Contents

[Introduction to IndexedDB 3](#_Toc370035972)

[What Is IndexedDB 3](#_Toc370035973)

[Asynchronous API 3](#_Toc370035974)

[Browser compatibility 4](#_Toc370035975)

[Storage limits 4](#_Toc370035976)

[Why Use IndexedDB 4](#_Toc370035977)

[How it differs to Modern Relational Databases 4](#_Toc370035978)

[Usages for IndexedDB 5](#_Toc370035979)

[IndexedDB Architectural Components 5](#_Toc370035980)

[IndexedDB Interface Objects 5](#_Toc370035981)

[Using the Code 6](#_Toc370035982)

[Callback Handlers 6](#_Toc370035983)

[Checking for IndexedDB support 6](#_Toc370035984)

[Creating and opening a database 7](#_Toc370035985)

[Version changes while a web application is open in another browser tab 8](#_Toc370035986)

[Database versioning 8](#_Toc370035987)

[Creating an Object Store (table) 8](#_Toc370035988)

[Creating a Key 8](#_Toc370035989)

[Creating a Name Index 8](#_Toc370035990)

[Creating Transactions 9](#_Toc370035991)

[1.     Scope 9](#_Toc370035992)

[2.     Mode 9](#_Toc370035993)

[Adding Data 10](#_Toc370035994)

[Removing Data 11](#_Toc370035995)

[Updating Data 12](#_Toc370035996)

[Clearing All Data\Object Store 13](#_Toc370035997)

[Cursors 14](#_Toc370035998)

[Using Cursor Ranges 14](#_Toc370035999)

[IDBKeyrange 14](#_Toc370036000)

[Using a Cursor to Get All the Records 15](#_Toc370036001)

[How to Filter by Multiple Fields (including non-indexed fields) 16](#_Toc370036002)

[Perform a Record Count 17](#_Toc370036003)

[Security 18](#_Toc370036004)

[Warning about Browser Shutdown 18](#_Toc370036005)

[No Cross Browser Sharing (even same origin) 18](#_Toc370036006)

[Browser Support 19](#_Toc370036007)

[View Database Data\Components in Chrome Developer 22](#_Toc370036008)

[Code 23](#_Toc370036009)

[ 23](#_Toc370036010)

[Conclusion 23](#_Toc370036011)

[References 23](#_Toc370036012)

# 

# Introduction to IndexedDB

## What Is IndexedDB

The IndexedDB database is a relatively new, in the sense that it replaced the older (W3C deprecated) [Web SQL](http://html5doctor.com/introducing-web-sql-databases/) database. The IndexedDB web database allows your HTML5 web application to store data associated with a *host/protocol/port,* locally on the client’s hard-drive. Unlike LocalStorage, 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 dropdown 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 applications 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)) | 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**.

## How it differs to Modern Relational Databases

IndexedDB does not have the concept of a relational relationship between objects, which 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 is to store data locally on browser\client side, so that offline mode is supported within your application, for e.g. you could perform CUD (create, update & delete) operations on an employee database table for your company’s HR module. This will reduce network latency once online again with the database server.

The synchronisation of data between the client and 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 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. The value can be a serialised JSON object.
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. An object-store can have multiple indexes (not the same concept as cluster indexes – as only one index can be used at a time when querying an object-store).

**NB:** An application may use multiple databases, each of which may have multiple object stores, each of which may have multiple indexes.

## 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!

};

A common callback is to catch error at a global level (throwing errors).

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

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 –create the indexes etc. When creating the database, the event ‘onupgradeneeded’ is called first then the ‘onsuccess’ or ‘onerror’ 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) {

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

openRequest.onerror = function (e) {

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

};

openRequest.onsuccess = function (event) {

localDatabase.db = openRequest.result;

};

openRequest.onupgradeneeded = function (evt) {

// check if OS\table not already added

if (!evt.currentTarget.result.objectStoreNames.contains(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 });

}

};

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 application.

## 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)) {

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
* when they timeout

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

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

## Adding Data

The snippet below will create 10,000 records and add them to the employee object store.

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 = {};

// create ten thousand records

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

customer = {};

customer.recid = loop;

customer.fname = 'Susan';

customer.lname = 'Ottie';

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

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);

}

}

}

}

## Removing Data

The snippet of code below will remove the record that has a “recid” of 7.

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) {

fetchAllEmployees();

};

request.onerror = function (e) {

writeToConsoleScreen(e);

};

}

}

catch (e) {

console.log(e);

}

}

## Updating Data

The snippet of code below will retrieve the record with a “recid” of 7 and update its email address, then PUT it back into the object-store.

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);

};

request.onerror = function (e) {

writeToConsoleScreen("Error " + e.value);

};

};

}

}

catch(e){

writeToConsoleScreen("Error " + e.value); }

}

## Clearing All Data\Object Store

function clearAllEmployees() {

try {

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

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

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

writeToConsoleScreen('Finished clearing records');

};

}

}

catch(e){

writeToConsoleScreen("Error " + e.value);

}

}

## Cursors

The IndexedDB way of enumerating records from an object store is to use a “cursor” object.  A cursor will then iterate over records from an underlying object-store.  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.

To perform asynchronous cursor fetches, you will need use a recursive programming strategy. Below is an example of such an action to loop over a cursor:

The snippet of code below, will for each record in the “records” array call the “addData” recursively.

function addData(txn, store, records, i, commitT) {

try {

if (i < records.length) {

var rec = records[i];

var req = store.add(rec);

req.onsuccess = function (ev) {

i++;

addData(txn, store, records, i, commitT);

}

req.onerror = function (ev) {

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

}

}

else if (i == records.length) {

// writeToConsoleScreen('Finished adding ' + records.length + " records");

}

}

catch (e) {

writeToConsoleScreen(e.message);

}

}

## 

## Using Cursor Ranges

In the IDBObjectStore interface, we have the “openCursor” method to create a new cursor for retrieving data. In the IDBIndex interface, we have 2 ways to create a new cursor. These methods are “openCursor” to retrieve the values from the index and “openKeyCursor” to retrieve the keys. There are two optional parameters that can be provided when calling these methods. The first parameter is an IDBKeyRange, with this, we can narrow the result by defining the bounds of the keys we want to retrieve. The second parameter is the direction the cursor must navigate through the results.

### IDBKeyrange

A key range is a continuous interval over some data type used for keys. A key range can have one of the following situations:

* **lower bounded:** The keys must have a value smaller than the provided lower bound
* **upper bounded:** The keys must have a value larger than the provided upper bound
* **lower and upper bounded:** The keys must have a value between the lower and the upper bound
* **unbounded:** All keys will be valid
* **Single value:**The key must be the provide value

The upper and lower bound can be open, this means the value of the bound won’t be included, or closed, and this means the value of the bound will be included.

## Using a Cursor to Get All the Records

The fetchAllEmployees() method below, is an example of using a cursor to iterate over all the records.

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;

}

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);

}

}

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

The snippet of code performs a cursor search with records that have a last name of “Silver”, then it will perform a check on the emails to see if the number one is in it’s name. If so, it will add the record to a collection and bind to the grid.

function fetchMultiFilterByEmailAndSurname() {

try {

records = [];

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);

}

cursor.continue();

}

else {

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) {

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).

# 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.

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).

# No Cross Browser Sharing (even same origin)

If a database is created (and populated) using Chrome and the same URL is opened in IE10, even though both origins are the same – a database is created for each browser and they don’t share any data.

# Browser Support

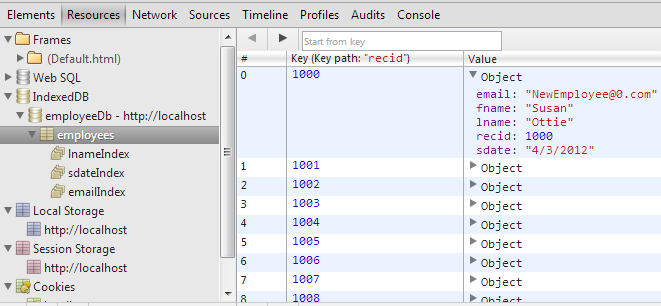
|  |
| --- |
|  |
| IE | Firefox | Chrome | Safari | Opera | iOS Safari | Chrome for Android | IE Mobile |
| 26 versions back |  |  | 4.0: Not supported |  |  |  |  |
| 25 versions back |  |  | 5.0: Not supported |  |  |  |  |
| 24 versions back |  | 2.0: Not supported | 6.0: Not supported |  |  |  |  |
| 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

You can view a database s object stores, indexes, key paths and data within the Developer Tools option in Chrome (Ctrl+Shift+i).



# Code

# 

# 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>