
Safari Client-Side Storage and Offline Applications Programming Guide

Data Management



2009-06-24



Apple Inc.
© 2009 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, and Safari are trademarks of Apple Inc., registered in the United States and other countries.

iPhone is a trademark of Apple Inc.

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS

PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction 7**

Organization of This Document 7
See Also 7

Chapter 1 **HTML 5 Offline Application Cache 9**

Creating a Manifest File 9
Declaring a Manifest File 10
Updating the Cache 11
Handling Cache Events 11

Chapter 2 **Key-Value Storage 13**

Using Session and Local Storage 13
 Storing and Retrieving Values 14
 Deleting Values 15
Handling Storage Events 15
A Complete Example 16
Concurrency Considerations 18

Chapter 3 **Relational Database Basics 19**

Relationship Models and Schema 20
SQL Basics 22
 CREATE TABLE Query 22
 INSERT Query 24
 SELECT Query 24
 UPDATE Query 25
 DELETE Query 26
 DROP TABLE Query 26
Transaction Processing 26
Relational Databases and Object-Oriented Programming 27
SQL Security and Quoting Characters in Strings 27

Chapter 4 **Using the JavaScript Database 29**

Creating and Opening a Database 29
Creating Tables 30
Executing a Query 31
Handling Result Data 31
Per-Query Error Callbacks 33

Error Codes 34
Transaction Callbacks 34
Working With Database Versions 35

Appendix A **Database Example: A Simple Text Editor 39**

Adding a Save Button to FancyToolbar.js 39
Creating the index.html File 40
Creating the SQLStore.js File 41

Document Revision History 49

Tables and Listings

Chapter 1 **HTML 5 Offline Application Cache 9**

Listing 1-1 Sample manifest file 10

Chapter 3 **Relational Database Basics 19**

Table 3-1 Relational database “family” table 19
Table 3-2 Relational database “familymember” table 20
Table 3-3 The “classes” table 21
Table 3-4 The “student_class” table 21

Appendix A **Database Example: A Simple Text Editor 39**

Listing A-1 Additions to FancyToolbar.js 40
Listing A-2 index.html 40
Listing A-3 SQLStore.js 41

Introduction

The HTML 5 specification describes an offline application cache that allows you to write web applications that work correctly when your computer or web-enabled device is not connected to the Internet.

To support these offline web applications (and online web applications), the HTML 5 specification provides two new mechanisms for client-side data storage: Key/Value storage and JavaScript database support.

JavaScript database support is available in Safari 3.1 and later, and in iPhone OS 2.0 and later.

Key/value storage is available in Safari 4 and later.

Offline application support is available in iPhone OS 2.1 and later and in Safari 4.0 and later.

HTML 5 is currently in development by the Web Hypertext Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C).

You should read this documentation if you are a web developer who wants to store data locally on a user's computer in amounts beyond what can reasonably be stored in an HTTP cookie.

Organization of This Document

This documentation is organized into the following chapters:

- [“Key-Value Storage”](#) (page 13)—Describes the new Key/Value storage objects (`localStorage` and `sessionStorage`).
- [“Relational Database Basics”](#) (page 19)—Provides an overview of relational databases and the SQLite dialect of SQL.
- [“Using the JavaScript Database”](#) (page 29)—Tells how to use the JavaScript interface to SQLite.
- [“Database Example: A Simple Text Editor”](#) (page 39)—Provides an example of how to use SQLite in JavaScript.

See Also

For more information about the JavaScript database and key/value storage APIs (including reference material), see the [HTML5 web storage specification](#).

For more information about JavaScript in general, read *Apple JavaScript Coding Guidelines* and *Safari DOM Extensions Reference*.

HTML 5 Offline Application Cache

Beginning in iPhone OS 2.1 and Safari 4.0, Safari provides an offline application cache. This cache allows you to create web-based applications that work correctly even when the user's computer or web-enabled device is not connected to the Internet.

This technology consists of two parts: a manifest file and JavaScript interfaces.

The manifest file is a text file that contains a list of resources to be cached.

The JavaScript programming interfaces allow you to trigger an update of the cached files when desired.

Creating a Manifest File

The manifest file specifies the resources—such as HTML, JavaScript, CSS, and image files—to download and store in the application cache. After the first time a webpage is loaded, the resources specified in the manifest file are obtained from the application cache, not the web server.

The manifest file has the following attributes:

- It must be served with type `text/cache-manifest`.
- The first line must contain the text `CACHE MANIFEST`.
- Subsequent lines may contain URLs for each resource to cache or comments.
- Comments must be on a single line and preceded by the `#` character.
- The URLs are file paths to resources you want to download and cache locally. The file paths should be relative to the location of the manifest file—similar to file paths used in CSS—or absolute.
- The manifest must also declare a whitelist that contains a prefix match for *all* external resources that it needs to access when online. This whitelist begins with the identifier `NETWORK:` on a line by itself.

You can declare additional cache entries after network entries by inserting the identifier `CACHE:` on a line by itself, if desired.

- The manifest may, if desired, contain fallback sections. These sections begin with the identifier `FALLBACK:` and contain a series of lines containing two URIs. The second URI is a URI that the browser should use if the first URI is inaccessible. The URIs must both be from the same origin as the manifest itself. Both URIs may be partial URIs (prefixes).

For example:

```
FALLBACK:  
/files/projects /projects
```

In the above example, if a file cannot be found in `/files/projects`, the client tries again, replacing `/files/projects` with `/projects`.

- The HTML file that declares the manifest file, described in “[Declaring a Manifest File](#)” (page 10), is automatically included in the application cache. You do not need to add it to the manifest file.

For example, “[Creating the index.html File](#)” shows a manifest file that contains URLs to some image resources.

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/ images/
```

Declaring a Manifest File

After you create a manifest file you need to declare it in the HTML file. You do this by adding a `manifest` attribute to the `<html>` tag as follows:

```
<html manifest="demo.manifest">
```

The argument to the `manifest` attribute is a relative or absolute path to the manifest file.

Important: In some browsers, the application cache does not work fully if you use a non-HTML 5 DOCTYPE declaration. Instead, use an HTML 5 DOCTYPE declaration:

```
<!DOCTYPE html>
```

In most cases, creating a manifest file and declaring it is all you need to do to create an application cache. After doing this, the resources are automatically stored in the cache the first time the webpage is displayed and loaded from the cache by multiple browser sessions thereafter. Read the following sections if you want to manipulate this cache from JavaScript.

Updating the Cache

You can wait for the application cache to update automatically or manually trigger an update using JavaScript. The application cache automatically updates only if the manifest file changes. It does not automatically update if resources listed in the manifest file change.

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

Note that errors can also occur when updating the application cache. If downloading the manifest file or a resource specified in the manifest file fails, the entire update process fails. If the update process fails, the current application cache is not corrupted—the browser continues to use the previous version of the application cache. If the update is successful, webpages begin using the new cache when they reload.

Use the following JavaScript class to trigger an update to the application cache and check its status. There is one application cache per document represented by an instance of the `DOMApplicationCache` class. The application cache is a property of the `DOMWindow` object.

For example, you get the `DOMApplicationCache` object as follows:

```
cache = window.applicationCache;
```

You can check the status of the application cache as follows:

```
if (window.applicationCache.status == window.applicationCache.UPDATEREADY)...
```

If the application cache is in the `UPDATEREADY` state, then you can update it by sending it the `update()` message as follows:

```
window.applicationCache.update();
```

If the update is successful, swap the old and new caches as follows:

```
window.applicationCache.swapCache();
```

The cache is ready to use when it returns to the `UPDATEREADY` state. See the documentation for `DOMApplicationCache` for other status values. Again, only webpages loaded after an update use the new cache, not webpages that are currently displayed by the browser.

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

Handling Cache Events

You can also listen for application cache events using JavaScript. Events are sent when the status of the application cache changes or the update process fails. You can register for these events and take the appropriate action.

For example, register for the `updateready` event to be notified when the application cache is ready to be updated. Also, register for the `error` event to take some action if the update process fails—for example, log an error message using the console.

```
cache = window.applicationCache;
cache.addEventListener('updateready', cacheUpdatereadyListener, false);
cache.addEventListener('error', cacheErrorListener, false);
```

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

Key-Value Storage

Beginning in Safari 4.0, Safari supports the HTML 5 client-side storage specification. One portion of that specification is key-value storage. The purposes of key-value storage are twofold:

- To provide storage for cookie-like data specific to a single browser window.
- To provide local storage that does not get retransmitted with every request to the server, thus making it practical to store larger quantities of data than are practical with cookies.

To support those goals, the HTML 5 specification provides two key-value storage objects: `sessionStorage` and `localStorage`.

The `localStorage` and `sessionStorage` JavaScript objects are functionally identical except in their persistence and scope rules:

- `localStorage`—used for long-term storage. This data persists after the window is closed and is shared across all browser windows.
- `sessionStorage`—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.

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 future changes made by either page are not reflected in the session storage for the other page.

Except for these differences, the `localStorage` and `sessionStorage` objects behave identically.

Like cookies, these objects provide different sets of data to pages served by different domains. Unlike cookies, the data is not transmitted back to the server during normal browsing. If you need to send the stored data back to the server, you must 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 store the same key in both storage objects.

Using Session and Local Storage

This section describes how to store, retrieve, and delete key-value pairs using the session storage or local storage objects. Because session storage and local storage objects behave similarly, the examples in this section all use session storage for consistency.

You can change these examples to use local storage by substituting the `localStorage` object in place of the `sessionStorage` object wherever it appears in the code.

Storing and Retrieving Values

To store a value in key-value storage, use the `setItem` method. This method takes two parameters: a name and a value.

Consider a website that holds regular contests with various prizes. You might elect to store the user's shirt size so that every time the user enters a contest that requests a shirt size, your JavaScript code can populate the shirt size field automatically with the last value chosen. You can store the value with the following line of code:

```
// Store the value of the variable "myShirtSize" into the
// session storage field named "shirt_size".
sessionStorage.setItem("shirt_size", myShirtSize);
```

If you want to store the value in local storage, simply use the `localStorage` object instead. For example:

```
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('Quota exceeded.');
```

In your `onload` method, you might then retrieve that value and populate the form field as appropriate with code like this:

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

// do something with the value here.
document.getElementById('shirt_size').value = myShirtSize;
```

If no key-value pair with the specified name exists, `myShirtSize` is set to `null`.

If your key name is a valid JavaScript token (no spaces, punctuation other than underscore, and so on), you can also retrieve the value like this:

```
var myShirtSize = sessionStorage.shirt_size;
```

You can find out the total number of keys (for your domain) in session storage by examining the value stored in `sessionStorage.length`. For example:

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

Finally, you can obtain a key based on a numeric index.

```
var value;
var index = 3;
try {
    value = sessionStorage(index);
} catch(e) {
    if (e == INDEX_SIZE_ERR) {
```

```
        alert('There are fewer than '+(index-1)+' keys in the session storage
object.');
```

This method throws an exception if the index is not within the range `[0..n-1]` where `n` is the number of keys in the storage object.

Deleting Values

There are two ways to delete values from key-value storage: individually or en masse. To delete a single value (continuing the shirt size example from the previous section), use the following line of code:

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

To remove all key-value pairs for your domain, use the following line of code:

```
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 also share the same session storage object because they descend from the same window.

Because this 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 so that your scripts can notice changes to local or session storage by other scripts and act on them.

Before you can receive storage events, you must first register for them by executing the following line of code:

```
document.body.addEventListener('storage', storage_handler, false);
```

Once you have registered an event handler, the specified function (in this case, `storage_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 storage_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 Complete Example

This HTML page demonstrates local and session storage. If you modify the values in either field, they are stored locally (both on modification and on page exit) and are retrieved when you reload the page. If you close the window and then load the page again, only the value stored in local storage is restored.

```
<html>
<head>
<script language="JavaScript" type="text/javascript"><!--

function restoreValues()
{
    if (typeof(sessionStorage) == 'undefined' || typeof(localStorage) ==
'undefined') {
        alert('local and session storage not supported by this browser.');
```

```

}

function setValue(value)
{
    if (value == 'myfield1') {
        sessionStorage.setItem(value, document.getElementById(value).value);
    } else {
        localStorage.setItem(value, document.getElementById(value).value);
    }
}

function saveChanges()
{
    setValue('myfield1');
    setValue('myfield2');

    return NULL;
}

function clearValue(value)
{
    if (value == 'myfield1') {
        sessionStorage.removeItem(value);
    } else {
        localStorage.removeItem(value);
    }
    document.getElementById(value).value = '';
}

--></script>

</head>
<body onload='restoreValues()''>

<p>Field 1 is stored in session storage, field 2 in local storage. Both should
be retrieved on a page reload, but field 1 should not be retrieved if you close
the window and reopen it.</p>

<p>
<input id='myfield1' onchange='setValue("myfield1")''>Field 1</input><br />
<input id='myfield2' onchange='setValue("myfield2")''>Field 2</input>
</p>

<p>
<a href='#' onclick='clearValue("myfield1")''>clear value 1</a><br />
<a href='#' onclick='clearValue("myfield2")''>clear value 2</a><br />
<a href='#' onclick='clearAll()''>clear all</a><br />
</p>

</body>
</html>

```

Concurrency Considerations

Key-value storage is no more concurrency-safe than cookies. Most JavaScript code tends to perform work in response to user actions, which means there is little actual concurrency involved. However, if your code needs to do regularly read and write to session storage objects for other purposes, you must do a bit of extra work.

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 be 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 21).

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 [TEMPORARY] 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 20):

```
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 20), 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 27).

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 19) and [Table 3-2](#) (page 20) in “[Relationship Models and Schema](#)” (page 20), 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 31).

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:

```
INSERT INTO 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 31).

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 iPhone OS 2.0, Safari supports the HTML5 JavaScript database class. The JavaScript database class, based on SQLite, provides a very basic relational database intended for local storage of content that is too large to conveniently store in cookies (or is too important to accidentally delete 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.

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:

```
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 == 2) {
    // Version number mismatch.
    alert("Invalid database version.");
  } else {
    alert("Unknown error "+e+".");
  }
  return;
}

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

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 35).

The short name is the name for your database as stored on disk. 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 19).

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

```
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 33).

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:

```

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 34).)

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 31).

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”:

```

function errorHandler(transaction, error)
{
    // Error is a human-readable string.
    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.

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:

```

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);
            }, 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:

```
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);
    }
}
```

Per-Query Error Callbacks

The 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 list of possible error codes that can appear in the `error.code` field, see “Error Codes” (page 34).

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.

Transaction Callbacks

In addition to handling errors on a per-query basis (as described in “Per-Query Error Callbacks” (page 33)), you can also check for success or failure of the entire transaction in a similar way. For example:

```
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 34).

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 34) 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:

```
var mydb = openDatabase(shortName, "", displayName, maxSize);
var version = mydb.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:

```
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.');
```

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.

You can generally work around this by calling `openDatabase` in your success handler, checking the database version of the resulting database handle, and calling `window.location.reload` to trigger a page reload.

```

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

    var mydb = openDatabase('mydatabase', '', 'My Important Database',
65536);

    var fail = 0;
    if (!mydb) {
        alert('could not reconnect to database');
        fail = 1;
    } else if (mydb.version != "2.0") {
```

```
        fail = 1;
    }
    if (fail) {
        alert('reloading page');
        window.location.reload();
    }
}
```


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, do 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 39).
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 40) and [“Creating the SQLStore.js File”](#) (page 41).

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 FancyToolbarIjs

```

    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;
```

Database Example: A Simple Text Editor

```

/*! 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='filenameecell'>"+name+"</td><td
class='filelinkcell'><a href='#' onClick=loadFile(\"+files_id+\")>edit</a>&nbsp;<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();
    }
}

```

Database Example: A Simple Text Editor

```

/!* 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);
    );
}

```

Database Example: A Simple Text Editor

```

/*! 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.');
```

Database Example: A Simple Text Editor

```

    }
  );
}

/*! 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');
```

Database Example: A Simple Text Editor

```

        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 is a human-readable string.
    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();
  };
}
```


Document Revision History

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

Date	Notes
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

REVISION HISTORY

Document Revision History