Using the HTML5 IndexedDB API

Local data persistence improves web app accessibility and mobile app responsiveness

Brian J Stewart
Principal Consultant
Aqua Data Technologies, Inc.

11 December 2012

The Indexed Database (IndexedDB) API, part of HTML5, is useful for creating rich, data-intensive, offline HTML5 web applications that need to locally store data. It's also useful for locally caching data to make traditional online web applications such as mobile web applications faster and more responsive. This article demonstrates how to manage IndexedDB databases.

A key feature of HTML5 is local data persistence, which enables web applications to be accessible whether online or offline. In addition, local data persistence enables mobile applications to be more responsive, consume less bandwidth, and work more efficiently in low-bandwidth scenarios. HTML5 offers a few options for local data persistence. The first option is `localstorage`, which lets you store data using a simple key-value pair. IndexedDB, a more powerful option, lets you locally store large numbers of objects and retrieve data using robust data access mechanisms.

The IndexedDB API replaces the Web Storage API, which was deprecated in the HTML5 specification. (Several leading browsers still support Web Storage, however, including Apple's Safari and the Opera web browser.) IndexedDB has several advantages over Web Storage, including indexing, transactions, and robust querying capabilities. Through a series of examples, this article shows how to manage IndexedDB databases. (See the Download section to get the full source code for the examples.)

Key concepts

A website can have one or more IndexedDB databases. Each database must have a unique name.

A database can contain one or more *object stores*. An object store, which is also uniquely identified by a name, is a collection of records. Each record has a *key* and a *value*. The value is an object that can have one or more properties or attributes. The key can be based on a key

generator, derived by a key path, or explicitly set. A key generator automatically generates a unique sequential positive integer. A key path defines the path to the key value. It can be a single JavaScript identifier or multiple identifiers separated by periods.

The specification includes both an asynchronous and a synchronous API. The synchronous API is meant to be used within web workers. The asynchronous API uses requests and callbacks.

In the following examples, the output is appended to a `div` tag with an ID of `result`. The `result` element is updated by clearing and setting the `innerHTML` property during each data operation. Each of the example JavaScript functions is invoked by an `onclick` event of HTML buttons.

Handling errors or exceptions and debugging

All asynchronous requests have an `onsuccess` callback that's invoked when a database operation is successful and an `onerror` callback that's invoked when an operation is unsuccessful. Listing 1 is an example of an `onerror` callback.

Listing 1. Asynchronous error handler

```
request.onerror = function(e) {
   // handle error
   ...
   console.log("Database error: " + e.target.errorCode);
};
```

When you're working with the IndexedDB API, using the JavaScript `try/catch` blocks is a good idea. This functionality is useful for handling errors and exceptions that might occur before the database operation, such as attempting to read or manipulate data when the database is not open or trying to write data while another read/write transaction is already open.

IndexedDB can be difficult to debug and troubleshoot, because in many cases the error messages are generic and uninformative. When developing applications, take advantage of `console.log` and JavaScript debugging tools such as Firebug for Mozilla Firefox or Chrome's built-in Developer Tools. These tools are invaluable for any JavaScript-intensive applications, but particularly for HTML5 applications that use the IndexedDB API.

Working with databases

Only a single version of a database can exist at a time. When a database is first created, its initial version is zero. After it's created, a database (and its object stores) can only be changed through a specialized type of transaction known as a `versionchange` transaction. To change a database after its creation, you must open the database with a higher version number. This action causes the `upgradeneeded` event to fire. Code to modify the database or object stores must reside in the `upgradeneeded` event handler.

The code fragment in Listing 2 shows how to create a database: You call the `open` method and pass the database name. The database is created if a database with the specified name does not exist.

Listing 2. Creating a new database

```
function createDatabase() {
   var openRequest = localDatabase.indexedDB.open(dbName);

   openRequest.onerror = function(e) {
      console.log("Database error: " + e.target.errorCode);
   };
   openRequest.onsuccess = function(event) {
      console.log("Database created");
      localDatabase.db = openRequest.result;
   };
   openRequest.onupgradeneeded = function (evt) {
         ...
   };
}
```

To delete an existing database, call the `deleteDatabase` method and pass the name of the database to be deleted, as shown in Listing 3.

Listing 3. Deleting an existing database

```
function deleteDatabase() {
   var deleteDbRequest = localDatabase.indexedDB.deleteDatabase(dbName);
   deleteDbRequest.onsuccess = function (event) {
      // database deleted successfully
   };
   deleteDbRequest.onerror = function (e) {
      console.log("Database error: " + e.target.errorCode);
   };
}
```

The code fragment in Listing 4 shows how to open a connection to an existing database.

Listing 4. Opening current version of a database

```
function openDatabase() {
   var openRequest = localDatabase.indexedDB.open("dbName");
   openRequest.onerror = function(e) {
      console.log("Database error: " + e.target.errorCode);
   };
   openRequest.onsuccess = function(event) {

      localDatabase.db = openRequest.result;
   };
}
```

That's all there is to creating, deleting, and opening a database. Now it's time to move on to working with object stores.

Working with object stores

An object store is a collection of data records. To create a new object store in an existing database, you need to version the existing database. You do so by opening the database for versioning. In addition to the database name, the `open` method also accepts the version number as a second parameter. If you want to create a new version of a database (that is, create or modify an

object store), just open the database with a higher version than the existing database version. This invokes the `onupgradeneeded` event handler.

To create an object store, call the `createObjectStore` method on the database object, as shown in Listing 5.

Listing 5. Creating the object stores

```
function createObjectStore() {
   var openRequest = localDatabase.indexedDB.open(dbName, 2);
   openRequest.onerror = function(e) {
      console.log("Database error: " + e.target.errorCode);
   };
   openRequest.onsuccess = function(event) {
      localDatabase.db = openRequest.result;
   };
   openRequest.onupgradeneeded = function (evt) {
      var employeeStore = evt.currentTarget.result.createObjectStore
         ("employees", {keyPath: "id"});
   };
}
```

You've seen how object stores function. Next, let's take a look at how *indexes* reference the object store containing the data.

Using indexes

In addition to retrieving a record in an object store using its key, you can also retrieve records using indexed fields. Object stores can have one or more indexes. An index is a special object store that references the object store containing the data. It is automatically updated when the referenced object store changes (that is, when a record is added, modified, or deleted).

To create an index, you must version the database using the approach shown in Listing 5. Indexes can either be unique or non-unique. A unique index requires that all values within the index be unique, such as with an email address field. You use a non-unique index when a value can be repeated, such as city, state, or country. The code fragment in Listing 6 shows how to create a non-unique index on the state field, a non-unique index on the ZIP code field, and a unique index on the email address field in the employee object:

Listing 6. Creating an index

```
function createIndex() {
   var openRequest = localDatabase.indexedDB.open(dbName, 2);
   openRequest.onerror = function(e) {
      console.log("Database error: " + e.target.errorCode);
   };
   openRequest.onsuccess = function(event) {
      db = openRequest.result;
   };
   openRequest.onupgradeneeded = function (evt) {
     var employeeStore = evt.currentTarget.result.objectStore("employees");
     employeeStore.createIndex("stateIndex", "state", { unique: false });
     employeeStore.createIndex("emailIndex", "email", { unique: true });
     employeeStore.createIndex("zipCodeIndex", "zip_code", { unique: false })
   };
}
```

Next, you'll use transactions to perform operations to an object store.

Using transactions

You perform all reading and writing operations on an object store using transactions. Similarly to how transactions in relational databases work, IndexedDB transactions provide an atomic set of database write operations that are either entirely committed or not committed at all. IndexedDB transactions also have an abort and commit facility for database operations.

Table 1 lists and describes the modes IndexedDB provides for transactions.

Table 1. IndexedDB transaction modes

Mode	Description
`readonly`	Provides read-only access to an object store and is used when querying object stores.
`readwrite`	Provides read and write access to an object store.
`versionchange`	Provides read and write access to modify an object store definition or create a new object store.

The default transaction mode is `readonly`. You can have multiple concurrent `readonly` transactions, but only a single `readwrite` transaction can be open at any given time. Because of that, you consider using `readwrite` transactions only when data is updated. The singleton (meaning no other concurrent transactions can be open) `versionchange` transaction manipulates a database or object store. Use the `versionchange` transaction in the `onupgradeneeded` event handler to create, modify, or delete an object store or to add an index to an object store.

To create a transaction for the `employees` object store in `readwrite` mode, use the statement: `var transaction = db.transaction("employees", "readwrite");`.

The JavaScript function in Listing 7 shows how to use a transaction to retrieve a specific employee record in the `employees` object store using the key.

Listing 7. Fetching a specific record using the key

```
function fetchEmployee() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";
   if (localDatabase != null && localDatabase.db != null) {
     var store = localDatabase.db.transaction("employees").objectStore("employees");
     store.get("E3").onsuccess = function(event) {
      var employee = event.target.result;
      if (employee == null) {
         result.value = "employee not found";
      }
      else {
         var jsonStr = JSON.stringify(employee);
         result.innerHTML = jsonStr;
      }
     };
   }
```

```
}
catch(e){
   console.log(e);
}
}
```

The JavaScript function in Listing 8 shows how to use a transaction to retrieve a specific employee record in the `employees` object store using the `emailIndex` index rather than object store key.

Listing 8. Fetching a specific record using an index

```
function fetchEmployeeByEmail() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
      var range = IDBKeyRange.only("john.adams@somedomain.com");

      var store = localDatabase.db.transaction("employees")
.objectStore("employees");

      var index = store.index("emailIndex");

      index.get(range).onsuccess = function(evt) {
         var employee = evt.target.result;
         var jsonStr = JSON.stringify(employee);
         result.innerHTML = jsonStr;
      };
   }
}
catch(e){
```

Listing 9 is an example of using a `readwrite` transaction to create a new employee record.

Listing 9. Creating a new employee record

```
function addEmployee() {
   try {
      var result = document.getElementById("result");
      result.innerHTML = "";

      var transaction = localDatabase.db.transaction("employees", "readwrite");
      var store = transaction.objectStore("employees");

      if (localDatabase != null && localDatabase.db != null) {
         var request = store.add({
            "id": "E5",
            "first_name" : "Jane",
            "last_name" : "Doh",
            "email" : "jane.doh@somedomain.com",
            "street" : "123 Pennsylvania Avenue",
            "city" : "Washington D.C.",
            "state" : "DC",
            "zip_code" : "20500",
         });
         request.onsuccess = function(e) {
           result.innerHTML = "Employee record was added successfully.";
         };

         request.onerror = function(e) {
            console.log(e.value);
            result.innerHTML = "Employee record was not added.";
```

```
         };
      }
   }
   catch(e){
      console.log(e);
   }
}
```

Listing 10 is an example of using a `readwrite` transaction to update an existing employee record. This example changes the email address of the employee whose record ID is `E3`.

Listing 10. Updating an existing employee record

```
function updateEmployee() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   var transaction = localDatabase.db.transaction("employees", "readwrite");
   var store = transaction.objectStore("employees");
   var jsonStr;
   var employee;

   if (localDatabase != null && localDatabase.db != null) {

      store.get("E3").onsuccess = function(event) {
         employee = event.target.result;
         // save old value
         jsonStr = "OLD: " + JSON.stringify(employee);
         result.innerHTML = jsonStr;

         // update record
         employee.email = "john.adams@anotherdomain.com";

         var request = store.put(employee);

         request.onsuccess = function(e) {
            console.log("Added Employee");
         };

         request.onerror = function(e) {
            console.log(e.value);
         };


         // fetch record again
         store.get("E3").onsuccess = function(event) {
            employee = event.target.result;
            jsonStr = "
NEW: " + JSON.stringify(employee);
            result.innerHTML = result.innerHTML  + jsonStr;
         }; // fetch employee again
      }; // fetch employee first time
   }
}
catch(e){
   console.log(e);
}
}
```

Listing 11 is an example of a `readwrite` transaction to clear or delete all records in an object store. Like other asynchronous transactions, the `clear` transaction invokes the `onsuccess` or `onerror` callback, depending on whether the object store is cleared.

Listing 11. Clear object store transaction

```
function clearAllEmployees() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
     var store = localDatabase.db.transaction("employees", "readwrite")
.objectStore("employees");

     store.clear().onsuccess = function(event) {
      result.innerHTML = "'Employees' object store cleared";
     };
   }
}
catch(e){
   console.log(e);
}
}
```

These examples demonstrate some common uses of transactions. Next you'll see how cursors work in Indexed DB.

Using cursors

Similarly to the way cursors function in relational databases, cursors in IndexedDB enable you to iterate over records in an object store. You can also iterate over records using one of the object store's indexes. Cursors in IndexedDB are bi-directional, so you can iterate forward and backward through the records, as well as skip duplicate records in a non-unique index. The `openCursor` method opens a cursor. It accepts two optional arguments, including range and direction.

Listing 12 opens a cursor for the `employees` object store and iterates through all employee records.

Listing 12. Iterating through all employee records

```
function fetchAllEmployees() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
      var store = localDatabase.db.transaction("employees")
.objectStore("employees");

      var request = store.openCursor();
      request.onsuccess = function(evt) {
         var cursor = evt.target.result;
         if (cursor) {
            var employee = cursor.value;
            var jsonStr = JSON.stringify(employee);
            result.innerHTML = result.innerHTML + "
" + jsonStr;
            cursor.continue();
         }
      };
   }
}
catch(e){
```

```
   console.log(e);
}
}
```

The next few examples use cursors with indexes. Table 2 lists and describes the range types or filters that the IndexedDB API provides when it opens a cursor with an index.

Table 2. Range types or filters the IndexedDB API provides when it opens a cursor with an index

Range type or filter	Description
`IDBKeyRange.bound`	Returns all records within the specified range. This range has a lower and an upper boundary. It also has two optional parameters, `lowerOpen` and `upperOpen`, which indicate whether records with the lower or upper boundary should be included in the range.
`IDBKeyRange.lowerBound`	Returns all records past the specified boundary value. This range has an optional parameter, `lowerOpen`, to indicate whether records with the lower boundary should be included in the range.
`IDBKeyRange.upperBound`	Returns all records before the specified boundary value. It also has the optional `upperOpen` parameter.
`IDBKeyRange.only`	Returns only records that match the specified value.

Listing 13 is a basic example of iterating through all employee records for a specific state. This query is the most common. It enables you to retrieve all records matching a specific criterion. This example uses the `stateIndex` and the `IDBKeyRange.only` range, which returns all records that match the specified value (in this case, `"New York"`).

Listing 13. Iterating through all employee records in New York

```
function fetchNewYorkEmployees() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
      var range = IDBKeyRange.only("New York");

      var store = localDatabase.db.transaction("employees")
.objectStore("employees");

      var index = store.index("stateIndex");

      index.openCursor(range).onsuccess = function(evt) {
         var cursor = evt.target.result;
         if (cursor) {
            var employee = cursor.value;
            var jsonStr = JSON.stringify(employee);
            result.innerHTML = result.innerHTML + "
" + jsonStr;
            cursor.continue();
         }
      };
   }
}
catch(e){
   console.log(e);
}
```

```
}
```

Listing 14 is an example using the `IDBKeyRange.lowerBound` range. It retrieves all employees with a ZIP code higher than 92000.

Listing 14. Using `IDBKeyRange.lowerBound`

```
function fetchEmployeeByZipCode1() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
     var store = localDatabase.db.transaction("employees").objectStore("employees");
     var index = store.index("zipIndex");

     var range = IDBKeyRange.lowerBound("92000");

     index.openCursor(range).onsuccess = function(evt) {
      var cursor = evt.target.result;
      if (cursor) {
         var employee = cursor.value;
         var jsonStr = JSON.stringify(employee);
         result.innerHTML = result.innerHTML + "
" + jsonStr;
         cursor.continue();
      }
     };
   }
}
catch(e){
   console.log(e);
}
}
```

Listing 15 is an example using the `IDBKeyRange.upperBound` range. It retrieves all employees with a ZIP code lower than 93000.

Listing 15. Using `IDBKeyRange.upperBound`

```
function fetchEmployeeByZipCode2() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
     var store = localDatabase.db.transaction("employees").objectStore("employees");
     var index = store.index("zipIndex");

     var range = IDBKeyRange.upperBound("93000");

     index.openCursor(range).onsuccess = function(evt) {
      var cursor = evt.target.result;
      if (cursor) {
         var employee = cursor.value;
         var jsonStr = JSON.stringify(employee);
         result.innerHTML = result.innerHTML + "
" + jsonStr;
         cursor.continue();
      }
     };
   }
```

```
}
catch(e){
   console.log(e);
}
}
```

Listing 16 is an example using the `IDBKeyRange.bound` range. It retrieves all employees with a ZIP code between 92000 and 92999 inclusive.

Listing 16. Using `IDBKeyRange.bound`

```
function fetchEmployeeByZipCode3() {
try {
   var result = document.getElementById("result");
   result.innerHTML = "";

   if (localDatabase != null && localDatabase.db != null) {
     var store = localDatabase.db.transaction("employees").objectStore("employees");
     var index = store.index("zipIndex");

     var range = IDBKeyRange.bound("92000", "92999", true, true);

     index.openCursor(range).onsuccess = function(evt) {
      var cursor = evt.target.result;
      if (cursor) {
         var employee = cursor.value;
         var jsonStr = JSON.stringify(employee);
         result.innerHTML = result.innerHTML + "
" + jsonStr;
         cursor.continue();
      }
     };
   }
}
catch(e){
   console.log(e);
}
}
```

These examples demonstrate how the functionality of cursors in IndexedDB is similar to that of cursors in relational databases. With IndexedDB cursors, you can iterate over records in an object store and over records using one of the object store's indexes. Cursors in IndexedDB are bi-directional, providing additional flexibility.

Conclusion

The IndexedDB API is robust; you can leverage it to create rich, data-intensive applications—particularly offline HTML5 web applications—that need to locally store data. You can also use the IndexedDB API to locally cache data to make traditional online web applications—particularly mobile web applications—faster and more responsive by eliminating the need to retrieve data from the web server each time. For example, you can cache data for selection lists in an IndexedDB database.

This article demonstrated how to manage IndexedDB databases, including creating a database, deleting a database, and establishing a connection to a database. It also demonstrated many of the more advanced capabilities of the IndexedDB API, including transactions, indexes, and

cursors. You can use these demonstrated concepts to build offline or mobile web applications that leverage the IndexedDB API.

Downloads

Description	Name	Size
Source code examples for article	IndexedDBSourceCode.zip	3KB

Resources

Learn

- "HTML5 fundamentals, Part 1: Getting your feet wet" (developerWorks, May 2011): Learn more about the changes in HTML5 in this four-part series.
- "HTML5 fundamentals, Part 2: Organizing inputs" (developerWorks, May 2011): Find out more about HTML5 form controls and the role of JavaScript and CSS3 in this four-part series.
- "HTML5 fundamentals, Part 3: The power of HTML5 APIs" (developerWorks, June 2011): The third installment in this four-part series introduces HTML5 APIs, using an example page to demonstrate functions.
- "HTML5 fundamentals, Part 4: The final touch" (developerWorks, July 2011): Learn more about the HTML5 Canvas element, including functions, in this four-part series.
- IndexedDB API specification: See the World Wide Web Consortium (W3C) documentation to learn more.
- IndexedDB: The Mozilla Developer Network provides more information about the IndexedDB API.
- HTML5: Learn more about it at w3schools.com.
- HTML5 fundamentals: Learn more about HTML5 from this Knowledge Path.
- developerWorks Web development zone: Find articles covering various web-based solutions. See the Web development technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks Live! briefings: Get up to speed quickly on IBM products and tools as well as IT industry trends.
- developerWorks on-demand demos: Watch demos ranging from product installation and setup for beginners, to advanced functionality for experienced developers.
- developerWorks on Twitter: Join today to follow developerWorks tweets.

Get products and technologies

- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

Discuss

- developerWorks community: Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
- Find other developerWorks members interested in web development.

About the author

Brian J Stewart

Brian J. Stewart is a principal consultant at Aqua Data Technologies, a company that he founded to focus on content management, XML technologies, and enterprise client/server and web systems. He architects and develops enterprise solutions based on the Java EE and Microsoft .NET platforms.