Safari Client-Side Storage and Offline Applications Programming Guide

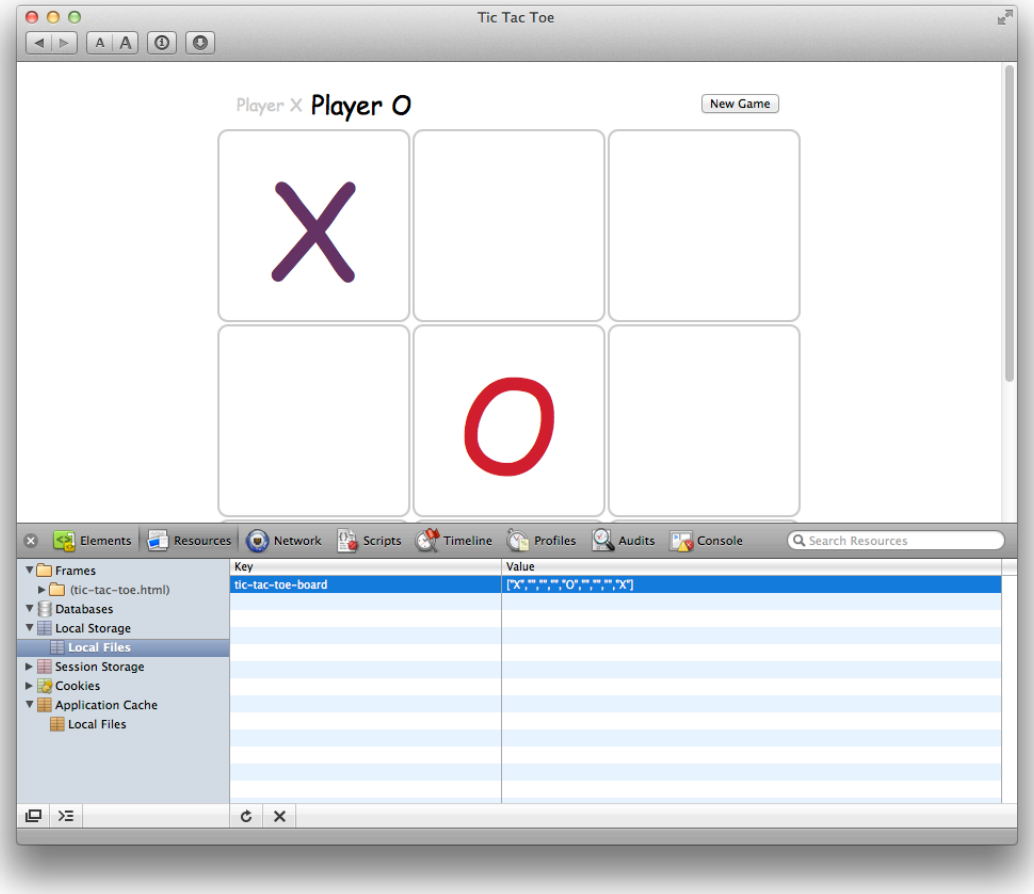
Developer

Contents

Tables and Listings

Introduction

If you develop websites, or if you are trying to decide whether to write a native iOS app or a web app, you should learn about HTML5 client-side caching and storage.



No matter what kind of website you develop, you can use Safari's HTML5 client-side storage to make your work easier and improve your site. For example, you can:

- Make your website work even when the user is offline or loses network connectivity by caching code and data locally.
- Make your website more responsive by caching resources—including audio and video media—so they aren't reloaded from the web server each time a user visits your site.

- Make your web app more user-friendly by saving user-generated data locally at frequent intervals, while reducing server load and delay by saving to your server only occasionally.
- Make every user interaction with your site faster by replacing large cookies with local storage. (Cookies are sent back to your site with every HTTP request, increasing network overhead.)
- Simplify your scripts by replacing unstructured data stored in cookies with structured key-value storage.
- Reduce server load by replacing server-side programs with client-side scripts, storing even large amounts of structured data locally using an SQL-compatible JavaScript database.

HTML5 client-side storage can make a web app nearly identical to a native app on iOS-based devices. Even without client-side storage, a web app can look and act much like an iOS native app—you can hide Safari's UI so your web app uses the full screen, use CSS to rotate the view when the user flips the device between portrait and landscape modes, and provide an icon that launches your web app from the user's start screen, just like a native app. When you add client-side storage to these other features, your web app comes very close to being a native app—the app is stored on the device, launched from the start screen, works when the user is offline, stores data locally, and looks and feels like a native app. Of course, there are still good reasons to write a native app and distribute it through the app store, but client-side storage puts web apps on a much more equal footing.

HTML5 client-side storage is supported in Safari on all platforms: Mac OS X, Windows, and iOS.

At a Glance

There are three main components to HTML5 client-side storage in Safari: the offline application cache, key-value storage, and JavaScript database storage. Each of the three components is independent of the others and can be used by itself or in combination with either or both of the others.

Use the Offline Application Cache to Make Your Website Usable Offline

To use the offline application cache, make a text file listing the resources of your website that you want to keep cached locally, then add an attribute to your HTML that identifies the list as a cache manifest. Safari loads the resources on the list into a cache that is dedicated to your domain; on subsequent visits to your website, the resources are loaded directly from the cache.

If the user is online, Safari compares the cached version of the manifest with the version on your website. If the manifest has changed, Safari checks each item on the manifest to see if the item has changed, then downloads any modified items. Items added to the manifest are automatically downloaded.

If your web app is a canvas-based game or a JavaScript calculator, for example, the entire website can be stored in the cache, allowing people to run your app even when they are offline. If your web app can be used offline, but relies on data from the web, such as an e-reader, for example, the offline application cache can be used to store webpages "lazily," as the user loads them, or the offline cache can be combined with key-value storage or a JavaScript database to store the most recent data—or user-selected data—allowing the user to read cached articles or access cached data when offline.

Relevant Chapter "HTML5 Offline Application Cache" (page 10)

Lighten Your Cookie Load By Using Key-Value Storage

Most websites store data on the user's computer by setting cookies. Cookies are strings of unstructured data, typically limited to about 4 KB each. You can expand the limit by creating multiple cookies, but this too has restrictions; some browsers limit the number of cookies per domain, and—since all the cookies in your domain are returned to you with every HTTP request—the HTTP header size itself can become a limit.

Cookies are a great place to store HTTPS authentication tokens, but for most other data storage needs, key-value storage is a better approach. Key-value storage is structured and plays well with JavaScript, has a size limit measured in Megabytes instead of Kilobytes, and doesn't add overhead to every interaction with your website by cluttering the HTTP header.

There are two types of key-value storage: session storage and local storage. Session storage is local to a browser window and goes away when the user closes the browser window or loads a new site. Local storage is shared among all open windows and is persistent, even if the user clears all cookies.

Put data into storage by defining properties of the `localStorage` or `sessionStorage` objects. For example:

```
localStorage.zip="92104"
```

The previous snippet creates a key named "zip" with a value of "92104" and puts the key-value pair into local storage. Retrieving data is just as easy:

```
var userZip = localStorage.zip
```

Key-value storage uses strings as values. Other data types can be serialized, but to store significant amounts of non-string data, a JavaScript database is probably more useful. For details, see "Using the JavaScript Database" (page 39).

Relevant Chapter "Key-Value Storage" (page 19)

Do the Heavy Lifting on the Client Instead of Your Server

When you have large amounts of data that the client needs to access in complex ways, the best approach is usually to create a relational database and provide client access using a GUI as a front end for SQL queries.

You can maintain a database on your server and use your website to provide a GUI to generate queries that your server executes, but this approach has some drawbacks:

- User-generated data needs to be sent to and stored on your server.
- The processing load for the SQL queries falls on your server, making it less responsive.
- The global data set is potentially exposed to SQL hacking.

Using an HTML5 JavaScript database lets you keep user-generated and user-consumed data on the user's computer, moves the processing load from your server to the user's CPU, and lets you limit the data set accessible to SQL over the Internet.

If the user needs access to large amounts of data stored on your server, and your server has enough power to handle multiple simultaneous user queries, a server-side database is probably best. If, on the other hand, the data is largely user-centric, or the number of simultaneous queries is creating bottlenecks on your server, a JavaScript database might be a better option.

A JavaScript database is also useful when you have large amounts of non-string data to store on the user's computer. There is more overhead to set up and maintain a relational database than a key-value store, however, so there is a trade-off involved.

You create a JavaScript database using the HTML5 `openDatabase` and `CREATE TABLE` methods. Access the database using the `executeSql` method, passing in SQL transaction strings.

Relevant Chapters "Relational Database Basics" (page 26), "Using the JavaScript Database" (page 39), "Database Example: A Simple Text Editor" (page 53).

Debug Client-Side Storage and Caching Using the Web Inspector

Safari includes a set of built-in tools you can use to inspect and debug the offline application cache, key-value storage, JavaScript databases, as well as the HTML, CSS, and JavaScript that glues them all together.

To turn on the debugging tools, enable the Develop Menu in Safari Preferences Advanced pane. Once the Develop Menu is enabled, choose Show Web Inspector from the Develop menu and click the Resources button to inspect local storage and caching. Click the Scripts button to interactively debug JavaScript. For more about how to use the debugging tools, see *Safari Developer Tools Guide*.

You can inspect and debug key-value storage and JavaScript databases using local files or using a website loaded from a web server. To use the offline application cache, however, you must load the site from a web server.

> **Tip** You can test files that use the offline application cache without uploading them to a remote web server by turning on web sharing in System Preferences and dragging the files to your Sites directory, using the web server built into Mac OS X.

Use the Companion File

Click the Companion File button at the top of the HTML version of this document to download a `.zip` file containing working sample code that shows you how to use a JavaScript database.

Prerequisites

You need a general familiarity with HTML and JavaScript to use this document effectively.

See Also

You may find the following documents helpful when working with caching and offline storage:

- HTML5 Web Storage Specification—Defines the HTML and JavaScript API for key-value storage.
- *DOMApplicationCache Class Reference*—Describes the JavaScript API for the application cache.
- *TicTacToe with HTML5 Offline Storage*—Sample code containing a game that maintains the current game state persistently using local key-value storage.
- *Safari Developer Tools Guide*—Shows how to enable and use Safari's built-in tools for inspecting and debugging local storage, databases, JavaScript, HTML, and CSS.

HTML5 Offline Application Cache

Use the offline application cache to store HTML, JavaScript, CSS, and media resources locally, to create web-based applications that work even when a returning user is not connected to the Internet. You can also use the offline application cache simply to store static resources locally, to speed access to your website and lessen the server load when a user returns to your site.

The offline application cache lets you create web apps—such as canvas-based games, e-readers, and JavaScript calculators—that people can return to your site and continue to use even when they have no internet connection.

To use the offline application cache, you must create and declare a manifest file. The manifest file is a text file that contains a list of resources to be cached. Declare the manifest file in your HTML using the `manifest` attribute:

```
<html manifest="path/filename">
```

When a user comes to your website for the first time, Safari loads the webpage that declares the manifest—along with all the resources listed in the manifest file—from the web and stores copies in a cache. On subsequent visits, the webpage and resources are loaded directly from the cache, whether the user is online or offline. If the user is online, and the manifest has changed since the user's last visit, a new cache is created and used on the following visit. (You can reload the site from the new cache immediately using JavaScript.)

> **Important** You must modify the manifest file to trigger a cache update. Changing the HTML file that declares the manifest has no effect on users returning to your site unless you change the manifest file as well. If the manifest file is unchanged, the HTML file that declares the manifest is loaded directly from the cache.

When you specify a manifest file, you are telling Safari that your website is designed to operate offline. Consequently, Safari does not attempt to access resources unless they are listed in the manifest and stored in the cache. If you need to access online resources for your website to work correctly, specify the URLs or URL patterns in the `NETWORK` section of the manifest, as described in "Creating a Manifest File" (page 11).

If your website consists of multiple HTML pages that should all be cached in order for your site to operate correctly, list all of the pages in the cache manifest, and include the `manifest` attribute in every one of the HTML pages.

If you are using the offline application cache to speed access to your site for return visitors, instead of using the cache so your site can be used when the user is offline, be sure to specify your network URL pattern as `*`, so that all network access is enabled, and remember to update the manifest file whenever you update your site.

Creating a Manifest File

The manifest file specifies the resources to download and store in the application cache. Resources can include HTML, JavaScript, CSS, and image files. In Safari 5.1 and later, the manifest can specify audio and video media files as well. After the first time a webpage is loaded, the resources specified in the manifest file are loaded directly from the application cache, not from the web server.

The manifest file has the following requirements:

- The file must be served with a MIME type of `text/cache-manifest`. For most web servers, the filename must end in `.manifest`.
- The file must have the same origin (domain and scheme) as the HTML file that declares it.
- The first line of the file must consist of the text `CACHE MANIFEST`.
- Subsequent lines may contain section headings, URLs of resources to cache, network URLs, or comments.
  - Section headings consist of a single word, followed by a colon. Legal values are `CACHE:`, `NETWORK:`, or `FALLBACK:`.
  - Resource URLs can be absolute or relative to the manifest file (*not* necessarily relative to the HTML page that uses the resource).
  - Network URLs may contain an asterisk as a wildcard character.
  - Comments must be on a single line and preceded by the `#` character. Comments may appear in any section of the manifest.

The manifest has one or more sections. There is normally a `CACHE` section, followed by an optional `NETWORK` section. There may also be `FALLBACK` sections and additional `CACHE` sections.

The CACHE Section

A `CACHE` section consists of a list of URLs of resources to be cached. By default, the manifest begins with a `CACHE` section declared implicitly. For example, your manifest may consist of the `CACHE MANIFEST` heading followed by a list of resource URLs, with no section heading.

If your manifest has other section types, followed by a `CACHE` section, however, you must declare any subsequent `CACHE` section explicitly using the `CACHE:` section header.

The `CACHE` sections of your manifest should list the URL of every resource your website needs to operate correctly when the user is offline. Resource URLs can be absolute or relative *to the manifest file*. Each URL must appear on a separate line.

You do not need to include the URL of the HTML file in which the manifest is declared; that file is included in the cache automatically.

The NETWORK Section

If your website requires online resources to operate correctly, you must declare the URLs of the resources in the optional `NETWORK` section of the manifest file. These URLs comprise a whitelist of resources to access using the network.

> **Important** Resources that are not in the cache, and are not included in the whitelist, are not loaded.

A `NETWORK` section begins with the section header `NETWORK:` on a line by itself. This is followed by a list of URLs or URL patterns, each on its own line.

A URL may be a complete file specifier or a prefix. If the URL is a prefix, all files sharing the prefix are on the whitelist. The asterisk character (`*`) may be used as a wildcard that matches all URLs. For example:

- `http://example.com/myDirectory/myFile.jpg` puts `myFile.jpg` on the whitelist.
- `http://example.com/myDirectory/` puts all files in `myDirectory` on the whitelist.
- `http://example.com/` puts all files in `example.com` on the whitelist.
- `*` puts all files on the Internet on the whitelist.

Fallback Sections

A manifest can contain optional `FALLBACK` sections. A `FALLBACK` section contains one or more pairs of URLs. The first URL in each pair is the preferred URL for a resource to be cached. The second URL in the pair is a fallback URL to use if the file is inaccessible using the first URL.

A `FALLBACK` section begins with the section header `FALLBACK:` on a line of its own. Each URL pair appears on a separate line, with the two URLs separated by a space. For example:

```
FALLBACK:

http://example.com/myFile http://server2.example.com/myFile

../media/image2.jpg ../legacy/genericImage.jpg
```

The example just given caches `myFile` from `example.com`, falling back to an alternate server in the same domain if the file is inaccessible using the first server. The example then caches an image from the `media` directory, falling back to a generic image in the `legacy` directory if the preferred image is unavailable.

> **Important** All `FALLBACK` URLs must have the same origin as the manifest file. That is, they must be in the same domain and use the same access scheme.

The fallback section can also be used to add pages from your site to the cache "lazily," as the user loads them. Such use of the fallback section allows the user to revisit previously opened pages while offline, but specifies a default page to use when an offline visitor attempts to open a new page. For example:

```
FALLBACK:

/ /myDirectory/notAvailable.html
```

In the example just given, the URL pattern "`/`" matches all files with the same origin as the manifest file—all the pages on your site. Any page from within your domain is therefore added to the cache when the user loads it and remains available when the user is offline. If the user attempts to load a new page while offline, the file `notAvailable.html` is used instead.

For lazy caching to work, each page to be cached must include the `manifest` attribute in the `<html>` tag.

Example Manifest File

A sample manifest file is shown in Listing 1-1.

Listing 1-1 Sample manifest file

```
CACHE MANIFEST
# This is a comment.
# Cache manifest version 0.0.1
# If you change the version number in this comment,
# the cache manifest is no longer byte-for-byte
# identical.

demoimages/clownfish.jpg
demoimages/clownfishsmall.jpg
demoimages/flowingrock.jpg
demoimages/flowingrocksmall.jpg
demoimages/stones.jpg
```

```
NETWORK:
# All URLs that start with the following lines
# are whitelisted.
http://example.com/examplepath/
http://www.example.org/otherexamplepath/

CACHE:
# Additional items to cache.
demoimages/stonessmall.jpg

FALLBACK:
demoimages/currentImg.jpg images/stockImage.jpg
```

Updating the Cache

When Safari revisits your site, the site is loaded from the cache. If the cache manifest has changed, Safari checks the HTML file that declares the cache, as well as each resource listed in the manifest, to see if any of them has changed.

A file is considered unchanged if it is byte-for-byte identical to the previous version; changing the modification date of a file does not trigger an update. You must change the contents of the file. (Changing a comment is sufficient.)

> **Tip** It's good practice to include a version number in a comment line of your manifest file. Whenever you update the resources your website uses, update the version number in the manifest file as well.

If the manifest file has changed, and checking reveals that a resource file has changed, the new version is copied into the cache.The newly cached resources are not immediately used, however, as that would result in reloading changed resources piecemeal as they are cached. Instead, all of the changed resources are loaded as a group the next time the user revisits your site.

In other words, users who return to your site do not see changes immediately. They see changes when they return a second time. If you want Safari to reload the site with the changed resources the first time a user returns to your site, you can trigger an update using JavaScript.

To interact with the application cache from JavaScript, work with the `applicationCache` object. To force a reload of the contents of the new cache, respond to an `"updateready"` event whose target is the cache by calling the cache's `swapCache()` method, as shown in the following snippet:

```
function updateSite(event) {
    window.applicationCache.swapCache();
}
window.applicationCache.addEventListener('updateready',
    updateSite, false);
```

The `"updateready"` event fires when all of the changed resources have been copied into a new cache. Calling `swapCache()` loads the new cache, causing your webpage to reload,

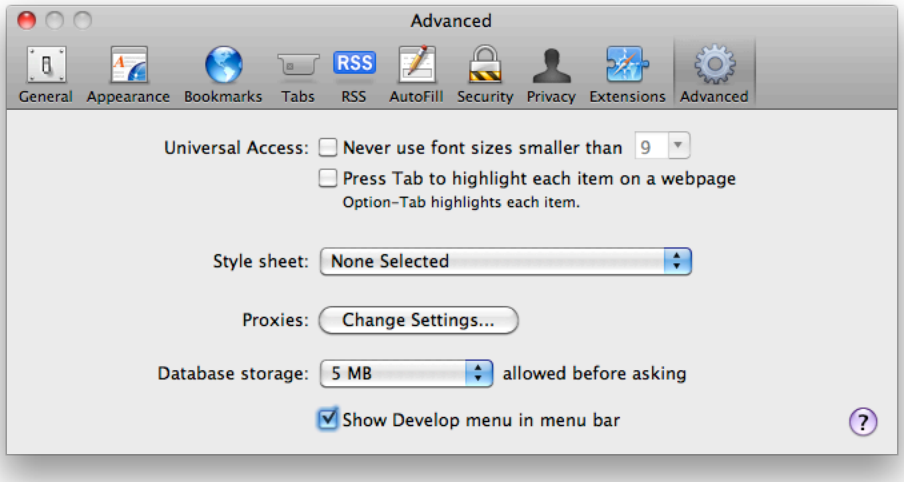
Note that errors can occur when updating the application cache. A resource may be inaccessible, for example, or the manifest file may be modified in the midst of a cache reload. If a failure occurs while downloading the manifest file, its parent HTML file, or a resource specified in the manifest file, the entire update process fails. The browser continues to use the existing version of the application cache and the `"updateready"` event does not fire. If the update is successful, Safari uses the new cache the next time the site is loaded.

Note Using JavaScript to add and remove resources from the application cache is not supported.

See the documentation for `DOMApplicationCache` for a complete list of application cache event types.

Debugging Offline Applications

Debug offline applications using Safari's built-in developer tools. Enable the developer tools in Safari preferences, open your webpage in Safari, and choose Show Web Inspector from the Develop menu.



Click the Resources button in the Web Inspector to inspect the application cache. The display of cached data is not updated dynamically as the cache is modified; to refresh the display, click the "recycle" button in the bottom bar.

For more information about using the Web Inspector, see *Safari Developer Tools Guide*.

Testing Files Locally

You cannot test offline applications by dragging a local webpage into Safari. Safari uses the manifest file only when it is loaded with the MIME type of `text/cache-manifest`, which requires loading the manifest from a web server.

> **Important** The manifest filename should end in `.manifest`.

If you are developing on Mac OS X, your Mac has a built-in web server that you can use to test your manifest file locally. Enable web sharing in System Preferences and drag the website into your Sites directory, then type the IP address of the local web server into Safari's address bar. The site should open, and the manifest file and all the listed resources should load into the application cache and be viewable using the Web Inspector.

To verify the behavior of your website when the user is offline, turn off web sharing in System Preferences and reload your website in Safari. The site should open normally using the cached files.

Testing Offline Applications Remotely

To test an offline application website installed on a remote server, load the site normally, then take Safari offline and load the site again.

Safari operates in offline mode only when there is no Internet connection. To test your web-based application in offline mode, disconnect the device you wish to test from the Internet. For example, put an iOS-based device into Airplane Mode, or create and choose a nonfunctioning network connection in Mac OS X System Preferences.

> **Tip** On iOS-based devices, you can add your offline application website to the home screen to make reloading your site easier.

Monitoring and Logging Cache Events

When debugging offline applications, it can be helpful to monitor and log cache events. The `applicationCache` element generates the following events:

- `"checking"`—Safari is reading the manifest file for the first time or checking to see if it has changed. This event should always fire once.
- `"noupdate"`—The manifest file has not changed.
- `"downloading"`—Safari is downloading a changed resource.
- `"cached"`—Safari has downloaded all listed resources into the cache.
- `"updateready"`—A new copy of the cache is ready to be swapped in.
- `"obsolete"`—The manifest file is code 404 or code 410; the application cache for the site has been deleted.
- `"error"`—An error occurred when loading the manifest, its parent page, or a listed resource, or the manifest file changed while the update was running. The cache update has been aborted.

The snippet in Listing 1-2 can be added to the `<head>` section of your webpage to monitor and log the cache events. For more information about logging using the console API, see *Safari Developer Tools Guide*.

Listing 1-2 Log cache events

```
<script type="text/javascript">

  function logEvent(event) {
      console.log(event.type);
  }
```

```
window.applicationCache.addEventListener('checking',logEvent,false);
window.applicationCache.addEventListener('noupdate',logEvent,false);
window.applicationCache.addEventListener('downloading',logEvent,false);
window.applicationCache.addEventListener('cached',logEvent,false);
window.applicationCache.addEventListener('updateready',logEvent,false);
window.applicationCache.addEventListener('obsolete',logEvent,false);
window.applicationCache.addEventListener('error',logEvent,false);

</script>
```

Key-Value Storage

Key-value storage allows you to store data locally, in either a session store or a local store. The `localStorage` and `sessionStorage` JavaScript objects are functionally identical except in their persistence and scope rules:

- The `localStorage` object is used for long-term storage. Data persists after the window is closed and is shared across all browser windows.
- The `sessionStorage` object is used for ephemeral data related to a single browser window. Data stored in the `sessionStorage` object does not persist after the window is closed and is not shared with other windows.

> **Note** If a new browser window is created when the user clicks a link, that new window gets a copy of the `sessionStorage` object as it exists at the time the window is created. The data is copied, not shared, however, so changes made from either page are not reflected in the session storage for the other page.

Key-value storage is similar to cookie storage; the storage objects are specific to a domain. Because key-value data is not sent back to the server along with every HTTP request, however, it is possible to store larger quantities of data with key-value storage than is practical with cookies. (If you need to send the stored data back to the server, you can do so explicitly using JavaScript and an `XMLHttpRequest` object.)

> **Note** The underlying storage pool is not shared between the `localStorage` and `sessionStorage` objects. Thus, it is not possible to obtain a key stored with `localStorage` by reading it from `sessionStorage` or vice versa. To avoid confusion, you should generally not use the same key name in both storage objects.

Using Key-Value Storage

The examples in this section use local storage. You can change these examples to use session storage by substituting `sessionStorage` for `localStorage` wherever `localStorage` appears in the code.

Storing and Retrieving Values

The easiest way to store key-value data is to synthesize a property of the `sessionStorage` or `localStorage` object by setting the property to a value. For example:

```
localStorage.shirt_size="Medium";
```

In the example just given, if the property `localStorage.shirt_size` does not yet exist, it is created. If the property already exists, its value is reset.

You can also store a value in key-value storage using the `setItem()` method. This method takes two parameters: a key name and a value. For example:

```
// Store the value of the variable "myShirtSize" as the
// value of the field named "shirt_size".
localStorage.setItem("shirt_size", myShirtSize);
```

Storing data can throw an exception if you exceed a browser-specific quota. If the data you are storing is important, you should check for this exception and handle it. For example:

```
try {
    sessionStorage.setItem("shirt_size", myShirtSize);
} catch (e) {
    if (e == QUOTA_EXCEEDED_ERR) {
        alert('Unable to save preferred shirt size.');
    }
}
```

If your key name is a valid JavaScript token (no spaces, punctuation other than underscore, and so on) you can retrieve a value from key-value storage using the key as a property of the `localStorage` or `sessionStorage` object. For example:

```
myShirtSize = localStorage.shirt_size
```

Alternatively, you can use the `getItem()` method to retrieve data, passing in the key:

```
var myShirtSize = localStorage.getItem("shirt_size");
```

If no key-value pair with the specified key exists, `localStorage` returns `null` as the value.

You can find the total number of items in storage for your domain by examining the storage object's `length` property. For example:

```
alert('there are ' + localStorage.length + ' items in the storage array.');
```

You can obtain a list of existing keys using the storage object's `key()` method (again, using the `length` property to determine the number of keys).

```
var keyNames[];
var values[];
// iterate through array
var numKeys = localStorage.length;
for(i=0;i<numKeys;i++) {
    // get key name into an array
    keyNames[i]=localStorage.key(i);
    // use key name to retreive value and store in array
    values[i]=localStorage.getItem(keyNames[i]);
}
```

Deleting Values

There are two ways to delete values from key-value storage: individually or en masse. To delete a single value, call the `removeItem()` method, passing in the key:

```
localStorage.removeItem("shirt_size");
```

To remove all key-value pairs for your domain, call the `clear()` method:

```
sessionStorage.clear();
```

Handling Storage Events

Like cookies, storage objects are a shared resource common to web content served from the same domain. All pages from the same domain share the same local storage object. Frames and inline frames whose contents are from the same origin also share the same session storage object because they descend from the same window.

Because the storage resource is shared, scripts running in multiple page contexts can potentially modify the data stored in a storage object that is actively being scrutinized or modified by a script running on a different page. If your scripts do not notice these changes, you may not get the results you expect.

To this end, storage objects generate an event of type `"storage"` whenever a script adds, deletes, or modifies a value in key-value storage. The event has a `key` property, a `newValue` property, and an `oldValue` property. If `newValue` is `null`, the key-value pair has been removed. If `oldValue` is `null`, a new key-value pair has been added.

To handle storage events, register your handler function as an event listener:

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

Once you have registered an event handler, the specified function (in this case, `myStorage_handler`) is called whenever a script modifies either local or session storage. Here is a simple handler that shows an alert for each field in a `storage` event:

```
function myStorage_handler(evt)
{
    alert('The modified key was '+evt.key);
    alert('The original value was '+evt.oldValue);
    alert('The new value is '+evt.newValue);
    alert('The URL of the page that made the change was '+evt.url);
    alert('The window where the change was made was '+evt.source);
}
```

This example, while simple, shows the five event fields relevant to a storage event. The fields are as follows:

`key`
: The key that was modified. If the session or local storage object is wiped with the `clear` method, the value of `key` is `null`.

`oldValue`
: The previous value of the modified key. If the key is newly created, the value of `oldValue` is `null`. If the storage object is wiped with the `clear` method, the value is also `null`.

`newValue`
: The current (new) value of the modified key. If the key is deleted with the `clear` method, the value of `key` is `null`. If the storage object is wiped with the `clear` method, the value is also `null`.

`url`

The URL of the page that added, modified, or deleted the key. The `url` field is only available if the page that made the change is in the same browsing context (a single tab in a single window). Otherwise, the `url` field value will be `null`.

`source`

The `Window` object containing the script that modified the key. The `source` field is only available if the page that made the change is in the same browsing context (a single tab in a single window). Otherwise, the `source` field value will be `null`.

A Simple Example

The HTML page in Listing 2-1 demonstrates local key-value storage. If you modify the values in either field, they are stored locally (both on modification and on page exit). The values are retrieved from storage when you reload the page.

Listing 2-1 Key-value storage example

```
<html>
<head>
    <title>Key-Value Storage</title>
<script type="text/javascript">
var sizeMenu
var colorMenu

function init() {
    sizeMenu = document.getElementById("sizeMenu");
    colorMenu = document.getElementById("colorMenu");
    if (localStorage.shirtSize)
        sizeMenu.value = localStorage.shirtSize;
    if (localStorage.shirtColor)
        colorMenu.value = localStorage.shirtColor;
}

function saveChoice() {
    localStorage.shirtSize=sizeMenu.value;
    localStorage.shirtColor=colorMenu.value;
```

```
    console.log(localStorage.shirtSize + ", " + localStorage.shirtColor);
}
</script>
</head>

<body onload="init()">
<h2>Local Storage</h2>

<p> Shirt size:
<select id="sizeMenu" onchange="saveChoice()">
    <option value="small">Small</option>
    <option value="medium">Medium</option>
    <option value="large">Large</option>
</select>
</p>

<p> Shirt color:
<select id="colorMenu" onchange="saveChoice()">
    <option value="red">Red</option>
    <option value="white">White</option>
    <option value="blue">Blue</option>
    <option value="tiedye">Tie-Dyed</option>
</select>
</p>
</body>
</html>
```

Debugging Key-Value Storage

You can inspect and debug key-value storage using the Web Inspector, as described in "Debugging Offline Applications" (page 16). The display of stored data is not updated dynamically as the store changes; to refresh the display, click the "recycle" button in the bottom bar.

The Web Inspector shows the storage objects that exist for each domain that has created a local store. Local files share the same domain, so when debugging a website locally you may want to clear the local store during initialization; otherwise the local storage object may contain data created while testing files from another site locally. When inspecting local files, you may see data in local storage used by the Web Inspector itself.

In addition to inspecting the key-value store, you can interactively modify the storage contents using the Web Inspector. You can also use the Web Inspector to interactively debug JavaScript. For details, see *Safari Developer Tools Guide* .

Relational Database Basics

There are many kinds of databases: flat databases, relational databases, object-oriented databases, and so on. Before you can understand relational databases, you must first understand flat databases.

A flat database consists of a single table of information. The rows in the table (also called records) each contain all of the information about a single entry—a single person's name, address, and ZIP code, for example. The row is further divided into fields, each of which represents a particular piece of information about that entry. A `name` field, for example, might contain a person's name.

> **Note** The terms column and field are often used interchangeably, but the term column typically refers collectively to a particular field in *every* row. In other words, the table as a whole is divided into rows and columns, and the intersection of a row and column is called a field.

This design allows you to rapidly access a subset of the information in a table. For example, you might want to get a list of the names and phone numbers of everyone in a particular ZIP code, but you might not care about the rest of the address.

Consider this example of a medical database for an insurance company. A flat database (not relational), might contain a series of records that look like this:

First Name	Last Name	Address	City	State	ZIP
John	Doe	56989 Peach St.	Martin	TN	38237
Jane	Doe	56989 Peach St.	Martin	TN	38237

This example contains two rows, each of which contains information about a single person. Each row contains separate fields for first and last name, address, and so on.

At first glance, this database seems fairly reasonable. However, when you look more closely, it highlights a common problem with flat databases: redundancy. Notice that both of these entries contain the same address, city, state, and ZIP code. Thus, this information (and probably other information such as phone numbers) is duplicated between these two records.

A relational database is designed to maximize efficiency of storage by avoiding this duplication of information. Assuming that John and Jane are members of the same family, you could create a relational version of this information as shown in Table 3-1 and Table 3-2.

Table 3-1 Relational database "family" table

ID	Last_Name	Address	City	State	ZIP
1	Doe	56989 Peach St.	Martin	TN	38237

Table 3-2 Relational database "familymember" table

ID	First_Name	Family_ID
1	John	1
2	Jane	1

Instead of two separate copies of the address, city, state, and ZIP code, the database now contains only one copy. The `Family_ID` fields in the `familymember` table tell you that both John and Jane are members of the family shown in the `family` table whose `ID` field has a value of 1. This relationship between a field in one table and a field in another is where relational databases get their name.

The advantages to such a scheme are twofold. First, by having only one copy of this information, you save storage (though in this case, the savings are minimal). Second, by keeping only a single copy, you reduce the risk of mistakes. When John and Jane have their third child and move to a bigger house, the database user only needs to change the address in one place instead of five. This reduces the risk of making a typo and removes any possibility of failing to update the address of one of their children.

Relationship Models and Schema

When working with relational databases, instead of thinking only about what information your database describes, you should think of the relationships between pieces of information.

When you create a database, you should start by creating a conceptual model, or schema. This schema defines the overall structure of your database in terms of associations between pieces of information.

There are three basic types of relationships in databases: one-to-one, one-to-many (or many-to-one), and many-to-many. In order to use relational databases effectively, you need to think of the information in terms of those types of relationships.

A good way to show the three different types of relationships is to model a student's class schedule.

Here are examples of each type of relationship in the context of a student class schedule:

- **one-to-one relationship**—A student has only one student ID number and a student ID number is associated with only one student.
- **one-to-many relationship**—A teacher teaches many classes, but generally speaking, a class has only one teacher of record.
- **many-to-many relationship**—A student can take more than one class. With few exceptions, each class generally contains more than one student.

Because of the differences in these relationships, the database structures that represent them must also be different to maximize efficiency.

- **one-to-one**—The student ID number should be part of the same table as other information about the student. You should generally break one-to-one information out into a separate table only if one or more of the following is true:
  - You have a large blob of data that is infrequently accessed (for example, a student ID photo) and would otherwise slow down every access to the table as a whole.
  - You have differing security requirements for the information. For example, social security numbers, credit card information, and so on must by stored in a separate database.
  - You have significantly separate sets of information that are used under different conditions. For example, student grade records might take advantage of a table that contains basic information about the student, but would probably not benefit from additional information about the student's housing or financial aid.
- **one-to-many**—You should really think of this relationship as "many-to-one." Instead of thinking about a teacher having multiple classes, think about each class having a single teacher. This may seem counterintuitive at first, but it makes sense once you see an example such as the one shown in Table 3-3.

Table 3-3 The "classes" table

ID	Teacher_ID	Class_Number	Class_Name
1	1	MUS111	Music Appreciation: Classical Music
2	1	MUS112	Music Appreciation: Music of the World

Notice that each class is associated with a teacher ID. This should contain an ID from the `teacher` table (not shown). Thus, by creating a relationship from each of the many classes to a single teacher, you have changed the relationship from an unmanageable one-to-many relationship into a very simple many-to-one relationship.

- **many-to-many**—Many-to-many relationships are essentially a convenient fiction. Your first instinct would be to somehow make each class contain a list of student IDs that were associated with this class. This, however, is not a workable approach because relational databases (or at least those based on SQL) do not support any notion of a list.

  Indeed, someone thinking about this from a flat database perspective might think of a class schedule as a table containing a student's name and a list of classes. In a relational database world, however, you would view it as a collection of students, a collection of classes, and a collection of lists that associate each person with a particular class or classes.

  Thus, you should think of a many-to-many relationship as multiple collections of many-to-one relationships. In the case of students and classes, instead of having a class associated with multiple students or a student associated with multiple classes, you instead have a third entity—a "student class" entity. This naming tells you that the entity expresses a relationship between a student and a class. An example of a `student_class` table is shown in Table 3-4 (page 29).

  Table 3-4 The "student_class" table

ID	Student_ID	Class_ID
1	19	37

  Instead of associating either the student or the class with multiple entries, you now simply have multiple `student_class` entries. Each entry associates one student with one class. Thus, you effectively have multiple students, each associated with multiple classes in a many-to-many relationship, but you did it by creating multiple instances of many-to-one relationship pairs.

SQL Basics

The Structured Query Language, or SQL, is a standard syntax for querying or modifying the contents of a database. Using SQL, you can write software that interacts with a database without worrying about the underlying database architecture; the same basic queries can be executed on every database from SQLite to Oracle, provided that you limit yourself to the basic core queries and data types.

The JavaScript database uses SQLite internally to store information. SQLite is a lightweight database architecture that stores each database in a separate file. For more information about SQLite, see the SQLite website at http://www.sqlite.org/.

For syntax descriptions in this section:

- Text enclosed in square brackets (`[]`)is optional.
- Lowercase text represents information that you choose.

- An ellipsis (`...`) means that you can specify additional parameters similar to the preceding parameter.
- Uppercase text and all other symbols are literal text and keywords that you must enter exactly as-is. (These keywords are not case sensitive, however.)

The most common SQL queries are `CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE`, `DELETE`, and `DROP TABLE`. These queries are described in the sections that follow.

> **Note** To easily distinguish between language keywords and other content, commands and other language keywords are traditionally capitalized in SQL queries. The SQL language, however, is case insensitive, so `SELECT` and `select` are equivalent. (Some SQL implementations do handle table and column names in a case-sensitive fashion, however, so you should always be consistent with those.)

For a complete list of SQL commands supported by SQLite and for additional options to the commands described above, see the SQLite language support specification at http://www.sqlite.org/lang.html.

CREATE TABLE Query

Creates a table.

```
CREATE [TEMP[ORARY]] TABLE [IF NOT EXISTS] table_name (
    column_name column_type [constraint],
    ...
    );
```

The most common values for `constraint` are `PRIMARY KEY`, `NOT NULL`, `UNIQUE`, and `AUTOINCREMENT`. Column constraints are optional. For example, the following `CREATE` command creates the table described by Table 3-1 in "Relationship Models and Schema" (page 27):

```
CREATE TABLE IF NOT EXISTS family (
    ID INTEGER PRIMARY KEY,
    Last_Name NVARCHAR(63) KEY,
    Address NVARCHAR(255),
    City NVARCHAR(63),
    State NVARCHAR(2),
    Zip NVARCHAR(10));
```

Each table should contain a primary key. A primary key implicitly has the `UNIQUE`, and `NOT NULL` properties set. It doesn't hurt to state them explicitly, as other database implementations require `NOT NULL` to be stated explicitly. This column must be of type `INTEGER` (at least in SQLite—other SQL implementations use different integer data types).

Notice that this table does not specify the `AUTOINCREMENT` option for its primary key. SQLite does not support the `AUTOINCREMENT` keyword, but SQLite implicitly enables auto-increment behavior when `PRIMARY KEY` is specified.

The data types supported by SQLite are as follows:

`BLOB`
: A large block of binary data.

`BOOL`
: A boolean (true or false) value.

`CLOB`
: A large block of (typically 7-bit ASCII) text.

`FLOAT`
: A floating-point number.

`INTEGER`
: An integer value.

Note Although integer values are stored internally as integers, all numerical comparisons are performed using 64-bit floating-point values. This may cause precision loss for very large numeric values (>15 digits). If you need to compare such large numbers, you should store them as a string and compare their length (to detect magnitude differences) prior to comparing their value.

`NATIONAL VARYING CHARACTER` or `NVCHAR`
: A Unicode (UTF-8) string of variable length (generally short). This data type requires a length parameter that provides an upper bound for the maximum data length. For example, the following statement declares a column named `Fahrenheit` whose UTF-8 data cannot exceed 451 characters in length:

```
...
    Fahrenheit NVARCHAR(451),
...
```

Note Unlike some SQL implementations, SQLite does not enforce this maximum length and does not truncate data to the length specified. However, you should still set reasonable bounds to avoid the risk of compatibility problems in the future.

SQLite also does not enforce valid Unicode encoding for this data. In effect, it is treated just like any other text unless and until you use a UTF-16 collating sequence (controlled by the `COLLATE` keyword) or SQL function. Collating sequences and SQL functions are beyond the scope of this document. See the SQLite documentation at http://www.sqlite.org/docs.html for more information.

`NUMERIC`

A fixed-precision decimal value that is expected to hold values of a given precision. Note that SQLite does not enforce this in any way.

`REAL`

A floating-point value. This is stored as a 64-bit (8-byte) IEEE double-precision floating point value.

`VARCHAR` or `VARYING CHARACTER`

A (generally short) variable-length block of text. This data type requires a length parameter that provides an upper bound for the maximum data length. For example, the following statement declares a column named `Douglas` whose data cannot exceed 42 characters in length:

```
...
    Douglas VARCHAR(42),
...
```

Note Unlike some SQL implementations, SQLite does not enforce this maximum length and does not truncate data to the length specified. However, you should still set reasonable bounds to avoid the risk of compatibility problems in the future.

To avoid unexpected behavior, you should be careful to store only numeric values into numeric (`REAL`, `INTEGER`, and so on) columns. If you attempt to store a non-numeric value into a numeric column, the value is stored as text. SQLite does not warn you when this happens. In effect, although SQLite supports typed columns, the types are not enforced in any significant way, though integer values are converted to floating point values when stored in a `REAL` column.

INSERT Query

Inserts a new row into a table.

```
INSERT INTO table_name (column_1, ...)
```

```
    VALUES (value_1, ...);
```

For example, to store the values shown in Table 3-1 in "Relationship Models and Schema" (page 27), you would use the following query:

```
INSERT INTO family (Last_Name, Address, City, State, Zip)
    VALUES ('Doe', '56989 Peach St.', 'Martin', 'TN', '38237');
```

You should notice that all non-numeric values must be surrounded by quotation marks (single or double). This is described further in "SQL Security and Quoting Characters in Strings" (page 37).

SELECT Query

Retrieves rows (or portions thereof) from a table or tables.

```
SELECT column_1 [, ...] from table_1 [, ...] WHERE expression;
```

Each SELECT query returns an result array (essentially a temporary table) that contains one entry for every row in the database table that matches the provided expression. Each entry is itself an array that contains the values stored in the specified columns within that database row. For example, the following SQL query returns an array of (name, age) pairs for every row in the people table where the value in the age column is greater than 18:

```
SELECT name, age FROM people WHERE age > 18;
```

Here are some other relatively straightforward examples of expressions that are valid in SELECT queries:

```
# Alphabetic comparison
SELECT name, age FROM people WHERE name < "Alfalfa";

# Equality and inequality
SELECT name, age FROM people WHERE age = 18;
SELECT name, age FROM people WHERE age != 18;

# Combinations
SELECT name, age FROM people WHERE (age > 18 OR (age > 55 AND AGE <= 65));
```

The complete expression syntax is beyond the scope of this document. For further details, see the SQLite language support specification at http://www.sqlite.org/lang.html.

To select all columns, you can use an asterisk (*) instead of specifying a list of column names. For example:

```
SELECT * FROM people WHERE age > 18;
```

You can also use a `SELECT` query to query multiple tables at once. For example:

```
SELECT First_Name, Last_Name FROM family, familymember WHERE familymember.Family_ID
 = family.ID AND familymember.ID = 2;
```

The query creates a temporary joined table that contains one row for each combination of a single row from the `family` table and a single row from the `familymember` table in which the combined row pair matches the specified constraints (`WHERE` clause). It then returns an array containing the fields `First_Name` and `Last_Name` from that temporary table. Only rows in the `familymember` table whose `ID` value is `2` are included in the output; all other rows in this table are ignored. Similarly, only rows in the `family` table that match against at least one of the returned rows from the `familymember` table are included; other rows are ignored.

For example, if you provide tables containing the values shown in Table 3-1 (page 27) and Table 3-2 (page 27) in "Relationship Models and Schema" (page 27), this would return only a single row of data containing the values ('Jane', 'Doe').

Notice the constraint `family.ID`. If a column name appears in multiple tables, you cannot just use the field name as part of a `WHERE` constraint because the SQL database has no way of knowing which ID field to compare. Instead, you must specify the table name as part of the column name, in the form `table_name.column_name`. Similarly, in the field list, you can specify `table_name.*` to request all of the rows in the table called `table_name`. For example:

```
SELECT familymember.*, Last_Name from family, familymember WHERE ...
```

UPDATE Query

Changes values within an existing table row.

```
UPDATE table_name SET column_1=value_1 [, ...]
    [WHERE expression];
```

If you do not specify a `WHERE` expression, the specified change is made to every row in the table. The expression syntax is the same as for the `SELECT` statement. For example, if someone gets married, you might change that person's `Family_ID` field with a query like this one:

```
UPDATE familymember set Family_ID=32767 WHERE ID=179;
```

DELETE Query

Deletes a row or rows from a table.

```
DELETE FROM table_name [WHERE expression];
```

If you do not specify a `WHERE` expression, this statement deletes every row in the table. The expression syntax is the same as for the `SELECT` statement. For example, if someone drops their membership in an organization, you might remove them from the roster with a query like this one:

```
DELETE FROM people WHERE ID=973;
```

DROP TABLE Query

Deletes an entire table.

```
DROP TABLE table_name;
```

For example, if your code copies the contents of a table into a new table with a different name, then deletes the old table, you might delete the old table like this:

```
DROP TABLE roster_2007;
```

Warning It is not possible to undo this operation. Be *absolutely sure* that you are deleting the correct table before you execute a query like this one!

Transaction Processing

Most modern relational databases have the notion of a transaction. A transaction is defined as an atomic unit of work that either completes or does not complete. If any part of a transaction fails, the changes it made are rolled back—restored to their original state prior to the beginning of the transaction.

Transactions prevent something from being halfway completed. This is particularly important with relational databases because changes can span multiple tables.

For example, if you are updating someone's class schedule, inserting a new `student_class` record might succeed, but a verification step might fail because the class itself was deleted by a previous change. Obviously, making the change would be harmful, so the changes to the first table are rolled back.

Other common reasons for failures include:

- Invalid values—a string where the database expected a number, for example, could cause a failure if the database checks for this. (SQLite does *not*, however.)
- Constraint failure—for example, if the class number column is marked with the `UNIQUE` keyword, any query that attempts to insert a second class with the same class number as an existing class fails.
- Syntax error—if the syntax of a query is invalid, the query fails.

The mechanism for performing a transaction is database-specific. In the case of the JavaScript Database, transactions are built into the query API itself, as described in "Executing a Query" (page 42).

Relational Databases and Object-Oriented Programming

Relational databases and data structures inside applications should be closely coupled to avoid problems. The best in-memory architecture is generally an object for each row in each table. This means that each object in your code would have the same relationships with other objects that the underlying database entries themselves have with each other.

Tying database objects to data structures is important for two reasons:

- It reduces the likelihood of conflicting information. You can be certain that no two objects will ever point to the same information in the database. Thus, changing one object will never require changing another object.
- It makes it a lot easier to keep track of what data is stored in which table when you are debugging.

It is best to keep the names of tables and classes as similar as possible. Similarly, it is best to keep the names of instance variables as similar as possible to the names of the database fields whose contents they contain.

SQL Security and Quoting Characters in Strings

When working with user-entered strings, additional care is needed. Because strings can contain arbitrary characters, it is possible to construct a value that, if handled incorrectly by your code, could produce unforeseen side effects. Consider the following SQL query:

```
UPDATE MyTable SET MyValue='VARIABLE_VALUE';
```

Suppose for a moment that your code substitutes a user-entered string in place of `VARIABLE_VALUE` without any additional processing. The user enters the following value:

```
'; DROP TABLE MyTable; --
```

The single quote mark terminates the value to be stored in `MyValue`. The semicolon ends the command. The next command deletes the table entirely. Finally, the two hyphens cause the remainder of the line (the trailing `';`) to be treated as a comment and ignored. Clearly, allowing a user to delete a table in your database is an undesirable side effect. This is known as a SQL injection attack because it allows arbitrary SQL commands to be injected into your application as though your application sent them explicitly.

When you perform actual queries using user-provided data, to avoid mistakes, you should use placeholders for any user-provided values and rely on whatever SQL query API you are using to quote them properly. In the case of the JavaScript database API, this process is described in "Executing a Query" (page 42).

If you need to manually insert strings with constant string values, however, using placeholders is overkill and can make the query harder to read. To insert these strings manually, use double-quote marks around any strings that contain a single-quote mark, and vice-versa. In the rare event that you must manually insert a string that contains both types of quotation marks, use single quotes, but add a backslash before every single-quote mark within the string. For example, to insert the value

```
"It's a boy," he whispered softly.
```

you would write it like this:

```
'"It\'s a boy," he whispered softly.'
```

Note Because the backslash is treated as a special quote character, you *must* similarly quote any backslashes within the string by adding a second backslash. If you only quote the single quote mark, you can still be the victim of a slightly tweaked injection attack like this one:

```
\'; DROP TABLE MyTable; --
```

If you merely added a backslash before the single quote, the backslash before it would quote that backslash, and the single quote would still end the string and allow the `DROP TABLE` command to execute.

Using the JavaScript Database

Beginning in Safari 3.1 and iOS 2.0, Safari supports the HTML5 JavaScript database class. The JavaScript database class, based on SQLite, provides a relational database intended for local storage of content that is too large to conveniently store in cookies (or is too important to risk accidentally deleting when the user clears out his or her cookies).

Because it provides a relational database model, the JavaScript database class makes it easy to work with complex, interconnected data in a webpage. You might use it as an alternative to storing user-generated data on the server (in a text editor, for example), or you might use it as a high-speed local cache of information the user has recently queried from a server-side database.

The sections in this chapter guide you through the basic steps of creating a JavaScript-based application that takes advantage of the JavaScript database.

> **Note** This chapter covers only the JavaScript API for making SQL queries, not the different types of queries themselves. At this level, all queries behave similarly (except that not all queries provide any actual data to their data callbacks).
>
> For more detailed coverage of what SQL queries you can actually make, read "Relational Database Basics" (page 26). That chapter provides an assortment of queries that cover most common database tasks.

Creating and Opening a Database

Before you can use a database or create tables within the database, you must first open a connection to the database. When you open a database, an empty database is automatically created if the database you request does not exist. Thus, the processes for opening and creating a database are identical.

To open a database, you must obtain a database object with the `openDatabase` method as follows:

Listing 4-1 Creating and opening a database

```
try {
    if (!window.openDatabase) {
        alert('not supported');
```

```
    } else {
        var shortName = 'mydatabase';
        var version = '1.0';
        var displayName = 'My Important Database';
        var maxSize = 65536; // in bytes
        var db = openDatabase(shortName, version, displayName, maxSize);

        // You should have a database instance in db.
    }
} catch(e) {
    // Error handling code goes here.
    if (e == 2) {
        // Version number mismatch.
        alert("Invalid database version.");
    } else {
        alert("Unknown error "+e+".");
    }
    return;
}

alert("Database is: "+db);
```

For now you should set the version number field to 1.0; database versioning is described in more detail in "Working With Database Versions" (page 48).

The short name is the name for your database as stored on disk (usually in `~/Library/Safari/Databases/`). This argument controls which database you are accessing.

The display name field contains a name to be used by the browser if it needs to describe your database in any user interaction, such as asking permission to enlarge the database.

The maximum size field tells the browser the size to which you expect your database to grow. The browser normally prevents a runaway web application from using excessive local resources by setting limits on the size of each site's database. When a database change would cause the database to exceed that limit, the user is notified and asked for permission to allow the database to grow further.

If you know that you are going to be filling the database with a lot of content, you should specify an ample size here. By so doing, the user is only asked for permission once when creating the database instead of every few megabytes as the database grows.

The browser may set limits on how large a value you can specify for this field, but the details of these limits are not yet fully defined.

Creating Tables

The remainder of this chapter assumes a database that contains a single table with the following schema:

```
CREATE TABLE people(
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL DEFAULT "John Doe",
    shirt TEXT NOT NULL DEFAULT "Purple"
);
```

Note For more information about schemas, see "Relational Database Basics" (page 26).

You can create this table and insert a few initial values with the following functions:

Listing 4-2 Creating a SQL table

```
function nullDataHandler(transaction, results) { }


function createTables(db)
{
    db.transaction(
        function (transaction) {


            /* The first query causes the transaction to (intentionally) fail if
the table exists. */
          transaction.executeSql('CREATE TABLE people(id INTEGER NOT NULL PRIMARY
 KEY AUTOINCREMENT, name TEXT NOT NULL DEFAULT "John Doe", shirt TEXT NOT NULL
DEFAULT "Purple");', [], nullDataHandler, errorHandler);
            /* These insertions will be skipped if the table already exists. */
```

```
            transaction.executeSql('insert into people (name, shirt) VALUES ("Joe",
 "Green");', [], nullDataHandler, errorHandler);
            transaction.executeSql('insert into people (name, shirt) VALUES ("Mark",
 "Blue");', [], nullDataHandler, errorHandler);
            transaction.executeSql('insert into people (name, shirt) VALUES ("Phil",
 "Orange");', [], nullDataHandler, errorHandler);
            transaction.executeSql('insert into people (name, shirt) VALUES ("jdoe",
 "Purple");', [], nullDataHandler, errorHandler);
        }
    );
}
```

The `errorHandler` function is shown and explained in "Per-Query Error Callbacks" (page 46).

Executing a Query

Executing a SQL query is fairly straightforward. All queries must be part of a transaction (though the transaction may contain only a single query if desired).

You could then modify the value as follows:

Listing 4-3 Changing values in a table

```
var name = 'jdoe';
var shirt = 'fuschia';

db.transaction(
    function (transaction) {
        transaction.executeSql("UPDATE people set shirt=? where name=?;",
            [ shirt, name ]); // array of values for the ? placeholders
    }
);
```

Notice that this transaction provides no data or error handlers. These handlers are entirely optional, and may be omitted if you don't care about finding out whether an error occurs in a particular statement. (You can still detect a failure of the entire transaction, as described in "Transaction Callbacks" (page 47).)

However, if you want to execute a query that returns data (a SELECT query, for example), you must use a data callback to process the results. This process is described in "Handling Result Data" (page 43).

Handling Result Data

The examples in the previous section did not return any data. Queries that return data are a little bit more complicated.

As noted in previous sections, every query *must* be part of a transaction. You must provide a callback routine to handle the data returned by that transaction—store it, display it, or send it to remote server, for example.

The following code prints a list of names where the value of the shirt field is "Green":

Listing 4-4 SQL query result and error handlers

```
function errorHandler(transaction, error)
{
    // error.message is a human-readable string.
    // error.code is a numeric error code
    alert('Oops.  Error was '+error.message+' (Code '+error.code+')');

    // Handle errors here
    var we_think_this_error_is_fatal = true;
    if (we_think_this_error_is_fatal) return true;
    return false;
}

function dataHandler(transaction, results)
{
    // Handle the results
    var string = "Green shirt list contains the following people:\n\n";
    for (var i=0; i<results.rows.length; i++) {
        // Each row is a standard JavaScript array indexed by
        // column names.
        var row = results.rows.item(i);
        string = string + row['name'] + " (ID "+row['id']+")\n";
    }
```

```
    alert(string);
}

db.transaction(
    function (transaction) {
        transaction.executeSql("SELECT * from people where shirt='Green';",
            [], // array of values for the ? placeholders
            dataHandler, errorHandler);
    }
);
```

> **Note** The `errorHandler` callback may be omitted in the call to `executeSql` if you don't want to capture errors.

This is, of course, a fairly simple example. Things get slightly more complicated when you are performing dependent queries, such as creating a new row in one table and inserting that row's ID into a field in another table to create a relationship between those rows. For more complex examples, see the appendix.

To obtain the number of rows modified by a query, check the `rowsAffected` field of the result set object. To obtain the ID of the last row inserted, check the `insertId` field of the result set object, then perform the second query from within the data callback of the first query. For example:

Listing 4-5 SQL insert query example

```
db.transaction(
    function (transaction) {
        transaction.executeSql('INSERT into tbl_a (name) VALUES ( ? );',
            [ document.getElementById('nameElt').innerHTML ],
            function (transaction, resultSet) {
                if (!resultSet.rowsAffected) {
                    // Previous insert failed. Bail.
                    alert('No rows affected!');
                    return false;
                }
                alert('insert ID was '+resultSet.insertId);
```

```
                transaction.executeSql('INSERT into tbl_b (name_id, color) VALUES
 (?, ?);',
                    [ resultSet.insertId,
                      document.getElementById('colorElt').innerHTML ],
                    nullDataHandler, errorHandler);
            }, errorHandler);
    }, transactionErrorCallback, proveIt);
}
```

One more issue that you may run into is multiple tables that contain columns with the same name. Because result rows are indexed by column name, you *must* alias any such columns to unique names if you want to access them. For example, the following query:

```
SELECT * FROM tbl_a,tbl_b ...
```

does not usefully allow access to `tbl_a.id` and `tbl_b.id`, but:

```
SELECT tbl_a.id AS tbl_a_id, tbl_b.id AS tbl_b_id, * FROM tbl_a, tbl_b ...
```

provides unique names for the `id` fields so that you can access them. The following snippet is an example of this query in actual use:

Listing 4-6 SQL query with aliased field names

```
function testAliases(){
        var db = getDB();

        if (!db) {
                alert('Could not open database connection.');
        }

db.transaction(
    function (transaction) {
       var query="SELECT tbl_a.id AS tbl_a_id, tbl_b.id AS tbl_b_id, * FROM tbl_a,
 tbl_b where tbl_b.name_id = tbl_a
.id;";
```

```
        transaction.executeSql(query, [],
                function (transaction, resultSet) {
                        var string = "";
                        for (var i=0; i<resultSet.rows.length; i++) {
                                var row = resultSet.rows.item(i);
                                alert('Alias test: Name: '+row['name']+'
('+row['tbl_a_id']+') Color: '+row['color']+' ('+row['tbl_b_id']+')');
                                // string = string + "ID: "+row['id']+" A_ID:
"+row['tbl_a_id']+" B_ID: "+row['tbl_b_id']+"\n";
                        }
                        // alert("Alias test:\n"+string);
                }, errorHandler);
    }, transactionErrorCallback);
}
```

Handling Errors

You can handle errors at two levels: at the query level and at the transaction level.

Per-Query Error Callbacks

The per-query error-handling callback is rather straightforward. If the callback returns `true`, the entire transaction is rolled back. If the callback returns `false`, the transaction continues as if nothing had gone wrong.

Thus, if you are executing a query that is optional—if a failure of that particular query should not cause the transaction to fail—you should pass in a callback that returns `false`. If a failure of the query should cause the entire transaction to fail, you should pass in a callback that returns `true`.

Of course, you can also pass in a callback that decides whether to return `true` or `false` depending on the nature of the error.

If you do not provide an error callback at all, the error is treated as fatal and causes the transaction to roll back.

For a sample snippet, see `errorHandler` in Listing 4-4 (page 43).

For a list of possible error codes that can appear in the `error.code` field, see "Error Codes" (page 47).

Transaction Error Callbacks

In addition to handling errors on a per-query basis (as described in "Per-Query Error Callbacks" (page 46)), you can also check for success or failure of the entire transaction.

For example:

Listing 4-7 Sample transaction error callback

```
function myTransactionErrorCallback(error)
{
    alert('Oops.  Error was '+error.message+' (Code '+error.code+')');
}

function myTransactionSuccessCallback()
{
    alert("J. Doe's shirt is Mauve.");
}

var name = 'jdoe';
var shirt = 'mauve';

db.transaction(
    function (transaction) {
        transaction.executeSql("UPDATE people set shirt=? where name=?;",
            [ shirt, name ]); // array of values for the ? placeholders
    }, myTransactionErrorCallback, myTransactionSuccessCallback
);
```

Upon successful completion of the transaction, the success callback is called. If the transaction fails because any portion thereof fails, the error callback is called instead.

As with the error callback for individual queries, the transaction error callback takes an error object parameter. For a list of possible error codes that can appear in the `error.code` field, see "Error Codes" (page 47).

Error Codes

The error codes currently defined are as follows:

`0`

Other non-database-related error.

`1`

Other database-related error.

`2`

The version of the database is not the version that you requested.

`3`

Data set too large. There are limits in place on the maximum result size that can be returned by a single query. If you see this error, you should either use the `LIMIT` and `OFFSET` constraints in the query to reduce the number of results returned or rewrite the query to return a more specific subset of the results.

`4`

Storage limit exceeded. Either the space available for storage is exhausted or the user declined to allow the database to grow beyond the existing limit.

`5`

Lock contention error. If the first query in a transaction does not modify data, the transaction takes a read-write lock for reading. It then upgrades that lock to a writer lock if a subsequent query attempts to modify data. If another query takes a writer lock ahead of it, any reads prior to that point are untrustworthy, so the entire transaction must be repeated. If you receive this error, you should retry the transaction.

`6`

Constraint failure. This occurs when an `INSERT`, `UPDATE`, or `REPLACE` query results in an empty set because a constraint on a table could not be met. For example, you might receive this error if it would cause two rows to contain the same non-null value in a column marked as the primary key or marked with the `UNIQUE` constraint.

Additional error codes may be added in the future as the need arises.

Working With Database Versions

To make it easier for you to enhance your application without breaking compatibility with earlier versions of your databases, the JavaScript database supports versioning. With this support, you can modify the schema atomically, making changes in the process of doing so.

When you open a database, if the existing version matches the version you specify, the database is opened. Otherwise, the `openDatabase` call throws an exception with a value of `2`. See "Error Codes" (page 47) for more possible exception values.

If you specify an empty string for the version, the database is opened regardless of the database version. You can then query the version by examining the database object's version property. For example:

Listing 4-8 Obtaining the current database version

```
var db = openDatabase(shortName, "", displayName, maxSize);
var version = db.version; // For example, "1.0"
```

Once you know what version you are dealing with, you can atomically update the database to a new version (optionally with a modified schema or modified data) by calling the `changeVersion` method.

For example:

Listing 4-9 Changing database versions

```
function cv_1_0_2_0(transaction)
{
        transaction.executeSql('alter table people rename to person', [],
nullDataHandler, errorHandler);
}

function oops_1_0_2_0(error)
{
    alert('oops in 1.0 -> 2.0 conversion.  Error was '+error.message);
    alert('DB Version: '+db.version);
    return true; // treat all errors as fatal
}

function success_1_0_2_0()
{
    alert("Database changed from version 1.0 to version 2.0.");
}

function testVersionChange()
{
    var db = getDB();

    if (!db) {
```

```
        alert('Could not open database connection.');
    }

    if (db.changeVersion) {
        alert('cv possible.');
    } else {
        alert('version changes not possible in this browser version.');
    }
    if (db.version == "1.0") {
        try {
            // comment out for crash recovery.
            db.changeVersion("1.0", "2.0", cv_1_0_2_0, oops_1_0_2_0,
success_1_0_2_0);
        } catch(e) {
            alert('changeversion 1.0 -> 2.0 failed');
            alert('DB Version: '+db.version);
        }
    }
}
```

Note Calling the above function renames the table `people` to `person`. If you create a page containing the examples from this chapter, the other code will recreate the `people` table on the next page load, and a second rename will fail because the `person` table will already exist from the previous rename. Thus, to test this function more than once, you would have to execute the query `DROP TABLE person;` prior to renaming the `people` table.

In some versions of Safari, the database version field does not change after a `changeVersion` call until you reload the page. Usually, this is not a problem. However, it is a problem if you call the `changeVersion` method more than once.

Unfortunately, the only way for your code to see the new version number is by closing the browser window. If you get an error code 2 (see "Error Codes" (page 47)) and the database version you passed in for the old version matches the version in `db.version`, you should either assume that the version change already happened or display an alert instructing the user to close and reopen the browser window.

A Complete Example

For a complete example of basic JavaScript database operations, see "Database Example: A Simple Text Editor" (page 53).

Document Revision History

This table describes the changes to *Safari Client-Side Storage and Offline Applications Programming Guide*.

Date	Notes
2011-09-21	Corrected minor typos.
2011-08-23	Updated description of offline application cache and key-value storage. Updated code samples.
2011-07-12	Added a note describing how to take Safari offline.
2010-08-20	Applied minor edits throughout.
2010-01-20	Minor edits.
2009-11-17	Added Companion Files archive for easier assembly of examples. Made minor typographical and organizational fixes.
2009-10-19	Fixed a few typos.
2009-06-24	Improved the offline app cache documentation.
2009-06-08	Fixed various typographical errors.
2009-02-20	Updated for Safari 4.0. Added description of HTML 5 offline storage and offline applications.
2009-01-06	Expanded SQL syntax coverage. Corrected database example instructions.
2008-03-18	TBD

Database Example: A Simple Text Editor

This example shows a practical, real-world example of how to use the SQL database support. This example contains a very simple HTML editor that stores its content in a local database. This example also demonstrates how to tell Safari about unsaved edits to user-entered content.

This example builds upon the example in the sample code project *HTML Editing Toolbar*, available from the ADC Reference Library. To avoid code duplication, the code from that example is not repeated here. The HTML Editing Toolbar creates an editable region in an HTML page and displays a toolbar with various editing controls.

To create this example, either download the attached Companion Files archive or perform the following steps:

1. Download the *HTML Editing Toolbar* sample and extract the contents of the archive.
2. From the toolbar project folder, copy the files `FancyToolbar.js` and `FancyToolbar.css` into a new folder.

   Also copy the folder `FancyToolbarImages`.

   You do not need to copy the `index.html` or `content.html` files provided by that project.
3. Add a save button in the toolbar. This change is described in "Adding a Save Button to FancyToolbar.js" (page 53).
4. Add the `index.html` and `SQLStore.js` files into the same directory. You can find listings for these files in "Creating the index.html File" (page 54) and "Creating the SQLStore.js File" (page 56).

To use the editor, open the `index.html` file in Safari. Click the Create New File link to create a new "file". Edit as desired, and click the save button in the toolbar.

Next, reload the `index.html` page. You should see the newly created file in the list of available files. If you click on its name, you will see the text you just edited.

Adding a Save Button to FancyToolbar.js

In the the `FancyToolbar.js` (which you should have copied from the HTML Editing Toolbar sample previously), you need to add a few lines of code to add a Save button to the toolbar it displays.

Immediately *before* the following line, which is near the bottom of the function `setupIfNeeded`:

```
    this.toolbarElement.appendChild(toolbarArea);
```

add the following block of code:

Listing A-1 Additions to FancyToolbar.js

```
    this.saveButton = document.createElement("button");
    this.saveButton.appendChild(document.createTextNode("Save"));
    this.saveButton.className = "fancy-toolbar-button fancy-toolbar-button-save";
    this.saveButton.addEventListener("click", function(event) { saveFile() },
false);
    toolbarArea.appendChild(this.saveButton);
```

Creating the index.html File

This file provides some basic HTML elements that are used by the JavaScript code to display text and accept user input. Save the following as `index.html` (or any other name you choose):

Listing A-2 index.html

```
<html><head><title>JavaScript SQL Text Editor</title>
<script language="javascript" type="text/javascript" src="FancyToolbar.js"></script>
<script language="javascript" type="text/javascript" src="SQLStore.js"></script>
<link rel="stylesheet" type="text/css" href="FancyToolbar.css">

<style>
body {
    // margin: 80px;
    // background-color: rgb(153, 255, 255);
}

iframe.editable {
    width: 80%;
```

```
    height: 300px;
    margin-top: 60px;
    margin-left: 20px;
    margin-right: 20px;
    margin-bottom: 20px;
}

table.filetable {
    border-collapse: collapse;
}

tr.filerow {
    border-collapse: collapse;
}

td.filelinkcell {
    border-collapse: collapse;
    border-right: 1px solid #808080;
    border-bottom: 1px solid #808080;
    border-top: 1px solid #808080;
}

td.filenamecell {
    border-collapse: collapse;
    padding-right: 20px;
    border-bottom: 1px solid #808080;
    border-top: 1px solid #808080;
    border-left: 1px solid #808080;
    padding-left: 10px;
    padding-right: 30px;
}
</style>

</head><body onload="initDB(); setupEventListeners(); chooseDialog();">
```

```
<div id="controldiv"></div>
<iframe id="contentdiv" style="display: none" class="editable"></iframe>

<div id="origcontentdiv" style="display: none"></div>
<div id="tempdata"></div>



</body>
</html>
```

Creating the SQLStore.js File

This script contains all of the database functionality for this example. The functions here are called from `index.html` and `FancyToolbar.js`.

The major functions are:

- `initDB`—opens a connection to the database and calls `createTables` to create tables if needed.
- `createTables`—creates tables in the database if they do not exist.
- `chooseDialog`—displays a "file" selection dialog
- `deleteFile`—displays a deletion confirmation dialog
- `reallyDelete`—flags a "file" for deletion
- `createNewFileAction`—creates a new "file" entry
- `saveFile`—saves a "file" into the database.
- `loadFile`—loads a "file" from the database.

In addition to these functions, this example contains several other functions that serve minor roles in modifying the HTML content or handling results and errors.

The function `saveChangesDialog` is also interesting to web application developers. It demonstrates one way to determine whether a user has made unsaved changes to user-entered content and to display a dialog allowing the user to choose whether to leave the page in such a state.

Save the following file as `SQLStore.js` (or modify the `index.html` file to refer to the name you choose):

Listing A-3 SQLStore.js

```
var systemDB;

/*! Initialize the systemDB global variable. */
function initDB()
{


try {
    if (!window.openDatabase) {
        alert('not supported');
    } else {
        var shortName = 'mydatabase';
        var version = '1.0';
        var displayName = 'My Important Database';
        var maxSize = 65536; // in bytes
        var myDB = openDatabase(shortName, version, displayName, maxSize);

        // You should have a database instance in myDB.

    }
} catch(e) {
    // Error handling code goes here.
    if (e == INVALID_STATE_ERR) {
        // Version number mismatch.
    alert("Invalid database version.");
    } else {
    alert("Unknown error "+e+".");
    }
    return;
}

// alert("Database is: "+myDB);
```

```
createTables(myDB);
systemDB = myDB;

}



/*! Format a link to a document for display in the "Choose a file" pane. */
function docLink(row)
{
    var name = row['name'];
    var files_id = row['id'];

    return "<tr class='filerow'><td class='filenamecell'>"+name+"</td><td
class='filelinkcell'>(<a href='#' onClick=loadFile("+files_id+")>edit</a>) (<a
 href='#' onClick=deleteFile("+files_id+")>delete</a>)</td></tr>\n";
}


/*! If a deletion resulted in a change in the list of files, redraw the "Choose a
 file" pane. */
function deleteUpdateResults(transaction, results)
{
    if (results.rowsAffected) {
        chooseDialog();
    }
}


/*! Mark a file as "deleted". */
function reallyDelete(id)
{
    // alert('delete ID: '+id);
    var myDB = systemDB;

    myDB.transaction(
        new Function("transaction", "transaction.executeSql('UPDATE files set
deleted=1 where id=?;', [ "+id+" ], /* array of values for the ? placeholders */"+
            "deleteUpdateResults, errorHandler);")
```

```
    );

}

/*! Ask for user confirmation before deleting a file. */
function deleteFile(id)
{
    var myDB = systemDB;

    myDB.transaction(
        new Function("transaction", "transaction.executeSql('SELECT id,name from
files where id=?;', [ "+id+" ], /* array of values for the ? placeholders */"+
            "function (transaction, results) {"+
                "if (confirm('Really delete '+results.rows.item(0)['name']+'?'))
{"+
                    "reallyDelete(results.rows.item(0)['id']);"+
                "}"+
            "}, errorHandler);")
    );
}

/*! This prints a list of "files" to edit. */
function chooseDialog()
{
    var myDB = systemDB;

    myDB.transaction(
        function (transaction) {
        transaction.executeSql("SELECT * from files where deleted=0;",
            [ ], // array of values for the ? placeholders
            function (transaction, results) {
                var string = '';
                var controldiv = document.getElementById('controldiv');
                for (var i=0; i<results.rows.length; i++) {
                    var row = results.rows.item(i);
                    string = string + docLink(row);
```

```
                }
                if (string == "") {
                    string = "No files.<br />\n";
                } else {
                    string = "<table class='filetable'>"+string+"</table>";
                }
                controldiv.innerHTML="<H1>Choose a file to
edit</H1>"+string+linkToCreateNewFile();
            }, errorHandler);
        }
    );

}

/*! This prints a link to the "Create file" pane. */
function linkToCreateNewFile()
{
    return "<p><button onClick='createNewFile()'>Create New File</button>";
}



/*! This creates a new "file" in the database. */
function createNewFileAction()
{
    var myDB = systemDB;
    var name = document.getElementById('createFilename').value

    // alert('Name is "'+name+'"');

    myDB.transaction(
        function (transaction) {
            var myfunc = new Function("transaction", "results", "/* alert('insert
 ID is'+results.insertId); */ transaction.executeSql('INSERT INTO files (name,
filedata_id) VALUES (?, ?);', [ '"+name+"', results.insertId], nullDataHandler,
killTransaction);");
```

```
                transaction.executeSql('INSERT INTO filedata (datablob) VALUES
("");', [],
                myfunc, errorHandler);
        }
    );

    chooseDialog();
}

/*! This saves the contents of the file. */
function saveFile()
{
    var myDB = systemDB;
    // alert("Save not implemented.\n");

    var contentdiv = document.getElementById('contentdiv');
    var contents = contentdiv.contentDocument.body.innerHTML;

    // alert('file text is '+contents);

    myDB.transaction(
        function (transaction) {
            var contentdiv = document.getElementById('contentdiv');
            var datadiv = document.getElementById('tempdata');

            var filedata_id = datadiv.getAttribute('lfdataid');
            var contents = contentdiv.contentDocument.body.innerHTML;

            transaction.executeSql("UPDATE filedata set datablob=? where id=?;",
              [ contents, filedata_id ], // array of values for the ? placeholders
                nullDataHandler, errorHandler);
            // alert('Saved contents to '+filedata_id+': '+contents);
            var origcontentdiv = document.getElementById('origcontentdiv');
            origcontentdiv.innerHTML = contents;
```

```
            alert('Saved.');
            }
    );
}

/*! This displays the "Create file" pane. */
function createNewFile()
{
    var myDB = systemDB;
    var controldiv = document.getElementById('controldiv');
    var string = "";

    string += "<H1>Create New File</H1>\n";
    string += "<form action='javascript:createNewFileAction()'>\n";
    string += "<input id='createFilename' name='name'>Filename</input>\n";
    string += "<input type='submit' value='submit' />\n";
    string += "</form>\n";

    controldiv.innerHTML=string;

}

/*! This processes the data read from the database by loadFile and sets up the
editing environment. */
function loadFileData(transaction, results)
{
    var controldiv = document.getElementById('controldiv');
    var contentdiv = document.getElementById('contentdiv');
    var origcontentdiv = document.getElementById('origcontentdiv');
    var datadiv = document.getElementById('tempdata');

    // alert('loadFileData called.');

    var data = results.rows.item(0);
    var filename = data['name'];
```

```
    var filedata = data['datablob'];
    datadiv.setAttribute('lfdataid', parseInt(data['filedata_id']));

    document.title="Editing "+filename;
    controldiv.innerHTML="";
    contentdiv.contentDocument.body.innerHTML=filedata;
    origcontentdiv.innerHTML=filedata;
    contentdiv.style.border="1px solid #000000";
    contentdiv.style['min-height']='20px';
    contentdiv.style.display='block';
    contentdiv.contentDocument.contentEditable=true;
}

/*! This loads a "file" from the database and calls loadFileData with the results.
 */
function loadFile(id)
{
    // alert('Loading file with id '+id);
    var datadiv = document.getElementById('tempdata');
    datadiv.setAttribute('lfid', parseInt(id));

    myDB = systemDB;
    myDB.transaction(
        function (transaction) {
            var datadiv = document.getElementById('tempdata');
            var id = datadiv.getAttribute('lfid');
            // alert('loading id' +id);
           transaction.executeSql('SELECT * from files, filedata where files.id=?
 and files.filedata_id = filedata.id;', [id ], loadFileData, errorHandler);
        }
    );

}

/*! This creates the database tables. */
function createTables(db)
```

```
{


/* To wipe out the table (if you are still experimenting with schemas,
   for example), enable this block. */
if (0) {
    db.transaction(
        function (transaction) {
        transaction.executeSql('DROP TABLE files;');
        transaction.executeSql('DROP TABLE filedata;');
        }
    );
}


db.transaction(
    function (transaction) {
        transaction.executeSql('CREATE TABLE IF NOT EXISTS files(id INTEGER NOT
NULL PRIMARY KEY AUTOINCREMENT, name TEXT NOT NULL, filedata_id INTEGER NOT NULL,
 deleted INTEGER NOT NULL DEFAULT 0);', [], nullDataHandler, killTransaction);
       transaction.executeSql('CREATE TABLE IF NOT EXISTS filedata(id INTEGER NOT
 NULL PRIMARY KEY AUTOINCREMENT, datablob BLOB NOT NULL DEFAULT "");', [],
nullDataHandler, errorHandler);
    }
);


}


/*! When passed as the error handler, this silently causes a transaction to fail.
 */
function killTransaction(transaction, error)
{
    return true; // fatal transaction error
}


/*! When passed as the error handler, this causes a transaction to fail with a
warning message. */
function errorHandler(transaction, error)
```

```
{
    // error.message is a human-readable string.
    // error.code is a numeric error code
    alert('Oops.  Error was '+error.message+' (Code '+error.code+')');

    // Handle errors here
    var we_think_this_error_is_fatal = true;
    if (we_think_this_error_is_fatal) return true;
    return false;
}

/*! This is used as a data handler for a request that should return no data. */
function nullDataHandler(transaction, results)
{
}

/*! This returns a string if you have not yet saved changes.  This is used by the
 onbeforeunload
    handler to warn you if you are about to leave the page with unsaved changes.
*/
function saveChangesDialog(event)
{
    var contentdiv = document.getElementById('contentdiv');
    var contents = contentdiv.contentDocument.body.innerHTML;
    var origcontentdiv = document.getElementById('origcontentdiv');
    var origcontents = origcontentdiv.innerHTML;

    // alert('close dialog');

    if (contents == origcontents) {
    return NULL;
    }

    return "You have unsaved changes."; //   CMP "+contents+" TO "+origcontents;
}
```

```
/*! This sets up an onbeforeunload handler to avoid accidentally navigating away
from the
    page without saving changes. */
function setupEventListeners()
{
    window.onbeforeunload = function () {
    return saveChangesDialog();
    };
}
```

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010