Contents

[Introduction to IndexedDB 4](#_Toc369434414)

[What Is IndexedDB 4](#_Toc369434415)

[Asynchronous API 4](#_Toc369434416)

[Browser compatibility 4](#_Toc369434417)

[Storage limits 5](#_Toc369434418)

[Why Use IndexedDB 5](#_Toc369434419)

[How it differs to Modern Relational Databases 5](#_Toc369434420)

[Usages for IndexedDB 5](#_Toc369434421)

[IndexedDB Architectural Components 5](#_Toc369434422)

[IndexedDB Interface Objects 6](#_Toc369434423)

[Using the Code 6](#_Toc369434424)

[Callback Handlers 6](#_Toc369434425)

[Checking for IndexedDB support 7](#_Toc369434426)

[Creating and opening a database 7](#_Toc369434427)

[Version changes while a web application is open in another browser tab 8](#_Toc369434428)

[Database versioning 8](#_Toc369434429)

[Creating an Object Store (table) 8](#_Toc369434430)

[Creating a Key 8](#_Toc369434431)

[Creating a Name Index 8](#_Toc369434432)

[Creating Transactions 9](#_Toc369434433)

[1.     Scope 9](#_Toc369434434)

[2.     Mode 9](#_Toc369434435)

[Adding Data 9](#_Toc369434436)

[Removing Data 10](#_Toc369434437)

[Updating Data 10](#_Toc369434438)

[Clearing All Data\Object Store 11](#_Toc369434439)

[Cursors 11](#_Toc369434440)

[Using Cursor Ranges 12](#_Toc369434441)

[Using a Cursor to Get All the Records 12](#_Toc369434442)

[How to Filter by Multiple Fields (including non-indexed fields) 13](#_Toc369434443)

[Perform a Record Count 15](#_Toc369434444)

[Security 15](#_Toc369434445)

[Warning about Browser Shutdown 15](#_Toc369434446)

[Browser Support 17](#_Toc369434447)

[View Database Data\Components in Chrome Developer 19](#_Toc369434448)

[Conclusion 20](#_Toc369434449)

[References 20](#_Toc369434450)

# 

# Introduction to IndexedDB

## What Is IndexedDB

The IndexedDB database is a relatively new HTML5 web database that allows your HTML5 web application to store data associated with a *host\protocol\port,* locally on the client’s hard-drive. Unlike [HTML5 localstorage](http://www.w3schools.com/html/html5_webstorage.asp) which lets you store data using a simple key-value pair only, the IndexedDB is more powerful and useful for applications that requires you to store a large amount of data. In addition, with its rich queries abilities, these applications can load faster and more responsive than having to perform a server side transaction and send the result to be displayed within the client’s html grid for example.

An IndexedDB is basically a persistent data store in the browser—a database on the client side.  Like regular relational databases, it maintains indexes over the records it stores, and developers use the IndexedDB JavaScript API to locate records by key or by looking up an index.  Each database is scoped by “origin,” i.e. the domain of the site that creates the database.

*If you’re new to IndexedDB, you can start here:*

1. Developers guide on [MSDN](http://msdn.microsoft.com/en-us/ie/hh440436.aspx#_IndexedDB)
2. Spec on [W3C](http://www.w3.org/TR/IndexedDB/)

## Asynchronous API

The IndexedDB API revolves around asynchronous methods that return without blocking the calling thread. To get asynchronous access to a database, call [open](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBFactory#open)() on the [indexedDB](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBEnvironment#attr_indexedDB) attribute of a [window](https://developer.mozilla.org/en-US/docs/DOM/window) object. This method returns an IDBRequest object (IDBOpenDBRequest); asynchronous operations communicate to the calling application by firing events on IDBRequest objects.

### Browser compatibility

#### Desktop

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Chrome** | **Firefox (Gecko)** | **Internet Explorer** | **Opera** | **Safari (WebKit)** |
| Asynchronous API | 24.0 11.0 webkit | 16.0 (16.0) 4.0 (2.0) moz | 10 | 15.0 | Not supported |
| Synchronous API (used with [WebWorkers](https://developer.mozilla.org/en-US/docs/DOM/Using_web_workers" \o "https://developer.mozilla.org/en-US/docs/Using_web_workers)) | Not supported | Not supported See [bug 701634](https://bugzilla.mozilla.org/show_bug.cgi?id=701634) | Not supported | Not supported | Not supported |

#### Mobile

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Android** | **Firefox Mobile (Gecko)** | **IE Phone** | **Opera Mobile** | **Safari Mobile** |
| Asynchronous API | Not supported | 6.0 (6.0) moz | Not supported | Not supported | Not supported |

## Storage limits

1. Firefox: no limit on the IndexedDB database's size. The user interface will ask permission for storing blobs bigger than 50 MB. This size quota can be customized through the dom.indexedDB.warningQuota preference (which is defined in <http://mxr.mozilla.org/mozilla-central/source/modules/libpref/src/init/all.js>).
2. Google Chrome: see [https://developers.google.com/chrome...rage#temporary](https://developers.google.com/chrome/whitepapers/storage#temporary)
3. IE10 storage size for each app is 250MB: see <http://msdnrss.thecoderblogs.com/2012/12/using-html5javascript-in-windows-store-apps-data-access-and-storage-mechanism-ii/>

## Why Use IndexedDB

W3C announced that the Web SQL database (another option for HTML5 storage) is a deprecated specification, and web developers should not use this technology anymore. Instead, they recommend the use of its replacement – **IndexedDB**, the new HTML5 data storage to manipulate their data on client-side.

## How it differs to Modern Relational Databases

IndexedDB does not have the concept of a relational relationship between objects that is not to say you cannot filter out other object stores based on the values from a query against another object store. You should think of object stores as class objects with properties that may have a unique field (if not you would use the *auto generate* key option when creating an object store).

## Usages for IndexedDB

The main usage of IndexedDB API is to store data locally on browser side, so that offline mode is supported, for e.g. you could perform CUD (create, update & delete) operations for e.g. on an employee database table for your company’s HR module. This will make things faster and reduce network latency once online again with the database server.

The synchronisation of data from the client to the backend relational database server, is something that you will have to design\implement yourself – but realistically, this is a matter of reserialising your stored json (client) objects back into server side classes and perform a database action on each edited class object.

## IndexedDB Architectural Components

1. Each origin (host, protocol, and port) has its own set of databases. A unique name identifies each database within an origin. IndexedDB has a same-origin policy, which requires that the database and the application be from the same origin.
2. A database is identified by a name and version number. A database can have only one version at a time.
3. An object store is identified by a unique name. You can create an object store only during an “upgrade needed” event. You store data in records in an object store. A database can have multiple named object stores.
4. A transaction provides reliable data access and data modification on a database. All interactions with the data in the database must happen within the scope of a transaction.
5. A record is a key-value pair, where the key is a unique identifier for the corresponding data value. You can set your own keys or you can have the object store create them for you.
6. An index is a specialized object store that maps database keys to the key field in the saved object. Using an index is optional.
7. An application may use multiple databases, each of which may have multiple object stores, each of which may have multiple records.

## IndexedDB Interface Objects

1. [IDBFactory](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBFactory) provides access to a database. This is the interface implemented by the global object IndexedDB and is therefore the entry point for the API.
2. [IDBCursor](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBCursor) iterates over object stores and indexes.
3. [IDBCursorWithValue](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBCursorWithValue) iterates over object stores and indexes and returns the cursor's current value.
4. [IDBDatabase](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBDatabase) represents a connection to a database. It's the only way to get a transaction on the database.
5. [IDBEnvironment](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBEnvironment) provides access to a client-side database. It is implemented by [window](https://developer.mozilla.org/en-US/docs/DOM/window) objects.
6. [IDBIndex](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBIndex) provides access to the metadata of an index.
7. [IDBKeyRange](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBKeyRange) defines a range of keys.
8. [IDBObjectStore](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBObjectStore) represents an object store.
9. [IDBOpenDBRequest](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBOpenDBRequest) represents a request to open a database.
10. [IDBRequest](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBRequest) provides access to results of asynchronous requests to databases and database objects. It is what you get when you call an asynchronous method.
11. [IDBTransaction](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBTransaction) represents a transaction. You create a transaction on a database, specify the scope (such as which object stores you want to access), and determine the kind of access (read only or write) that you want.
12. [IDBVersionChangeEvent](https://developer.mozilla.org/en-US/docs/IndexedDB/IDBVersionChangeEvent) indicates that the version of the database has changed.

# Using the Code

## Callback Handlers

The asynchronous design of IndexedDB, means that callbacks are needed to process the return values of a transaction be it in an erroneous or successful state. The callbacks are like any JavaScript asynchronous callback approach (see below):

request.onerror = function (event) {

// Do something with request.errorCode!

};

request.onsuccess = function (event) {

// Do something with request.result!

};

// Generic error handler for all errors targeted at this database's requests!

db.onerror = function (event) {

alert("Database error: " + event.target.errorCode);

};

## Checking for IndexedDB support

Your application will need to perform a verification check, to determine if your browser supports IndexedDB.

if (!window.indexedDB) {

window.alert("Your browser doesn't support a stable version of IndexedDB. Such and such feature will not be available.");

}

## Creating and opening a database

The snippet of code below, will delete the database if it already exists, then perform a create action, within the cerate’s success callback – to perform the creation of indexes etc. When creating the database, the event ‘onupgradeneeded’ is called first then the ‘onsuccess’ callback.

function createDatabase() {

var deleteDbRequest;

try {

if (localDatabase.db != null) localDatabase.db.close();

deleteDbRequest = localDatabase.indexedDB.deleteDatabase(dbName); // delete old db

deleteDbRequest.onsuccess = function (event) {

writeToConsoleScreen('Started database create - ' + dbName);

var openRequest = localDatabase.indexedDB.open(dbName, 1); //version used

openRequest.onerror = function (e) {

writeToConsoleScreen("Database error: " + e.target.errorCode);

};

openRequest.onsuccess = function (event) {

writeToConsoleScreen('Finished database create - ' + dbName);

localDatabase.db = openRequest.result;

};

openRequest.onupgradeneeded = function (evt) {

// check if OS\table not already added

if (!evt.currentTarget.result.objectStoreNames.contains(osTableName)) {

writeToConsoleScreen("Started creating object-store - '" + osTableName);

var employeeStore = evt.currentTarget.result.createObjectStore(osTableName, { keyPath: "recid" }); // key id ID

employeeStore.createIndex("lnameIndex", "lname", { unique: false });

employeeStore.createIndex("emailIndex", "email", { unique: true }); // email has to be unique (a constraint)

employeeStore.createIndex("sdateIndex", "sdate", { unique: false });

}

writeToConsoleScreen("Finished creating object-store - '" + osTableName); // onupgradeneeded called first

};

deleteDbRequest.onerror = function (e) {

writeToConsoleScreen("Database error: " + e.target.errorCode);

};

};

}

catch (e) {

writeToConsoleScreen(e.message);

}

}

### Version changes while a web application is open in another browser tab

When your web application changes in such a way that a version change is required for your database, you need to consider what happens if the user has the old version of your application open in one tab and then loads the new version of your app in another. When you call open() with a greater version than the actual version of the database, all other open databases must explicitly acknowledge the request before you can start making changes to the database. Here's how it works:

### Database versioning

IndexedDB databases have a version string associated with them.  This can be used by web applications to determine whether the database on a particular client has the latest structure or not.

This is useful when you make changes to your database’s data model and want to propagate those changes to existing clients who are on the previous version of your data model.  **You can simply change the version number for the new structure and check for it the next time the user runs your app**

## Creating an Object Store (table)

Once the database Open method has been called, the ‘onupgradeneeded’ callback method will be executed if a newer database version has been specified.

var openRequest = localDatabase.indexedDB.open(dbName, 2); //version used

openRequest.onupgradeneeded = function (evt) {

// check if OS\table not already added

if (!evt.currentTarget.result.objectStoreNames.contains(osTableName)) {

writeToConsoleScreen("Started creating object-store - '" + osTableName);

var employeeStore = evt.currentTarget.result.createObjectStore(osTableName, { keyPath: "recid" }); // key id ID

employeeStore.createIndex("lnameIndex", "lname", { unique: false });

employeeStore.createIndex("emailIndex", "email", { unique: true }); // email has to be unique (a constraint)

employeeStore.createIndex("sdateIndex", "sdate", { unique: false });

}

writeToConsoleScreen("Finished creating object-store - '" + osTableName); // onupgradeneeded called first

};

## Creating a Key

var employeeStore = evt.currentTarget.result.createObjectStore(osTableName, { keyPath: "recid" });

## Creating a Name Index

employeeStore.createIndex("lnameIndex", "lname", { unique: false });

employeeStore.createIndex("emailIndex", "email", { unique: true }); // email has to be unique (a constraint)

employeeStore.createIndex("sdateIndex", "sdate", { unique: false });

## Creating Transactions

Like relational databases, IndexedDB also performs all of its I/O operations under the context of transactions.  Transactions are created through connection objects and enable atomic, durable data access and mutation.  There are two key attributes for transaction objects:

### 1.     Scope

The scope determines which parts of the database can be affected through the transaction.  This basically helps the IndexedDB implementation determine what kind of isolation level to apply during the lifetime of the transaction.  Think of the scope as simply a list of tables (known as “object stores”) that will form a part of the transaction.

### 2.     Mode

The transaction mode determines what kind of I/O operation is permitted in the transaction.  The mode can be:

1. Read only  
   Allows only “read” operations on the objects that are a part of the transaction’s scope.
2. Read/write  
   Allows “read” and “write” operations on the objects that are a part of the transaction’s scope.
3. Version change  
   The “version change” mode allows “read” and “write” operations and also allows the creation and deletion of object stores and indexes.

Transaction objects auto-commit themselves unless they have been explicitly aborted.  Transaction objects expose events to notify clients of:

* when they complete
* when they abort and
* when they timeout

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

writeToConsoleScreen('Started adding 10,000 records');

var transaction = localDatabase.db.transaction(osTableName, "readwrite");

## Adding Data

if (transaction) {

transaction.oncomplete = function () {

}

transaction.onabort = function () {

writeToConsoleScreen("transaction aborted.");

localDatabase.db.close();

}

transaction.ontimeout = function () {

writeToConsoleScreen("transaction timeout.");

localDatabase.db.close();

}

var store = transaction.objectStore(osTableName);

if (store) {

var req;

var customer = {};

for (var loop = 0; loop < 10000; loop++) {

customer = {};

customer.recid = loop; // past existing range of previous inserts

customer.fname = 'Susan';

customer.lname = 'Ottie';

customer.email = 'NewEmployee@' + loop + '.com';

customer.sdate = '4/3/2012';

req = store.add(customer);

req.onsuccess = function (ev) {

}

req.onerror = function (ev) {

writeToConsoleScreen("Failed to add record." + " Error: " + ev.message);

}

}

writeToConsoleScreen("Finished adding 10,000 records");

}

}

## Removing Data

function deleteEmployee() {

try {

writeToConsoleScreen('Started deleting record # 7');

var transaction = localDatabase.db.transaction(osTableName, "readwrite");

var store = transaction.objectStore(osTableName);

var jsonStr;

var employee;

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

var request = store.delete(7);

request.onsuccess = function (e) {

writeToConsoleScreen('Finished deleting record # 7');

fetchAllEmployees();

};

request.onerror = function (e) {

writeToConsoleScreen(e);

};

}

}

catch (e) {

console.log(e);

}

}

## Updating Data

function updateEmployee() {

try {

writeToConsoleScreen('Started record update');

var transaction = localDatabase.db.transaction(osTableName, "readwrite");

var store = transaction.objectStore(osTableName);

var jsonStr;

var employee;

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

store.get(7).onsuccess = function(event) {

employee = event.target.result;

// save old value

jsonStr = "Old: " + JSON.stringify(employee);

writeToConsoleScreen(jsonStr);

// update record

employee.email = "bert.oneill@kofax.com";

jsonStr = "New: " + JSON.stringify(employee);

var request = store.put(employee);

request.onsuccess = function (e) {

writeToConsoleScreen("Finished Updating employee - " + jsonStr);

w2ui.grid.clear();

records = [];

fetchAllEmployees();

};

request.onerror = function (e) {

writeToConsoleScreen("Error " + e.value);

};

};

}

}

catch(e){

console.log(e);

}

}

## Clearing All Data\Object Store

function clearAllEmployees() {

try {

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

writeToConsoleScreen('Started clearing records');

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

store.clear().onsuccess = function (event) {

writeToConsoleScreen('Finished clearing records');

records = [];

w2ui.grid.clear();

};

}

}

catch(e){

console.log(e);

}

}

## Cursors

The IndexedDB way of enumerating records from an object store is to use a “cursor” object.  A cursor can iterate over records from an underlying object store or an index.  A cursor has the following key properties:

1. A **range** of records in either an index or an object store.
2. A **source** that references the index or object store that the cursor is iterating over.
3. A **position** indicating the current position of the cursor in the given range of records.

While the concept of a cursor is fairly straightforward, writing the code to actually iterate over an object store is somewhat tricky given the asynchronous nature of all the API calls.  Let’s implement the listNotes method of our NotesStore object and see what the code looks like.

Let’s break this implementation down:

1. First, we acquire a transaction object by calling the database object’s transaction method.  Note that this time we’re indicating that we require a “read-only” transaction.
2. Next we retrieve a reference to the object store via the objectStore method of the transaction object.
3. Then we issue an async call to the openCursor API on the object store.  The tricky part here is that every single iteration over a record in the cursor is itself an async operation!  To prevent the code from drowning in a sea of callbacks, we define a local function called iterate to encapsulate the logic of iterating over every record in the cursor.
4. This iterate function makes an async call to the cursor object’s move method and recursively invokes itself again in the callback if it detects that there are more rows to be retrieved.  Once all the rows in the cursor have been retrieved we finally invoke the callback method passed by the caller handing in the retrieved data as a parameter.

## Using Cursor Ranges

ToDo

## Using a Cursor to Get All the Records

When the app starts, if there’s an existing database and that database has milestones, the count\_down app reads the records from the database and initializes the internal list of milestones from that information.

The fetchAllEmployees() method gets called after the database has been successfully opened. It is within this method that the database is read.

function fetchAllEmployees() {

try {

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

records = [];

if (!localDatabase.db.objectStoreNames.contains(osTableName)) {

writeToConsoleScreen("No employees table exists - click on Create");

return;

}

writeToConsoleScreen('Started fetching all recrds');

var store = localDatabase.db.transaction(osTableName).objectStore(osTableName);

var request = store.openCursor();

request.onsuccess = function (evt) {

var cursor = evt.target.result;

if (cursor) {

var employee = cursor.value;

records.push(employee);

cursor.continue();

}

else {

try {

writeToConsoleScreen('Finished fetching ' + records.length + ' recrds');

w2ui.grid.clear();

w2ui.grid.add(records); // bind to grid - auto refresh

} catch (ex) {

writeToConsoleScreen("Exception..." + ex);

}

}

};

}

}

catch (e) {

writeToConsoleScreen(e.message);

}

}

Because getting records does not modify the database, this transaction is readonly.

You can use a [Cursor](https://api.dartlang.org/dart_indexedDb/Cursor.html) object to step through the records in the database one by one, creating a Milestone object for each. The object store uses a stream to fire an event for each record retrieved from the database. By listening on the stream, your code can be notified for each record read. The count\_down app creates a corresponding Milestone object in the internal list in memory for each database record retrieved.

Open a cursor on a transaction’s object store with openCursor. The cursor indicates the current position in the database. The count\_down app opens a cursor and sets the optional autoAdvance argument to true. This means that after reading a record from the database and returning its value, the cursor advances automatically to the next record. If you don’t use autoAdvance, your code has to use the next() method to advance the cursor to the next record.

The openCursor() method returns a stream on which you can listen for events. Here, the stream is a broadcast stream, so the app can listen both for read events and for a final count of records retrieved.

For each record retrieved from the database, an event fires and the callback function is called. A [CursorWithValue](https://api.dartlang.org/dart_indexedDb/CursorWithValue.html" \t "_blank) object, named cursor in this example, is passed to the callback function. Use cursor.key to get the key for the record just retrieved. Usecursor.value to get the value for that record.

## How to Filter by Multiple Fields (including non-indexed fields)

function fetchMultiFilterByEmailAndSurname() {

try {

records = [];

writeToConsoleScreen("Started fetching records by multiple filters");

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

var range = IDBKeyRange.only("Silver");

var store = localDatabase.db.transaction(osTableName).objectStore(osTableName);

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

index.openCursor(range).onsuccess = function (evt) {

var cursor = evt.target.result;

if (cursor) {

var employee = cursor.value;

if (employee.email.indexOf('1') > 0) { // filter by another field in json object (could of used the extra indexes)

var jsonStr = JSON.stringify(employee);

records.push(employee);

writeToConsoleScreen(jsonStr);

}

cursor.continue();

}

else {

writeToConsoleScreen("Finished adding records by multiple filters - found " + records.length);

w2ui.grid.clear();

w2ui.grid.add(records); // bind to grid - auto refresh

}

};

}

}

catch (e) {

writeToConsoleScreen(e.message);

}

}

## Perform a Record Count

The IndexedDB framework as of yet does not allow for a simple way to perform a record count (though going by the W3C specification this will change). But by using a cursor, it is relatively simple to get a record count of an object store.

function countRecords()

{

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

writeToConsoleScreen("Starting count");

var transaction = localDatabase.db.transaction(osTableName, "readwrite");

if (transaction) {

transaction.oncomplete = function () {

}

transaction.onabort = function () {

writeToConsoleScreen("transaction aborted.");

localDatabase.db.close();

}

transaction.ontimeout = function () {

writeToConsoleScreen("transaction timeout.");

localDatabase.db.close();

}

var store = transaction.objectStore(osTableName);

if (store) {

var keyRange = IDBKeyRange.lowerBound(0);

var cursorRequest = store.openCursor(keyRange);

var count = 0;

cursorRequest.onsuccess = function (e) { // success called for each cursor action

var result = e.target.result;

result ? ++count && result.continue() : alert(count);

};

}

}

}

else

{

writeToConsoleScreen("Database needs to be created first");

}

}

# 

# Security

IndexedDB uses the same-origin principle, which means that it ties the store to the origin of the site that creates it (typically, this is the site domain or subdomain), so it cannot be accessed by any other origin.

It's important to note that IndexedDB doesn't work for content loaded into a frame from another site (either [<frame>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/frame) or [<iframe>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe). This is a security and privacy measure and can be considered analogous the blocking of 3rd-party cookies.  For more details, see [bug 595307](https://bugzilla.mozilla.org/show_bug.cgi?id=595307)

# Warning about Browser Shutdown

When the browser shuts down (e.g., when the user selects Exit or clicks the Close button), any pending IndexedDB transactions are (silently) aborted -- they will not complete, and they will not trigger the error handler.  Since the user can exit the browser at any time, this means that you cannot rely upon any particular transaction to complete or to know that it did not complete.  There are several implications of this behaviour.

First, you should take care to always leave your database in a consistent state at the end of every transaction.  For example, suppose that you are using IndexedDB to store a list of items that you allow the user to edit.  You save the list after the edit by clearing the object store and then writing out the new list.  If you clear the object store in one transaction and write the new list in another transaction, there is a danger that the browser will close after the clear but before the write, leaving you with an empty database.  To avoid this, you should combine the clear and the write into a single transaction.

Second, you should never tie database transactions to unload events.  If the unload event is triggered by the browser closing, any transactions created in the unload event handler will never complete.  An intuitive approach to maintaining some information across browser sessions is to read it from the database when the browser (or a particular page) is opened, update it as the user interacts with the browser, and then save it to the database when the browser (or page) closes.  However, this will not work.  The database transactions will be created in the unload event handler, but because they are asynchronous they will be aborted before they can execute.

In fact, there is no way to guarantee that IndexedDB transactions will complete, even with normal browser shutdown.  See [bug 870645](https://bugzilla.mozilla.org/show_bug.cgi?id=870645).

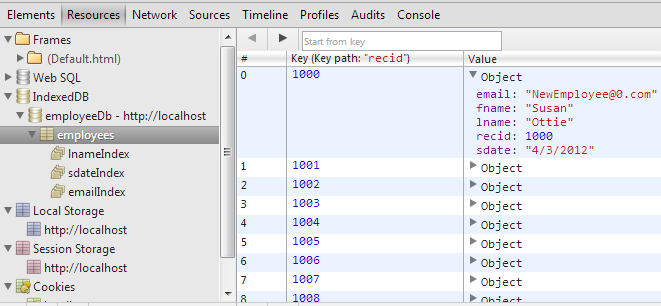
# Browser Support

|  |
| --- |
|  |
| IE | Firefox | Chrome | Safari | Opera | iOS Safari | Chrome for Android | IE Mobile |
| 26 versions back |  |  | 4.0: Not supported |  |  |  |  |
| 25 versions back |  |  | 5.0: Not supported |  |  |  |  |
| 24 versions back |  | 2.0: Not supported | 6.0: Not supported |  |  |  |  |
| 23 versions back |  | 3.0: Not supported | 7.0: Not supported |  |  |  |  |
| 22 versions back |  | 3.5: Not supported | 8.0: Not supported |  |  |  |  |
| 21 versions back |  | 3.6: Not supported | 9.0: Not supported |  |  |  |  |
| 20 versions back |  | 4.0: Partial supportmoz | 10.0: Not supported |  |  |  |  |
| 19 versions back |  | 5.0: Partial supportmoz | 11.0: Partial supportwebkit |  |  |  |  |
| 18 versions back |  | 6.0: Partial supportmoz | 12.0: Partial supportwebkit |  |  |  |  |
| 17 versions back |  | 7.0: Partial supportmoz | 13.0: Partial supportwebkit |  |  |  |  |
| 16 versions back |  | 8.0: Partial supportmoz | 14.0: Partial supportwebkit |  |  |  |  |
| 15 versions back |  | 9.0: Partial supportmoz | 15.0: Partial supportwebkit |  |  |  |  |
| 14 versions back |  | 10.0: Supportedmoz | 16.0: Partial supportwebkit |  |  |  |  |
| 13 versions back |  | 11.0: Supportedmoz | 17.0: Partial supportwebkit |  | 9.0: Not supported |  |  |
| 12 versions back |  | 12.0: Supportedmoz | 18.0: Partial supportwebkit |  | 9.5-9.6: Not supported |  |  |
| 11 versions back |  | 13.0: Supportedmoz | 19.0: Partial supportwebkit |  | 10.0-10.1: Not supported |  |  |
| 10 versions back |  | 14.0: Supportedmoz | 20.0: Partial supportwebkit |  | 10.5: Not supported |  |  |
| 9 versions back |  | 15.0: Supportedmoz | 21.0: Partial supportwebkit |  | 10.6: Not supported |  |  |
| 8 versions back |  | 16.0: Supported | 22.0: Partial supportwebkit |  | 11.0: Not supported |  |  |
| 7 versions back |  | 17.0: Supported | 23.0: Supportedwebkit |  | 11.1: Not supported |  |  |
| 6 versions back |  | 18.0: Supported | 24.0: Supported |  | 11.5: Not supported | 10.0: Not supported |  |
| 5 versions back | 5.5: Not supported | 19.0: Supported | 25.0: Supported | 3.1: Not supported | 11.6: Not supported | 11.0: Not supported |  |
| 4 versions back | 6.0: Not supported | 20.0: Supported | 26.0: Supported | 3.2: Not supported | 12.0: Not supported | 11.1: Not supported |  |
| 3 versions back | 7.0: Not supported | 21.0: Supported | 27.0: Supported | 4.0: Not supported | 12.1: Not supported | 11.5: Not supported |  |
| 2 versions back | 8.0: Not supported | 22.0: Supported | 28.0: Supported | 5.0: Not supported | 15.0: Supported | 12.0: Not supported |  |
| Previous version | 9.0: Not supported | 23.0: Supported | 29.0: Supported | 5.1: Not supported | 16.0: Supported | 12.1: Not supported |  |
| Current | 10.0: Supported | 24.0: Supported | 30.0: Supported | 6.0: Not supported | 17.0: Supported | 16.0: Supported | 24.0: Supported |
| Near future | 11.0: Supported | 25.0: Supported | 31.0: Supported | 7.0: Support unknown |  |  |  |
| Farther future |  | 26.0: Supported | 32.0: Supported |  |  |  |  |

[Link to data](http://caniuse.com/indexeddb)

## 

# View Database Data\Components in Chrome Developer



# Conclusion

The HTML5 IndexedDB API is very useful and powerful. You can leverage it to create rich, online and offline HTML5 application. In addition, with IndexedDB API, you can cache data to make traditional web applications especially mobile web applications load faster and more responsive without need to retrieve data from the web server each time (especially ideal for static or long living data).

# References

1. [IndexedDB API Reference](https://developer.mozilla.org/en/IndexedDB)
2. [Indexed Database API Specification](http://www.w3.org/TR/IndexedDB/)
3. [Using IndexedDB in chrome](https://developer.mozilla.org/en-US/docs/IndexedDB/Using_IndexedDB_in_chrome)
4. <https://developer.mozilla.org/en/docs/IndexedDB>
5. Cookbook demo on [IETestDrive](http://ie.microsoft.com/testdrive/HTML5/Cookbook/)
6. IE10 Updates - <https://blogs.msdn.com/b/ie/archive/2012/03/21/indexeddb-updates-for-ie10-and-metro-style-apps.aspx?Redirected=true>
7. Microsoft Labs - <http://www.html5labs.com/prototypes/indexeddb/indexeddb/download>