search term (otherwise, it would only return exact matches).

e.target references the cursor, so get the result set from the cursor.

Count the number of tasks passed to showTask. The resulting value will be used by the transaction event handler, tx.onComplete, to determine if an empty task list should be rendered.

Use result['continue'] to find the next matching task in the index or next task in the object store (if not searching). Using result.continue, rather than result['continue'], might result in a conflict with the JavaScript reserved word continue.

If IndexedDB isn't supported and Web SQL is, build a query that will retrieve the tasks from the database.

NOTE You may have noticed that the `loadTasks` function accepts an optional argument, `q`. The application will only pass a query to `loadTasks` when it wants to filter the results by what the user has entered in the search box.

STEP 5: IMPLEMENT THE SEARCH INTERFACE FOR THE TASK VIEW LIST

To implement the search interface for the application, add the following code immediately after the code from the previous listing.

Listing 5.13 app.js—Searching for tasks

```
var searchTasks = function(e) {
    e.preventDefault();
    var query = document.forms.search.query.value;
    if(query.length > 0) {
        loadTasks(query);
    } else {
        loadTasks();
    }
}

document.forms.search.addEventListener('submit', searchTasks, false);
```

If a query was typed in, pass the query as an argument to the loadTasks function.

When the user submits the search form, call the searchTasks function.

TRY IT OUT

If you reload the application in your browser, you should see a friendly message telling you that you have no tasks to display, as shown in figure 5.6.

As you can see from figure 5.6, displaying a list of tasks isn't very useful if you have no way of adding tasks to the database. Let's solve that problem right now.

5.3.5 Adding data to a database using IndexedDB or Web SQL

Core API

Adding data to an IndexedDB database requires the creation of a transaction to define an array of object stores you'll be using to store the data and the type of transaction needed, in this case `'readwrite'`. Once you have the transaction created, you then call its method `objectStore`, with the name of the object store you want to add a data item to. The method will respond to this call by returning the object store. From here, adding the data item to the store is easy. Call the object

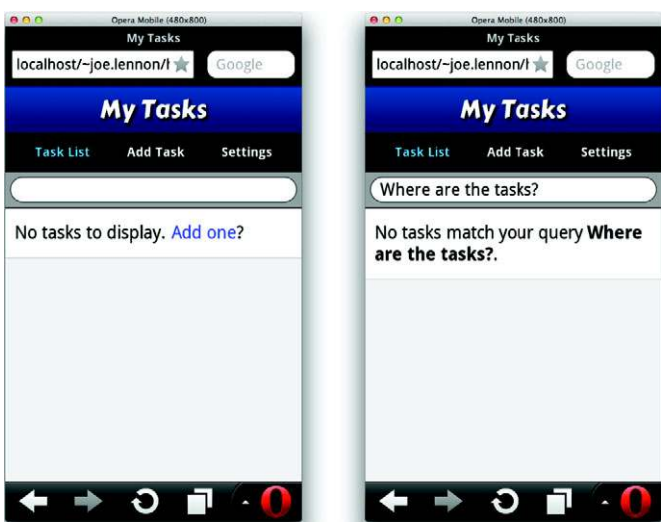

Figure 5.6 In the left screenshot, the application finds no tasks in the database. Therefore, it displays a message and links the question "Add one?" to the Add Task form. In the right screenshot, if you try to search for a task, you'll see that no tasks match your query, no matter what you enter.

store's method `add`, and pass the new data item to its only argument. The method will immediately return a `request` object. If you'd like the application to respond to the object store's successful addition of the data item, then define an event handler for the transaction's `oncomplete` event.

Now, let's see how this addition procedure can be applied to the application.

STEP 6: ADD NEW TASKS FROM THE ADD TASK VIEW TO THE DATABASE

This code creates a new function called `insertTask` that manages the process of inserting the task into the database and updating the display of the Task List view. `InsertTask` first constructs a new `task` object from the Add Task form; second, it adds the task to the IndexedDB database (or Web SQL if the browser doesn't support IndexedDB). Finally, it triggers the callback function, `updateView`, when the task has been successfully added to the database. Add the code from the following listing after the code from the previous listing.

Listing 5.14 app.js—Adding new tasks

```
var insertTask = function(e) {
    e.preventDefault();
    var desc = document.forms.add.desc.value,
        dueDate = document.forms.add.due_date.value;
    if(desc.length > 0 && dueDate.length > 0) {
        var task = {
            id: new Date().getTime(),
            desc: desc,
            descUpper: desc.toUpperCase(),
            due: dueDate,
            complete: false
        }
        if(indexedDB) {
            var tx = db.transaction(['tasks'], 'readwrite');
            var objectStore = tx.objectStore('tasks');
            var request = objectStore.add(task);
            tx.oncomplete = updateView;
        } else if(webSQLSupport) {
            db.transaction(function(tx) {
                var sql = 'INSERT INTO tasks(desc, due, complete) '+
                          'VALUES(?, ?, ?)',
                    args = [task.desc, task.due, task.complete];
                tx.executeSql(sql, args, updateView);
            });
        }
    } else {
        alert('Please fill out all fields', 'Add task error');
    }
}
function updateView(){
        loadTasks();
        alert('Task added successfully', 'Task added');
        document.forms.add.desc.value = '';
        document.forms.add.due_date.value = '';
```

Construct a task object to store in the database. The key is the id property, which is the current time, and you also store the uppercase version of the description in order to implement case-insensitive indexing.

Add the task to the object store using the IndexedDB method add.

When a task has been successfully added, call the event handler updateView. The definition for updateView appears immediately after insertTask.

For the Web SQL fallback, use an INSERT statement to add the task.

updateView loads tasks from the database, clears input fields in the Add Task form, and redirects the user to the Task List view.

```
                location.hash = '#list';
                }
document.forms.add.addEventListener('submit', insertTask, false);
```

Add the event handler insertTask to the Add Task form's submit button.

TRY IT OUT

At this point, you should be able to add tasks to the database using the Add Task form. When the task has been saved, you are taken back to the Task List view, which should display the task you just created. Feel free to try it now—add some tasks. Also, be sure to try out the Search form, because this should now be fully functional. The application is starting to take shape, but you still have a small number of features to add before it's complete. Next, you'll write code to allow users to update and delete existing tasks.

5.3.6 Updating and deleting data from an IndexedDB database

The IndexedDB database has a relatively simple procedure for changing existing `data` objects in the object store. First, the application needs to define the database transaction about to occur, and then the application uses the transaction to write a `data` object to the specified object store.

In order to define the database transaction for updating an object store, the application would call the IndexedDB method `transaction` to define the type and scope of the transaction. Because updating a database requires writing to the object store, the type is specified as `'readwrite'`. The second parameter, the scope of the transaction, specifies the various object stores the application will be writing to.

With the transaction defined, the application can now get the object store it needs to update. Calling the transaction's method, `objectStore`, with a parameter specifying the name of the object store will return the object store. At this point, the application can update the object store by invoking its `put` method, with the changed `data` object as its parameter.

Deleting task items follows a similar procedure. But once the application has the object store, it will invoke the object store's `delete` method, with the `data` object's key as a parameter. `Delete` will use the key to find and delete the `data` object within the object store.

Let's apply these update and delete operations to the application.

STEP 7: UPDATE AND DELETE TASKS FROM THE TASK LIST VIEW

You've already done some of the work required for updating and deleting a task. If you look at the Task List view in the application, you'll notice that each task has a check box and a Delete button. The check box has an `updateTask` embedded into the `markAsComplete` event handler, and the Delete button has a `deleteTask` embedded into the `remove` event handler.

All that's left to do is to insert the procedures for updating and deleting the object store into their respective `updateTask` and `deleteTask` function definitions. Because not all browsers support IndexedDB, you'll also insert a Web SQL fallback. Add the code from this listing right beneath the code from the previous listing..

Listing 5.15 app.js—Updating and deleting tasks

```
var updateTask = function(task) {
    if(indexedDB) {
        var tx = db.transaction(['tasks'],  'readwrite');
        var objectStore = tx.objectStore('tasks');
        var request = objectStore.put(task);
    } else if(webSQLSupport) {
        var complete = (task.complete) ? 1 : 0;
        db.transaction(function(tx) {
            var sql = 'UPDATE tasks SET complete = ? WHERE id = ?',
                args = [complete, task.id];
            tx.executeSql(sql, args);
        });
    }
}

var deleteTask = function(id) {
    if(indexedDB) {
        var tx = db.transaction(['tasks'], 'readwrite');
        var objectStore = tx.objectStore('tasks');
        var request = objectStore['delete'](id);
        tx.oncomplete = loadTasks;
    } else if(webSQLSupport) {
        db.transaction(function(tx) {
            var sql = 'DELETE FROM tasks WHERE id = ?',
                args = [id];
            tx.executeSql(sql, args, loadTasks);
        });
    }
}
```

Use the put method, passing the task object as an argument, to update the task in the database. The task object must have the correct key value, or the database may create a new object in the store rather than update the existing one.

Use the delete method to remove a task. Some browsers will choke if you use dot-notation here, because delete is a reserved word in JavaScript. So to be safe, use the square bracket notation.

When the delete operation has successfully completed, load the Task List view to show the updated items.

TRY IT OUT

You should now be able to mark the completed check box and delete items in the Task List view. But one final function remains to complete the application: the `dropDatabase` function. This will delete the entire tasks database (or truncate the tasks table if using the Web SQL fallback).

5.3.7 Dropping a database using IndexedDB

Core API

Dropping a database in IndexedDB is easy and involves just one method: the `deleteDatabase` method of the `IndexedDB` object. Call `deleteDatabase` while passing the name of the target object store, and then the entire database will be removed.

STEP 8: DROP THE DATABASE TO CLEAR ALL TASKS

To enable a user to clear all tasks from the application, you need to do two things:

1. Create a new function, `dropDatabase`, that will remove the tasks database, and therefore all task items, from the application.
2. Call `dropDatabase` from the `resetSettings` function you created earlier in the `localStorage` section of this chapter. Adding this call now completes `resetSettings`'s function, which is to reset a user's personal settings and erase all of a user's tasks.

For browsers that don't support IndexedDB, you'll need to provide a Web SQL fallback as well. In this case, you won't drop the database; you'll just delete the tasks table from the Web SQL database.

To define the `dropDatabase` function, add the code from the next listing directly below the code from the previous listing in your app.js file.

Listing 5.16 app.js—Dropping the database

```
var dropDatabase = function() {
    if(indexedDB) {
        var delDBRequest = indexedDB.deleteDatabase('tasks');
        delDBRequest.onsuccess = window.location.reload();
    } else if(webSQLSupport) {
        db.transaction(function(tx) {
            var sql = 'DELETE FROM tasks';
            tx.executeSql(sql, [], loadTasks);
        });
    }
}
```

Use the deleteDatabase method to drop the tasks database.

Reload the page to initiate a load event. This will trigger the load event handler to create a fresh copy of the database.

In your Web SQL fallback, clear down the tasks table rather than drop the entire database.

With the `dropDatabase` function defined, you can now call it from the `resetSettings` function you created in section 5.2.3. In this function, locate the line `location.hash = '#list';` and add the following line just beneath it:

```
dropDatabase();
```

TRY IT OUT

That's it! The sample application should now be fully functional. Try it out on a device or browser that supports IndexedDB or Web SQL. (iOS, Android, BlackBerry Torch, Opera Mobile, Chrome, Firefox, Safari, and Opera all work.) If both IndexedDB and Web SQL are available in the browser, the application will favor the former. In the next and final section of this chapter, you'll learn how to ensure an application will work offline using an application cache manifest file. You should then have an application that stores all of its data on the client and is usable both online and offline.

5.4 Creating a web application that works offline: using the application cache manifest

Until recently, web applications have been used primarily in connected environments, on desktop or laptop computers, where the majority of the time an internet connection is available. But as rich web applications become more prominent as realistic alternatives to their desktop counterparts, and as mobile applications continue to gather momentum, the need grows for web applications to work in scenarios where connectivity is not available.

Application cache manifest	4.0	3.5	10.0	10.6	4.0

To address these demands, HTML5 provides a file called the application cache manifest. This file, in its most basic form, specifies a list of web resources needed by a web application. Browsers that support the manifest feature will use the list to provide a web application with access to a local cache of these web resources. As a result, the web application can run offline.

For resources only available from the network, the cache manifest can specify fallback client-side URIs for offline activity. For instance, if an application relies on a JavaScript file to save data to a server, then the cache manifest would specify a client-side URI pointing to a JavaScript file that uses local requests for client-side storage.

> **NOTE** If you've been working through this chapter's example without a web server, it's worth pointing out that you won't be able to use the application cache manifest unless your application resides on an actual web server (rather than just sitting in a local directory). You'll also need to do a small bit of configuration to get cache manifests to work, which we'll cover later in the section.

The cache manifest can also specify URIs that must be fetched from the network. They will never be downloaded from the application cache, even if the application is offline.

In this section, you'll learn

- How to configure a web server for an application cache manifest MIME type
- How to create a cache manifest file
- How to detect changes in the manifest file

Now that you have a basic understanding of the application cache manifest, let's implement offline functionality for My Tasks. This process will be broken down into three steps:

- Step 1: Configure the web server to serve application cache manifest files for My Tasks.
- Step 2: Create an application cache manifest file for My Tasks.
- Step 3: Detect changes in the My Tasks application cache manifest file.

5.4.1 Configuring a web server for an application cache manifest's MIME type

In order for a manifest file to be correctly loaded, your web server needs to serve a manifest file using the correct MIME type. The manifest MIME type is not typically set by default in a web server's configuration, so you'll need to add the MIME type, `text/cache-manifest`, to your web server's configuration.

STEP 1: CONFIGURE THE WEB SERVER TO SERVE APPLICATION MANIFEST FILES FOR MY TASKS

If you're using the Apache web server, you can typically add MIME types by either modifying the httpd.conf configuration file or by serving an .htaccess file in the root of your

web application. If you're using Python's built-in web server, then create an .htaccess file in the root directory of your web application, and then add the MIME type to the .htaccess file. In either case, to serve the correct MIME type for files with the extension .appcache, you need to add the following line to the end of the configuration or .htaccess file:

```
addType text/cache-manifest .appcache
```

> **NOTE** A cache manifest file can have any file extension, but the file must be served with the MIME type `text/cache-manifest`.

If you're using the nginx web server, you add MIME types by adding an entry to the mime.types file in the nginx conf directory. This file typically has the following format:

```
types {
  text/html                 html htm shtml;
  text/css                  css;
  text/xml                  xml;
  ...
}
```

To enable the cache manifest MIME type, add an entry to this file as follows:

```
  text/cache-manifest       appcache;
```

After editing the configuration file, restart your web server, and your cache manifest file should be served correctly from now on. If you're using another web server, please consult your web server's documentation for further information on how to add MIME types.

With the web server configured correctly, you're now ready to create a cache manifest file, which we'll cover next.

5.4.2 Creating a cache manifest file

The manifest file is a basic text file that contains a title header, `CACHE MANIFEST`, and up to three subsections with the headings `CACHE`, `NETWORK`, and `FALLBACK`. For explanatory purposes only, here's a sample cache manifest file:

```
CACHE MANIFEST
# Rev 3

CACHE:
index.html
pics/logo.png
stylesheet.css

FALLBACK:
*.html      /offline.html

NETWORK:
http://api.stockwebsite.com
```

The `CACHE` section represents the default section for entries. URIs listed under this header will be cached after they're downloaded for the first time.

NOTE You can also forgo specifying a CACHE header and simply place the URIs to be cached immediately under the title header, CACHE MANIFEST.

The FALLBACK section is optional and specifies one or more pairs of URIs to use when a resource is offline. The first URI in a pair is the online resource; the second is the local fallback resource. Wildcards can be used.

NOTE Both URIs must have a relative path name. Also, the URIs here, as well as in other sections of the cache manifest, must have the same scheme, host, and port as the manifest.

The NETWORK section serves as the application's whitelist for online access. All URIs listed under this header must bypass the cache and access an online source. Wildcards can be used.

You can also specify comments in the application cache manifest. They consist of any number of tabs or spaces followed by a single # and then followed by a string of characters. Comments must exist on a line separate from other section headers and URIs.

Now, equipped with knowledge of the basic structure and syntax of an application cache manifest, let's put that knowledge to work by creating one for My Tasks.

STEP 2: CREATE THE APPLICATION CACHE MANIFEST FILE FOR MY TASKS

Your cache manifest will have a CACHE section and a NETWORK section. The CACHE section will list the index.html, style.css, and app.js files as cacheable resources. The NETWORK section will contain only an asterisk, the wildcard character. Create a new file named tasks.appcache in the root directory of your web application, then add the contents of the following listing to tasks.appcache.

NOTE After entering this code listing, don't try to run the application. It will work, but you'll have to do extra work in the final section of this chapter, "Automating application updates," to get it working correctly.

Listing 5.17 tasks.appcache—Defining resources that are available offline

```
CACHE MANIFEST
# Rev 1
CACHE:
index.html
style.css
app.js

NETWORK:
*
```

This denotes the start of the cache manifest file.

Use a comment in your manifest to define the current revision number of the web application. This allows you to easily monitor and log application revisions, even if no changes are being made to the manifest file itself. Later, we'll show how to use these revision numbers to trigger application updates.

The wildcard under NETWORK specifies that the online whitelist is open; any other URIs not listed under CACHE MANIFEST, CACHE must be retrieved from the network.

In order for your application to read this file, you need to modify your HTML document with the manifest's filename. Open index.html and replace the current opening <html> element definition with the following:

```
<html lang="en" class="blue" manifest="tasks.appcache">
```

We're almost there. In the final step, you will give My Tasks the ability to detect changes in the manifest file. My Tasks will use this ability to determine when to download a newer version of My Tasks.

5.4.3 Automating application updates

When you created the cache manifest file, you used a comment with a revision number to update the manifest, to document changes in the manifest or in one or more of the web resources listed in the manifest. This practice has a function beyond documentation; it can also be used to detect and trigger application updates.

If any change is made to the text in the manifest, the application will download the new manifest and all files listed in the `CACHE MANIFEST` or `CACHE` section. When this is done, a new cache is created and an `updateready` event is fired. To update the application, you have to attach an event handler to `updateready`. The handler will swap the old cache for the new one, then ask the user for permission to update the application. If the user grants permission, the event handler will force an application reload. The reload ensures that resources from the new cache are loaded into the application. If the user declines the update, the application will use the new cache the next time the user loads the application.

Now, let's add this update feature to My Tasks.

STEP 3: DETECT CHANGES IN THE MY TASKS APPLICATION CACHE MANIFEST FILE

Core API

As mentioned before, you'll use the `updateready` event to detect changes in the application manifest. So, all you need to do is define and attach an event handler to the application cache's `updateready` event. The event handler will call the application cache's `swapCache` method and ask the user for permission to reload the application using the new version of the cache. If the user confirms, the event handler will call `window.location.reload` to reload the application using the new cache version.

Add the code from the following listing to app.js, just after the `dropDatabase` function you created in listing 5.16.

Listing 5.18 app.js—Automatic update detection and loading

```
if('applicationCache' in window) {
    var appCache = window.applicationCache;
    appCache.addEventListener('updateready', function() {
        appCache.swapCache();
        if(confirm('App update is available. Update now?')) {
            window.location.reload();
        }
    }, false);
}
```

Detect if the user's browser supports the Application Cache API.

Ask the user if they want to update the application now. If they click Yes, the page will reload using the new cache; otherwise, the new cache will be used the next time they load the page.

When updateready fires, the browser will have already redownloaded the resources listed in the manifest and created a new cache. The event handler for updateready will call swapCache to replace the old cache with the new cache.

Figure 5.7 My Tasks application running offline. You'll notice the airplane icon in the top left indicating that the phone has no network access. You may also notice that the jazzy font we used in the heading is no longer showing; this font was loaded from the Google Font API, which isn't available when you're offline.

TRY IT OUT

That's all there is to it! If you've followed these steps correctly, you should now be able to use your application offline. If you put this application on a server with a registered domain name, you could test this application on your mobile device's browser. Just visit the site in order to load the application for the first time. Now, turn on Airplane Mode on your device, which should kill all network connectivity. Refresh the page in your device's web browser, and you should still be able to use the application in full. The result can be seen in the screenshots in figure 5.7.

If you are trying to run this app on your desktop browser with the Python web server, start the My Tasks app by entering localhost:8000 into your browser's address box. (If you configured the web server with a different port number, use that number instead.)

To simulate an offline condition for the My Tasks app running in your desktop browser, kill the Python web server process, then refresh the page in your web browser. You should still be able to use the application in full.

> **NOTE** If you tried to run this application with the cache manifest before entering the code from this final listing, then you must first flush your browser's cache before loading the application from the server.

To test the application's ability to load a newer version, update the revision number in the tasks.appcache file and save it. Next, reload the application. You should see the

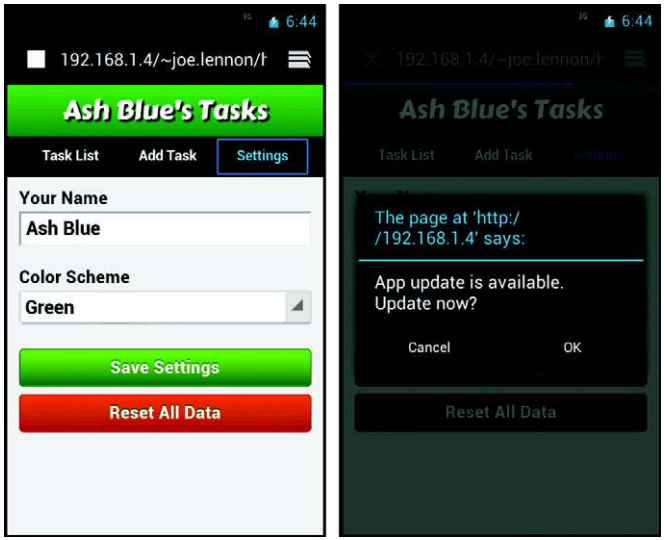

Figure 5.8 When the manifest file has been updated and a new application cache has been created, the `updateready` event is fired. The application attaches a handler to this event that swaps the old cache for the new cache, then asks the user if they want to update the application (this simply reloads the page, which loads the latest application version from the new cache).

confirmation dialog asking you if you want to update. This is illustrated in the Android device screenshot in figure 5.8.

5.5 Summary

As you've learned in this chapter, HTML5 makes it possible to create offline database applications using client-side code. This allows you to build faster, more responsive applications that store data on the device itself and work regardless of the browser's state of internet connectivity. These abilities expand the range of web applications and make the web a more viable platform for cross-platform mobile application development.

In the next chapter, you'll learn about the 2D canvas API in HTML5 and how it allows you to build animations and games using native JavaScript APIs. The chapter will introduce you to Canvas's support for drawing graphic elements using gradients, paths, and arcs. You'll also learn how to use the API to create smooth, high-frame-rate animations. In addition, the chapter will show the API in action while building an entire game.