AnswerHub

Transform Knowledge Into Success

KNOWLEDGE MANAGEMENT BUILT USING MODERN WEB TECHNOLOGIES

Delivers a Great Experience in Any Browser on Any Device

YOU CAN LAUNCH OUR SANDBOX IN SECONDS

TRY IT NOW!

DZone Refcardz

CONTENTS INCLUDE:

HTML5: IndexedDB

By: Gerard Gallant

INTRODUCTION

IndexedDB is an exciting browser-based database technology capable of holding large amounts of structured data and conducting high-performance searches using indexes. IndexedDB functionality is available online, offline, and even from within HTML5 Web Workers!

Cookies

This Refcard is aimed at intermediate to advanced web developers, but should be useful for anyone who is starting to build more heavy-duty client-side apps.

Cookies have their uses: when you log into a website, for example, a special authentication cookie is used to tell the server that you've logged in and may now view a given page.

In general, however, cookies in modern websites and web applications are discouraged -- partly because of their size (typically 4KB per cookie), and partly because all cookie data is passed back and forth with every call to the server, even if the data isn't required for that request.

All of that extra data costs your users money: you're both increasing their data usage bills and reducing the number of requests the server can handle at once, which can slow down responses.

HTML5 Web Storage

Web storage has several advantages: most notably, perhaps, it is supported by nearly every modern browser, including Internet Explorer 8.

By default, web storage allows you 5-10MB of space to work with, and your data is stored locally on the device rather than passed back-and-forth with each request to the server. Web storage is useful for storing small amounts of key/value data and preserving functionality online and offline.

With web storage, both the keys and values are stored as strings.

Web storage is not ideal, however, if you want to store large amounts of structured data. That's where HTML5 IndexedDB databases come in.

HTML5 IndexedDB

IndexedDB works both online and offline, allowing for client-side storage of large amounts of structured data, in-order key retrieval, searches over the values stored, and the option to store multiple values per key.

With IndexedDB, all calls are asynchronous and all interactions happen within a transaction.

Note: The specification for IndexedDB includes a section on synchronous calls, originally intended for use within HTML5 Web Workers, but that section of the specification might be removed in the future because no browsers currently support synchronous calls.

Also, you may have run across a technology called HTML5 Web SQL and wonder why that wasn't included in this discussion. The HTML5 Web SQL technology has been deprecated by W3C and it's now highly recommended that developers avoid that technology because browser vendors are not encouraged to support it. IndexedDB is the recommended alternative.

Browser support

Browser support for IndexedDB has been increasing and, as of mid-2013, it's available in the following browsers:

Internet Explorer 10, Firefox, Chrome, Chrome for Android, Firefox for Android, and BlackBerry 10

The following website is updated regularly and can save considerable testing time when you need to determine if a desired feature is available in the browser(s) you intend to support: http://caniuse.com. This tool also indicates the version of the browser in which a given feature became available.

For extended support in browsers that don't support IndexedDB but do support the deprecated HTML5 Web SQL technology like Safari (the default browser on the Mac OS, iPhone, and iPad), there is a special JavaScript shim available that manipulates the Web SQL database in the background and exposes the IndexedDB API for your code to interact with. The shim can be found here: http://nparashuram.com/IndexedDBShim/

Test if the feature exists

Even if you plan to support only certain browsers, it's always best to test for feature support before trying to use that feature. You can test to see if the browser supports HTML5 IndexedDB like this:

```
if (window.indexedDB) { /* IndexedDB is supported */ }
```

Same-origin policy

When you store information on a user's device, you don't want other websites reading or manipulating the data you've stored. To avoid this, browsers enforce a same-origin policy for IndexedDB databases so that only the website that created the database can access it. The protocol, domain, and port of any site requesting access must match that of the originating site.

So if WebSiteA.com creates an IndexedDB database, then WebSiteA.com/ProductInfo/ could access the database since it belongs to the same domain.

Pages from WebSiteA.com, however, would not be able to access the database. This is because the domains are the same, but the protocol differs (https rather than http).

AnswerHub
Transform Knowledge Into Success
KNOWLEDGE MANAGEMENT BUILT USING MODERN WEB TECHNOLOGIES
delivers a great experience in any browser on any device
YOU CAN LAUNCH OUR SANDBOX IN SECONDS
TRY IT NOW!

A crafty website might try to get around the same-origin policy restriction by loading WebSiteB's page into a Frame or iFrame to try and get a hold of the data. To prevent this attack vector, if a page is loaded into a Frame or iFrame and belongs to a different domain than the main page, as a security precaution, the page within the Frame/iFrame will not be allowed to access its own database.

Storage limits

There are no restrictions on the size of the data stored within the database but, depending on the browser, there may be size restrictions on the database itself.

With the desktop version of Firefox, an IndexedDB can use up to 50MB of storage before the user is presented with a message to find out if they want to allow the website to use more storage. If the user allows for an increase in storage, the IndexedDB growth will then only be limited by the capacity of the user's hard drive.

The mobile version of Firefox follows the same approach, except that the threshold for a user prompt is 5MB rather than 50MB.

With Internet Explorer 10, a database can use up to 10MB of storage by default before the user is presented with a message asking if additional storage is permitted. There also appears to be a hard coded upper limit of 250MB (in testing, adjusting the Temporary Internet Files limit above 250MB had no effect even with a reboot).

In Internet Explorer, if the database has reached its maximum size (either the user has chosen not to allow additional storage to be used or the upper limit of 250MB has been reached), the browser will not throw an error if your code tries to add additional records. Instead, if you watch the disk performance, or the Indexed DB folder itself (%AppData%\Local\Microsoft\Internet Explorer\Indexed DB\Internet.edb), it appears that Internet Explorer still tries to process the information but the Internet.edb file doesn't grow in size.

USING INDEXEDDB

All IndexedDB functionality is accessed through the window.indexedDB object, which is an IDBFactory object containing an open and a deleteDatabase method (there is also a cmp method available for comparing two keys).

Every operation with IndexedDB is asynchronous: you make a request for something to happen and you're returned an IDBRequest object. You then attach to the IDBRequest object's onerror and onsuccess event handlers to act on the results when they become available.

The IDBFactory's open and deleteDatabase methods return an IDBOpenDBRequest object, which is derived from the IDBRequest object and provides two additional event handlers: onblocked and onupgradeneeded.

Opening a database

To open a database, or to create one if it doesn't already exist, we simply call the open method of the window.indexedDB object and specify the desired database name.

The open method accepts a version number for the second parameter and will default to 1 if the database doesn't exist (if a database doesn't exist, when open is called, the database gets created). If specified, the version number must be greater than 0.

If the request to open a database was successful, the event object passed into the onsuccess event handler will contain an IDBDatabase object which we can then use to manipulate the objects in the database.

The following is an example of how you request that a database be opened:

```
var g_db = null;
var dbRequest = window.indexedDB.open(“Example”);
dbRequest.onerror = function (evt) {
alert(“Database error: “ + evt.target.error.name);
}
dbRequest.onsuccess = function (evt) {
g_db = evt.target.result; // IDBDatabase object
}
```

Creating an Object Store

Each database can have one or more object stores which hold your records as key/value pairs. Each object store has a name and the name must be unique within the database.

The records within an object store are sorted in ascending order based on the keys and each key must be unique for that object store.

Each object store has an optional key generator and an optional key path. If you specify a key path, the object store is said to use in-line keys (the key is stored as part of the value). If you don't specify a key path, the object store is said to use out-of-line keys (the key is stored separately from the value that is being stored).

There are three ways that keys can be specified for an object store:

- A key generator which can generate a monotonically increasing number every time a key is needed
- Keys can be derived from a key path
- Keys can be explicitly specified when data is being added to the object store

Each record in an object store has a value which can be primitives (string, number, boolean, null, and undefined) as well as anything supported by the structured clone algorithm which includes things like objects, arrays, dates, and even files from the HTML5 File API.

When a database is first created, or when a version number is specified that is higher than the current database version, an onupgradeneeded event handler will be triggered as part of the database versionchange transaction. If the upgradeneeded event is triggered, the onsuccess event handler of the open database request, will only be triggered after the versionchange transaction has completed.

The onupgradeneeded event handler is where all database structure changes are performed, including the creation of object stores.
To create an object store, you need to call the createObjectStore method on the IDBDatabase object which can be found in the event's target.result property of the onupgradeneeded event handler.

The following example illustrates how to create an object store called "Employees" that will use a key path:

```
dbRequest.onupgradeneeded = function (evt) {
var db = evt.target.result;
var objectStore = db.createObjectStore(“Employees”, { keyPath:
“EmpId” });
}
```

Creating an index

Indexes allow you to look up of data within an object store using properties of the values stored rather than just using the keys.

An index is a special object store for looking up records in another object store (also known as a referenced object store). The records in an index are automatically adjusted when items are added, updated, or deleted in the referenced object store. Several indexes can reference the same object store.

Indexes can also be used to enforce some database rules. For example, an index contains a unique flag which, when set to true, can be used to prevent having two records with the same value in an object store (an error would be thrown if the result of an insert/update results in a duplicate value).

Indexes also contain a multiEntry flag which effects how an index behaves when it comes to arrays. When set to false, a single record whose key is an array is added to the index. When set to true, however, a record is added to the index for each array item allowing you to filter on the individual items of the array.
To create an index, you need to call the createIndex method on the IDBDatabase object passing in a name for the index as the first parameter, a keypath as the second parameter, and an optional third parameter object indicating if the index is to be unique or not or if multiEntry is enabled or not.

Since indexes modify the database structure, indexes must be created and modified within the onupgradeneeded event handler as in the following example:

```
dbRequest.onupgradeneeded = function (evt) {
// ...object stores created...
/* Create an index so we can search by employee names. Multiple
employees might have the same name so we don't want the index to
be unique */
objectStore.createIndex("EmpName", "EmpName", { unique : false
});

/* Create an index to search employees by company email address.
Since, typically, no two employees would share the same work
email address, we can put a restriction on the records to enforce
that each record have a unique email address */
objectStore.createIndex("EmpEmail", "EmpEmail", { unique : true
});
}
```

Starting a Transaction

All reading or modification of data within a database must happen within a transaction.

The scope of a transaction is determined when the transaction is created and is simply what object stores and indexes the transaction has access to. A database can have multiple active transactions at once and, with readonly transactions, any number of them can execute at the same time even if they overlap scopes.

Unlike transactions of other database technologies, with IndexedDB, the transactions are auto-committed if the call is successful (your code doesn't need to explicitly call a commit method). If an error is thrown, or the call is explicitly aborted by code, then the transaction automatically rolls back. A database can have multiple active transactions at once and, with readonly transactions, any number of them can execute at the same time even if they overlap scopes.

Be especially careful with readwrite transactions because, if the scope overlaps, transactions will not execute at the same time. They will instead be queued up which could slow down the responsiveness of your web application.

There are three transaction modes available:

- **readonly**
- **readwrite**
- **versionchange** - This transaction is automatic when a database is first created or when the version number is changed. We saw this transaction in action earlier with the onupgreadeneeded event handler which is triggered by this transaction.

To create a transaction we need to call the transaction method of our IDBDatabase object, which will return us an IDBTransaction object.

The transaction method accepts two parameters. The first parameter is an array of object store names that we want as part of the transaction. The second parameter is optional and indicates the transaction mode we want. If not specified, the second parameter defaults to readonly mode.

The following is an example of how you would create a readwrite transaction on the Employees object store:

```
var dbTrans = g_db.transaction(["Employees"], "readwrite");
```

Note: Remember transactions are expected to be short-lived.

If a transaction takes too long, the browser may terminate it to free up the storage resources that the transaction has locked.

Adding data to an object store

To add data to an object store, we could populate the object store during the onupgradeneeded event handler since we're already in a transaction. Because we would already have a reference to the object store, it's just a matter of calling the object store's add method as in the following example:

```
dbRequest.onupgradeneeded = function (evt) {
// ...object stores and indexes created...
// Add the employee array items to the object store
for (var i in arrEmployees) {
objectStore.add(arrEmployees[i]);
}
}
```

Adding records to an object store during the upgradeneeded event is fine when the database is being created or upgraded but what about when the database already exists and we don't need to update the version?

To add a record to an object store, we first need to obtain a readwrite transaction on the object store we wish to add data to by calling the transaction method on our IDBDatabase object.

Once we have the IDBTransaction object we can call that object's objectStore method requesting one of the object stores that we specified when we called the transaction method.

The objectStore method returns us an IDBObjectStore object which we can then use to call the add method to request our object be inserted as in the following example:

```
var objEmployee = { "EmpId": "101", "EmpName": "Sam Smith",
"EmpEmail": "SamSmith@SomeCompany.com" };
// Create a readwrite transaction for the Employees object store
var dbTrans = g_db.transaction(["Employees"], "readwrite");
// Get the IDBObjectStore object for the Employees object store
var dbObjectStore = dbTrans.objectStore("Employees");
// Request the addition of our object to the object store
var dbAddRequest = dbObjectStore.add(objEmployee);
dbAddRequest.onsuccess = function (evt) { alert("Success!"); }
dbAddRequest.onerror = function (evt) { /* handle the error */ }
```

There is also a put method that can be used. The put method will act like an add method if the record doesn't exist yet but, if the record does exist, the put method overwrites the existing record with the new data. The code works exactly the same as in the above example with the exception that you request a put rather than an add.

Both the add and put requests accept an optional second parameter which is how you would specify the key for the record if you're not using inline keys.

Querying data

Requesting a record from an object store is very similar to adding it to the object store: You create your transaction (a readonly transaction will do in this case), request the object store object, and then request the record as in the following example:

```
/* Since the 2nd parameter of the transaction method is optional
and defaults to "readonly" we've left it out to reduce code.
Also, since we don't need the IDBTransaction object after we call
the objectStore method we can chain the calls together and not
bother storing the transaction object to a variable */
var dbObjectStore = g_db.transaction(["Employees"]).
objectStore("Employees");
// A Get is based on the key. In our case, it's the EmpId value
var dbGetRequest = dbObjectStore.get("101");
dbGetRequest.onsuccess = function (evt) {
alert("Success! The employee's name: " + evt.target.result.
EmpName); }
dbGetRequest.onerror = function (evt) { /* handle the error */ }
```

Using a cursor

A get is useful if you only want one record and you know the key value of the record you're looking for.

If you wish to iterate over multiple records, perhaps to build up a list of items for display, you'll need to use a cursor.

Note: If you wish to loop over records in search of a specific record, cursors are not the recommended approach. See the "Using an Index" section which will discuss using cursors on indexes.

As with a get request, to use a cursor you must first create a transaction and then obtain the object store object which you can then call the openCursor method on.

The first parameter of openCursor method is an optional key range which is used to restrict the number of records returned (if not specified, all records from the object store are returned).

You can create a key range object by calling one of the following methods on the IDBKeyRange object: only, lowerBound, upperBound, and bound. The second parameter of openCursor is also optional and specifies the direction that you will iterate over the results: "next" is for ascending order which is the default and "prev" is for descending order.

If you want to specify the second parameter without having to specify the first one, you can pass in a null for the first parameter.

The following is an example of using a cursor to loop over an entire object store's records (no range filter and using the default direction):

```
var dbCursorRequest = dbObjectStore.openCursor();
dbCursorRequest.onsuccess = function (evt) {
var curCursor = evt.target.result;
if (curCursor) {
/* Grab the current employee object from the cursor's value
property (we could also grab the key) */
var objEmployee = curCursor.value;
// ...do something with objEmployee...

// Cause onsuccess to fire again with the next item
curCursor.continue();
} // End if
}
dbCursorRequest.onerror = function (evt) { /* handle the error */
}
```

A few notes about cursors:

- To move to the next record in the object store, you call the continue method of the cursor object which will cause the onsuccess event handler to fire again for the next record.
- If there are no more records in the object store, the cursor will be undefined. You will need to test for this condition before trying to access the key or value.

Using an index

If you ever need to look for records based on a value other than the key, it's much more efficient to use an index.

You can request a get on an index which will return you a single record. Even if there are multiple records for the search value, you will always get the record with the lowest key value.

The following is an example of doing a get request on an index:

```
var dbIndex = dbObjectStore.index("EmpName");
var dbGetRequest = dbIndex.get("Sam Smith");
dbGetRequest.onsuccess = function (evt) {
var objEmployee = evt.target.result;
/* do something with the result */
}
```

If you want to take advantage of the better performance of indexes but still need to loop over multiple records, you have an option of two different types of cursors.

The standard index cursor behaves the same as a normal cursor, where the value returned is the whole object from the object store. The following is an example of how you obtain a standard index cursor:

```
var dbCursorRequest = dbIndex.openCursor();
dbCursorRequest.onsuccess = function (evt) {
var curCursor = evt.target.result;
if (curCursor) {
/* Grab the current employee object from the cursor's value
property (we could also grab the key). Do something with the
object */
var objEmployee = curCursor.value;

// Cause onsuccess to fire again with the next item
curCursor.continue();
} // End if
}
```

The other type of cursor that's possible on an index is a key cursor where the cursor's key is the index value and the cursor's value is the object store's key. For example:

```
var dbCursorRequest = dbIndex.openKeyCursor();
dbCursorRequest.onsuccess = function (evt) {
var curCursor = evt.target.result;
if (curCursor) {
/* curCursor.key will hold a value like "Sam Smith"
curCursor.value will hold the record's key (e.g. "101" in our
case) */

// Cause onsuccess to fire again with the next item
curCursor.continue();
} // End if
}
```

With both types of index cursors you can specify a key range filter like you can with normal cursors.

You can also specify the direction like you can with normal cursors but, in this case, you can make use of two additional values on indexes when those indexes are not specified as unique: "nextunique" and "prevunique" (filters the returned records to just unique items)

Deleting data

To delete records from an object store there are a couple of options available.

The first option is to use a delete request on the object store requesting that a specific record be deleted (bear in mind that the transaction shown in the examples below need to be set to readwrite):

```
var dbDeleteRequest = dbObjectStore.delete("101");
dbDeleteRequest.onsuccess = function(evt){ /* record is gone */ }
dbDeleteRequest.onerror = function(evt){}
```

The other option is to use the clear request on the object store requesting that all records within the object store be deleted along with any index records that reference the object store:

```
var dbClearRequest = dbObjectStore.clear();
dbClearRequest.onsuccess = function(evt){ /* records are gone */
}
dbClearRequest.onerror = function(evt){}
```

Handling errors

Most documentation you will find online about the IDBRequest onerror event handlers will tell you that you need to access the errorCode value. This is no longer the case.

The event's target value now holds an error object, which is a DOMError object. Rather than errorCode you now have access to the name property of the error object as in the following example:

```
dbRequest.onerror = function (evt) {
alert("Error: " + evt.target.error.name);
}
```

Aside from the onerror handlers of the IDBRequest object, it's also very important to wrap your database code in a try/catch statement since several actions will throw exceptions even if you have set up error event handlers.

For example, you will receive a ConstraintError exception if you try to create the same index twice which gets by the onerror handler and shows up in the console window (if your user is using IE and has IE set up to show errors, the user will receive a prompt with the error).

The W3C specification lists some common exceptions, if you're interested in knowing what could trigger one: http://www.w3.org/TR/IndexedDB/#exceptions.

Closing a database connection

A database can be closed in several ways, including by the garbage collector or if the user navigates away from your page.

You can explicitly close a database connection by calling the close method on the IDBDatabase object as in the following example:

```
/* g_db is a global object holding the evt.target.result value
(our IDBDatabase object) from the onsuccess event handler when we
called window.indexedDB.open */
g_db.close();
```

ADVANCED USAGE

Upgrading an existing database to a new version

To upgrade an existing database to a new version we need to specify a version number that is higher than the current database version when we open the database which will trigger the onupgradeneeded event handler.

The parameter received by the event handler will hold two values oldVersion and newVersion which can help your upgrade code know what code needs to run in order to bring the database up to the current version.

```
var dbRequest = window.indexedDB.open(“Example”, 2);
dbRequest.onupgradeneeded = function (evt) {
if (evt.oldVersion < 1) {
// create objects for version 1
}
// other version upgrade paths
}
```

Deleting an object store

Deleting an object store is a modification to the database structure and needs to be handled within the onupgradeneeded event handler.

To delete an object store, it's simply a matter of calling the deleteObjectStore method on the IDBDatabase object, specifying the name of the object store that you want deleted, as in the following example:

```
dbRequest.onupgradeneeded = function (evt) {
var db = evt.target.result;
db.deleteObjectStore(“Employees”);
}
```

Deleting a database

If you ever need to delete a database, this can be accomplished by requesting a deleteDatabase on the IDBFactory object as in the following example:

```
var dbRequest = window.indexedDB.deleteDatabase(“Example”);
dbRequest.onsuccess = function(evt){ /* database is gone */ }
dbRequest.onerror = function(evt){}
```

IndexedDB from within Web Workers

One nice feature with IndexedDB is that it's available to you from within a web worker rather than just from the UI thread!

Since there is no global 'window' object in a web worker, the approach used to access the indexedDB object needs to be modified slightly to use the global self instead.

The following example shows how you would work with IndexedDB from within a worker:

```
// Use ‘self’ instead of ‘window’ when in a worker
var g_db = null;
var dbRequest = self.indexedDB.open(“Example”);
dbRequest.onerror = function (evt) {
// Tell the UI thread about the error
self.postMessage(“Database error: “ + evt.target.error);
}
dbRequest.onsuccess = function (evt) {
g_db = evt.target.result;
}
dbRequest.onupgradeneeded = function (evt) {
// Create an objectStore to hold Employee information
evt.target.result.createObjectStore(“Employees”, { keyPath:
“EmpId” });
}
```

Aborting a transaction

If for some reason you need to abort a transaction and rollback any changes, you can do so at any point before the transaction finishes, including if he transaction isn't currently active or hasn't started yet, by calling abort on the transaction object.

The following is an example of calling abort on a transaction:

```
var dbTrans = g_db.transaction([“Employees”], “readwrite”);
var dbObjectStore = dbTrans.objectStore(“Employees”);
var dbAddRequest = dbObjectStore.add(objEmployee);
/* ...success and error handlers... */
dbTrans.abort();
```

Success events don't bubble and can't be cancelled, but error events do bubble and may be cancelled.

Error events will abort the transaction in which they occur unless they're cancelled by calling preventDefault().

I would generally recommend against canceling an error so that a transaction doesn't abort: if there's an error, typically it's because there's a problem, and you could end up with bad data in your database.

That said, you might run across special cases where certain errors are acceptable and you may simply log the error allowing the transaction to continue.

Viewing the data in an IndexedDB

Currently, the only browser with built-in tools to view the contents of an IndexedDB database is Google Chrome.

In Chrome, if you open up the Developer Tools (Tools, Developer Tools menu or by pressing Ctrl+Shift+I), you'll find a Resources tab with an IndexedDB item that you can expand to explore any databases currently held by your browser.

The following Microsoft website details a small web application allowing you to explore IndexedDB databases in other browsers like Internet Explorer:

http://blogs.msdn.com/b/ie/archive/2012/01/25/debugging-indexeddb-applications.aspx

KNOWN ISSUES

For security reasons, IndexedDB will only work for websites using http or https. Browsers will not allow a local file (file://) to use IndexedDB.

The only exception to the local file restriction is when IndexedDB is accessed from the JavaScript of a Windows 8 application (ms-wwa or ms-wwa-web protocols in Internet Explorer)

Like with any other data stored in the browser, the user has the ability to remove your databases by deleting the cookies and other local browser data.

This could be a bit of an edge case, but it may also help in determining why IndexedDB isn't working for one user when it works for others: It is possible to disable IndexedDB in Internet Explorer, Chrome, and Firefox:

- In IE, the setting can be found in Internet Options on the General tab. The Settings button brings up a Website Data Settings window. Click on the Caches and Databases tab and you will find an 'Allow website caches and databases' checkbox. This checkbox needs to be on to support IndexedDB in IE.
- In Firefox, if you enter about:config in the address bar, one of the settings you will find is dom.indexedDB.enabled. If the value column is set to false, double-click on the row to toggle the value to true.
- In Chrome, if you open up Settings and then show the advanced settings, you will find a Privacy Settings section with a 'Content settings...' button. Clicking that button brings up a window with a Cookies section at the top. 'Allow local data to be set' needs to be enabled.

EXAMPLE CODE

Some sample code has been written to show a possible real-world use case of IndexedDB and can be found in the following GitHub repository: https://github.com/cggallant/IndexedDB

RESOURCES

http://caniuse.com/ can help you discover which browsers support a particular feature and in which browser version the feature became available.

As mentioned earlier in this article, the following is a link to a special JavaScript shim that mimics the IndexedDB API in browsers that only support the deprecated HTML5 Web SQL API: http://nparashuram.com/IndexedDBShim/

Of the browsers that support IndexedDB, only the BlackBerry 10 browser still uses vendor prefixes for IndexedDB which is a good sign since browser vendors usually don't remove the vendor prefixes from new technologies until they are fairly certain that there won't be any more major changes to the specification. That said, IndexedDB is still a working draft and some aspects may still change in time. If you find that what you read in books or on the internet is not working as you expect, I find it's often easier to go back to the source and read the latest published version of the W3C specification: http://www.w3.org/TR/IndexedDB/

Due to the fast pace of browser updates, information and tutorials on HTML5 often grows outdated quite quickly. I recommend reading the HTML5 specifications first, if possible, but because specifications are sometimes difficult to understand, I try to cross-reference what I read in the specifications with various articles and by writing test code.

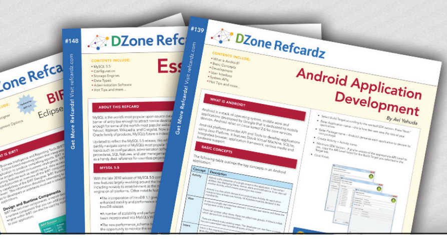
I've found the following sites helpful in understanding some of the fine details of various technologies:

- Mozilla Developer Network: https://developer.mozilla.org/en-US/docs/HTML/HTML5
- HTML5 Zone: www.DZone.com/mz/html5

ABOUT THE AUTHOR

Gerard Gallant is a Senior Software Developer / Software Architect with Dovico Software. For over a decade, he has played a major role in developing most software products released by Dovico ranging from the creation of the Microsoft Project link ActiveX control to the creation of the new Hosted Services API. He is married with two beautiful and smart girls. He unwinds by shooting hoops or watching a movie and also enjoys traveling extensively. Recent publications can be found on his blog: http://cggallant.blogspot.ca/

RECOMMENDED BOOK



DZone

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome

refcardz@dzone.com

Sponsorship Opportunities

sales@dzone.com

ISBN-13: 978-1-936502-80-6
ISBN-10: 1-936502-80-1
50795
9 781936 502806
$7.95

Version 1.0