Safari JavaScript Database Programming Guide

Internet & Web > Scripting & Automation

2009-01-06

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

Simultaneously published in the United States and Canada.

Contents

Tables and Listings

Introduction

The HTML 5 specification provides a new mechanism for client-side data storage: JavaScript database support. HTML 5 is currently in development by the Web Hypertext Application Technology Working Group (WHATWG).

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

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:

- "Relational Database Basics" (page 9)—Provides an overview of relational databases and the SQLite dialect of SQL.
- "Using the JavaScript Database" (page 19)—Tells how to use the JavaScript interface to SQLite.
- "Database Example: A Simple Text Editor" (page 23)—Provides an example of how to use SQLite in JavaScript.

See Also

For more information about the JavaScript database support (including reference material), see the SQL section of the HTML 5 Working Draft.

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

Introduction

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 1-1 and Table 1-2.

Table 1-1 Relational database "family" table

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

Table 1-2 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 1-3.

  Table 1-3 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 1-4 (page 11).

  Table 1-4 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 1-1 in "Relationship Models and Schema" (page 10):

```
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 UFT-8 data cannot exceed 451 characters in length:

```
...
    Douglas NVARCHAR(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.

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 1-1 in "Relationship Models and Schema" (page 10), 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 17).

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 1-1 (page 9) and Table 1-2 (page 10) in "Relationship Models and Schema" (page 10), 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 21).

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 unforseen 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 21).

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

The JavaScript database, based on SQLite, is a very basic relational database intended to provide local storage for 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).

In addition, by providing a relational database model, the JavaScript database makes it easy to work with complex, interconnected data in a webpage. You might use it as an alternative to storing user data on the server, 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 == INVALID_STATE_ERR) {
        // Version number mismatch.
        alert("Invalid database version.");
    } else {
        alert("Unknown error "+e+".");
    }
    return;
}

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

The version upgrade feature is not yet supported, so for now you should just set the version number field to 1.0 and ignore it.

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. This prevents a runaway web application from using excessive local resources. 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 9).

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 Data and Error Callbacks" (page 21).

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 22).)

Per-Query Data and Error Callbacks

The examples in the previous section did not return any data. For a query that actually returns data, you have to provide a callback function to handle the data.

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++) {
        var row = results.rows.item(i);
        string = string + row['name'] + '\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.

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.

Transaction Callbacks

In addition to handling errors and data on a per-query basis (as described in "Per-Query Data and Error Callbacks" (page 21)), you can also check for success or failure of the entire transaction in the same 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.

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 23).
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 24) and "Creating the SQLStore.js File" (page 25).

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 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 JavaScript Database Programming Guide*.

Date	Notes
2009-01-06	Expanded SQL syntax coverage. Corrected database example instructions.
2008-03-18	TBD

Document Revision History