IndexedDB

# Overview

IndexedDB lets you store and retrieve objects that are indexed with a "key."All changes that you make to the database happen within transactions. Like most web storage solutions, **IndexedDB follows a same-origin policy.** So while you can access stored data within a domain, you cannot access data across different domains.

IndexedDB is an **asynchronous** API that can be used in most contexts, including WebWorkers.

While both IndexedDB and WebSQL are solutions for storage, they do not offer the same functionalities. WebSQL Database is a relational database access system, whereas IndexedDB is an indexed table system.

## Definitions

This section defines and explains terms used in the IndexedDB API.

### Database

#### Object store

The mechanism by which data is stored in the database. The object store persistently holds records, which are key-value pairs. Records within an object store are sorted according to the keys in an ascending order.

Every object store must have a name that is unique within its database. The object store can optionally have a [key generator](#_Key_generator) and a [key path](#_Key_Path). If the object store has a key path, it is using [in-line keys](#_In-line_key); otherwise, it is using [out-of-line keys](#_Out-of-line_keys).

For the reference documentation on object store, see IDBObjectStore.

#### Version

When a database is first created, its version is the integer 1. Each database has one version at a time; a database can't exist in multiple versions at once. The only way to change the version is by opening it with a greater version than the current one. This will start a versionchange transaction and fire an upgradeneeded event. *The only place where the schema of the database can be updated is inside the handler of that event.*

#### Database connection

An operation created by opening a database. A given database can have multiple connections at the same time.

#### Transaction

An atomic set of data-access and data-modification operations on a particular database. It is how you interact with the data in a database. In fact, any reading or changing of data in the database must happen in a transaction.

A database connection can have several active transaction associated with it at a time, so long as the writing transactions do not have overlapping scopes. The scope of transactions, which is defined at creation, determines which object stores the transaction can interact with and remains constant for the lifetime of the transaction.

Transactions are expected to be short-lived, so the browser can terminate a transaction that takes too long, in order to free up storage resources that the long-running transaction has locked. You can abort the transaction, which rolls back the changes made to the database in the transaction. And you don't even have to wait for the transaction to start or be active to abort it.

The three modes of transactions are: readwrite, readonly, and versionchange. The only way to create and delete object stores and indexes is by using a versionchange transaction.

To learn more about transactions, see IDBTransaction.

#### Index

An index is a specialized object store for looking up records in another object store, called the *referenced object store*.The index is a persistent key-value storage where the value part of its records is the key part of a record in the referenced object store.

The records in an index are automatically populated whenever records in the referenced object store are inserted, updated, or deleted. Each record in an index can point to only one record in its referenced object store, but several indexes can reference the same object store. When the object store changes, all indexes that refer to the object store are automatically updated.

For the reference documentation on index, see IDBKeyRange.

### Key and Value

#### Key

A data value by which stored values are organized and retrieved in the object store. The object store can derive the key from one of three sources: a [key generator](#_Key_generator), a [key path](#_Key_Path), or an explicitly specified value.

The key must be of a data type that has a number that is greater than the one before it. Each record in an object store must have a key that is unique within the same store, so you cannot have multiple records with the same key in a given object store.

A key can be one of the following types: **string**, **date**, **float**, and **array**. For arrays, the key can range from an empty value to infinity. And you can include an array within an array. There is also no requirement to only use keys of type string or integer .

Alternatively, you can also look up records in an object store using the index.

#### Key generator

A mechanism for producing new keys in an ordered sequence. If an object store does not have a key generator, then the application must provide keys for records being stored. Generators are not shared between stores. This is more a browser implementation detail, because in web development, you don't really create or access key generators.

#### Key Path

Defines where the browser should extract the key from in the object store or index. A valid key path can include one of the following: an empty string, a JavaScript identifier, or multiple JavaScript identifiers separated by periods or an array containing any of those. It cannot include spaces.

#### In-line keys

A key that is stored as part of the stored value. It is found using a key path. An in-line key can be generated using a generator. After the key has been generated, it can then be stored in the value using the key path or it can also be used as a key.

#### Out-of-line keys

A key that is stored separately from the value being stored.

### Range and Scope

#### Scope

*The set of object stores and indexes to which a transaction applies.* The scopes of read-only transactions can overlap and execute at the same time. On the other hand, the scopes of writing transactions cannot overlap. You can still start several transactions with the same scope at the same time, but they just queue up and execute one after another.

#### Cursor

A mechanism for iterating over multiple records with a key range. The cursor has a source that indicates which index or object store it is iterating. It has a position within the range, and moves in a direction that is increasing or decreasing in the order of record keys. For the reference documentation on cursors, see IDBCursor.

#### Key Range

A continuous interval over some data type used for keys. Records can be retrieved from object stores and indexes using keys or a range of keys. You can limit or filter the range using lower and upper bounds.

## Concepts

* **IndexedDB databases store key-value pairs**. The values can be complex structured objects, and keys can be properties of those objects. You can create indexes that use any property of the objects for quick searching, as well as sorted enumeration.
* **IndexedDB is built on a transactional database model**.Everything you do in IndexedDB always happens in the context of a transaction.Transactions have a well-defined lifetime, so attempting to use a transaction after it has completed throws exceptions. Also, transactions auto-commit and cannot be committed manually.
* **The IndexedDB API is mostly asynchronous**. The API doesn't give you data by returning values; instead, you have to pass a callback function.Instead, you "request" that a database operation happens. You get notified by a DOM event when the operation finishes, and the type of event you get lets you know if the operation succeeded or failed. It’s similar to XMLHttpRequest.
* **IndexedDB uses a lot of requests**. Requests are objects that receive the success or failure DOM events that were mentioned previously. They have onsuccess and onerror properties, and you can call addEventListener() and removeEventListener()on them. They also have readyState, result, and errorCode properties that tell you the status of the request. The result property can be many different things, depending on how the request was generated (for example, an IDBCursor instance, or the key for a value that you just inserted into the database).
* **IndexedDB uses DOM events to notify you when results are available**.DOM events always have a type property (in IndexedDB, it is most commonly set to "success" or "error"). DOM events also have a target property that indicates where the event is headed. In most cases, the target of an event is the IDBRequest object that was generated as a result of doing some database operation*. Success events don't bubble up and they can't be canceled. Error events, on the other hand, do bubble, and can be cancelled. This is quite important, as error events abort whatever transactions they're running in, unless they are cancelled.*
* **IndexedDB does not use Structured Query Language (SQL)**.

Full text searching. The API does not have an equivalent of the LIKE operator in SQL.

# Using IndexedDB

The basic pattern that IndexedDB encourages is the following:

1. Open a database
2. Create an object store in the database
3. Start a transaction and make a request to do some database operation, like adding or retrieving data.
4. Wait for the operation to complete by listening to the right kind of DOM event.
5. Do something with the results (which can be found on the request object).

## Createing and structuring the Object Store

### Opening database

We start the whole process like this:

*// open database with databasename:string and databaseversion:number***let** request: IDBOpenDBRequest = ***indexedDB***.open(**"mydatabase"**, 3);

The version of the database determines the database schema — the object stores in the database and their structure.

If the database doesn't already exist, it is created by the open operation, then an onupgradeneeded event is triggered and you create the database schema in the handler for this event.

If the database does exist but you are specifying an upgraded version number, an onupgradeneeded event is triggered straight away, allowing you to provide an updated schema in its handler.

### Generating Handlers

Add success and error handlers:

**var** db: IDBDatabase;  
**var** request:IDBOpenDBRequest = ***indexedDB***.open(**"MyTestDatabase"**);  
request.**onerror** = **function**(event) {  
*// Generic error handler for all errors targeted at this database's requests!  
alert*(**"Database error: "** + (<IDBOpenDBRequest>event.**target**).**error**.**name**);};  
request.**onsuccess** = **function**(event) {  
db = (<IDBOpenDBRequest>event.**target**).**result**;  
};

One of the common possible errors when opening a database is VER\_ERR. It indicates that the version of the database stored on the disk is greater than the version that you are trying to open. This is an error case that must always be handled by the error handler.

When you create a new database or increase the version number of an existing database, the onupgradeneeded event will be triggered. In the handler for the upgradeneeded event, you should create the object stores needed for this version of the database:

request.**onupgradeneeded** = (event:IDBVersionChangeEvent) =>{  
**this**.**db** = (<IDBOpenDBRequest>event.**target**).**result**;  
**let** params: IDBObjectStoreParameters = {**keyPath**: **"mykey"**, **autoIncrement**: **false**};  
**var** objectStore:IDBObjectStore = **this**.**db**.createObjectStore(**"storename"**, params);  
}

Object stores are created with a single call to createObjectStore().The method takes a name of the store, and a parameter of type IDBObjectStoreParameters. the parameter object is optional.

In our case, we've asked for an object store named "customers" and defined a keyPath, which is the property that makes an individual object in the store unique. That property in this example is "ssn" since a social security number is guaranteed to be unique. "ssn" must be present on every object that is stored in the objectStore.

If you need to change an existing object store (e.g., to change the keyPath), then you must delete the old object store and create it again with the new options.(Note that this will delete the information in the object store! If you need to save that information, you should read it out and save it somewhere else before upgrading the database.)

Trying to create an object store with a name that already exists (or trying to delete an object store with a name that does not already exist) will throw an error.

If the onupgradeneeded event exits successfully, the onsuccess handler of the open database request will then be triggered.

### Structuring the database

A single database can contain any number of object stores.Whenever a value is stored in an object store, it is associated with a key. There are several different ways that a key can be supplied depending on whether the object store uses a key path or a key generator.

The following table shows the different ways the keys are supplied:

| **Key Path (keyPath)** | **Key Generator (autoIncrement)** | **Description** |
| --- | --- | --- |
| No | No | This object store can hold any kind of value, even primitive values like numbers and strings. You must supply a separate key argument whenever you want to add a new value. |
| Yes | No | This object store can only hold JavaScript objects. The objects must have a property with the same name as the key path. |
| No | Yes | This object store can hold any kind of value. The key is generated for you automatically, or you can supply a separate key argument if you want to use a specific key. |
| Yes | Yes | This object store can only hold JavaScript objects. Usually a key is generated and the value of the generated key is stored in the object in a property with the same name as the key path. However, if such a property already exists, the value of that property is used as key rather than generating a new key. |

You can also create indices on any object store, provided the object store holds objects, not primitives. An index lets you look up the values stored in an object store using the value of a property of the stored object, rather than the object's key.

request.**onupgradeneeded** = **function**(event) {  
**var** db = (<IDBOpenDBRequest>event.**target**).**result**;  
*// Create an objectStore to hold information about our customers. We're  
 // going to use "ssn" as our key path because it's guaranteed to be  
 // unique - or at least that's what I was told during the kickoff*

*// meeting.***var** objectStore = db.createObjectStore(**"customers"**, { **keyPath**: **"ssn"** });  
  
*// Create an index to search customers by name (keyPath). We may have*

*// duplicates so we can't use a unique index.*objectStore.createIndex(**"name"**, **"name"**,{**unique**: **false,multiEntry**: **false**});  
  
*// Create an index to search customers by email. We want to ensure that  
// no two customers have the same email, so use a unique index.*objectStore.createIndex(**"email"**, **"email"**, { **unique**: **true** });  
  
*// Use transaction oncomplete to make sure the objectStore creation is  
// finished before adding data into it.*objectStore.**transaction**.**oncomplete** = **function**(event) {

*// Store values in the newly created objectStore.*

**var** transaction = db.transaction([**"customers"**], ***IDBTransaction***.**READ\_WRITE**);**var** customerObjectStore = db.transaction(**"customers"**,**"readwrite"**)

.objectStore(**"customers"**);  
**for** (**var** i **in** customerData) {  
 customerObjectStore.add(customerData[i]);  
}  
 };  
};

## Using a key generator

Setting up an autoIncrement flag when creating the object store would enable the key generator for that object store. By default this flag is not set.

The current number of a key generator is always set to 1 when the object store for that key generator is first created.

Basically the newly auto-generated key is increased by 1 based on the previous key. The current number for a key generator never decreases, other than as a result of database operations being reverted, for example, the database transaction is aborted. Therefore deleting a record or even clearing all records from an object store never affects the object store's key generator.

## Transaction

Before you can do anything with your new database, you need to start a transaction. Transactions come from the database object, and you have to specify which object stores you want the transaction to span.

Next, you need to decide if you're going to make changes to the database or if you just need to read from it. Transactions have three available modes: readonly, readwrite, and versionchange.

Transactions are started when the transaction is created, not when the first request is placed.

**this**.**db**.createObjectStore(**"customers"**, {**keyPath**: **'ssn'**});

**var** store1 =

**this**.**db**.transaction(**"customers"**, **"readwrite"**).objectStore(**"customers"**);  
**var** store2 =

**this**.**db**.transaction(**"customers"**, **"readwrite"**).objectStore(**"customers"**);  
store2.put({**ssn**:**"666-66-6666"**, **name**:**"kevin"**, **age**:40, **email**:**"kevin@home.org"**});  
store1.put({**ssn**:**"666-66-6666"**, **name**:**"John"**, **age**:30, **email**:**"john@home.org"**});

After the code is executed the object store should contain the value with name “kevin”, since store2 should run after store1.

You open such transactions with transaction():IDBTransaction opration. The method accepts two parameters: the **storeNames** (the scope, defined as an array of object stores that you want to access) and the **mode** (readonly or readwrite) for the transaction. IDBTransaction has objectStore() method, which you can use to access your object store.

By default, where no mode is specified, transactions open in readonly mode.

### Lifetime of transaction

Transactions are tied very closely to the event loop. If you make a transaction and return to the event loop without using it then the transaction will become inactive. The only way to keep the transaction active is to make a request on it.

When the request is finished you'll get a DOM event and, assuming that the request succeeded, you'll have another opportunity to extend the transaction during that callback. If you return to the event loop without extending the transaction then it will become inactive, and so on. As long as there are pending requests the transaction remains active. Transaction lifetimes are really very simple but it might take a little time to get used to. A few more examples will help, too. If you start seeing TRANSACTION\_INACTIVE\_ERR error codes then you've messed something up.

### Event of transaction

Transactions can receive DOM events of three different types: error, abort, and complete.

A error events will bubble, so a transaction receives error events from any requests that are generated from it. A more subtle point here is that the default behavior of an error is to abort the transaction in which it occurred. Unless you handle the error by first calling stopPropagation() on the error event then doing something else, the entire transaction is rolled back.

If you don't handle an error event or if you call abort() on the transaction, then the transaction is rolled back and an abort event is fired on the transaction.

Otherwise, after all pending requests have completed, you'll get a complete event. If you're doing lots of database operations, then tracking the transaction rather than individual requests can certainly aid your sanity.

## Adding, retrieving, and removing data

### Adding data to database

**var objectStore** = transaction.objectStore(**"customers"**);  
 **for** (**var i in customerData**) {  
 **var request** = objectStore.add(customerData[i]);  
 **request**.*onsuccess* = **function**(event) {

*// result === customerData.ssn*  
 **var** result = (<IDBOpenDBRequest>event.**target**).**result**; *// event.target.result == customerData[i].ssn;*

*}* };  
}

The result of a request generated from a call to add() is the key of the value that was added. So in this case, it should equal the ssn property of the object that was added, since the object store uses the ssn property for the key path.

Note that the **add()** function requires that no object already be in the database with the same key. If you're trying to modify an existing entry, or you don't care if one exists already, you can use the **put()** function.

### Removing data from database

**var request** = db.transaction([**"customers"**], **"readwrite"**)  
 .objectStore(**"customers"**)  
 .delete(**"444-44-4444"**);  
 **request**.*onsuccess* = **function**(event) {  
 *// It's gone!*};

### Getting data from database

#### Using get

**this**.**db**.**transaction**(**"customers"**).objectStore(**"customers"**).get(**"444-44-4444"**).**onsuccess** =  
 **function** (event) {  
 ***console***.log(**"Name for SSN 444-44-4444 is "** + (<IDBRequest>event.**target**).**result**.**name**);  
 };

#### Using cursor

**var** objectStore = **this**.**db**.**transaction**(**"customers"**).objectStore(**"customers"**);  
objectStore.openCursor().**onsuccess** = **function**(event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *alert*(**"Name for SSN "** + cursor.key + **" is "** + cursor.value.**name**);  
 cursor.continue();  
 }  
 **else** {  
 *alert*(**"No more entries!"**);  
 }  
};

Sometimes you may want to iterate in descending order rather than in ascending order (the default direction for all cursors). Switching direction is accomplished by passing **prev** to the openCursor() function:

objectStore.openCursor(**null**, ***IDBCursor***.prev).onsuccess = **function**(event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *// Do something with the entries.* cursor.continue();  
 }  
};

#### Benutzung eines Index

If you need to look up a customer by name, however, you'll need to iterate over every SSN in the database until you find the right one. Searching in this fashion would be very slow, so instead you can use an index.

**var** index = objectStore.index(**"name"**);  
index.get(**"Donna"**).**onsuccess** = **function** (event) {  
 ***console***.log(**"Donna's SSN is "** + event.**target**.**result**.**ssn**);  
};

The "name" cursor isn't unique, so there could be more than one entry with the name set to "Donna". In that case you always get the one with the **lowest** key value.

If you need to access all the entries with a given name you can use a cursor. You can open two different types of cursors on indexes. A normal cursor maps the index property to the object in the object store. A key cursor maps the index property to the key used to store the object in the object store. The differences are illustrated here:

index.openCursor().onsuccess = **function**(event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *// cursor.key is a name, like "Bill", and cursor.value is the whole object.  
 alert*(**"Name: "** + cursor.key + **", SSN: "** + cursor.value.**ssn** + **", email: "** + cursor.value.**email**);  
 cursor.continue();  
 }  
};  
  
index.openKeyCursor().onsuccess = **function**(event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *// cursor.key is a name, like "Bill", and cursor.value is the SSN.  
 // No way to directly get the rest of the stored object.  
 alert*(**"Name: "** + cursor.key + **", SSN: "** + cursor.value);  
 cursor.continue();  
 }  
};

If you would like to limit the range of values you see in a cursor, you can use a key range object and pass it as the first argument to openCursor() or openKeyCursor(). You can make a key range that only allows a single key, or one the has a lower or upper bound, or one that has both a lower and upper bound. The bound may be "closed" (i.e., the key range includes the given value) or "open" (i.e., the key range does not include the given value).

*// Only match "Donna"***var** singleKeyRange = ***IDBKeyRange***.*only*(**"Donna"**);  
  
*// Match anything past "Bill", including "Bill"***var** lowerBoundKeyRange = ***IDBKeyRange***.*lowerBound*(**"Bill"**);  
  
*// Match anything past "Bill", but don't include "Bill"***var** lowerBoundOpenKeyRange = ***IDBKeyRange***.*lowerBound*(**"Bill"**, **true**);  
  
*// Match anything up to, but not including, "Donna"***var** upperBoundOpenKeyRange = ***IDBKeyRange***.*upperBound*(**"Donna"**, **true**);  
  
*//Match anything between "Bill" and "Donna", but not including "Donna"***var** boundKeyRange = ***IDBKeyRange***.*bound*(**"Bill"**, **"Donna"**, **false**, **true**);  
  
index.openCursor(boundKeyRange).onsuccess = **function** (event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *// Do something with the matches.* cursor.continue();  
 }  
};

Since the "name" index isn't unique, there might be multiple entries where name is the same. Note that such a situation cannot occur with object stores since the key must always be unique. If you wish to filter out duplicates during cursor iteration over indexes, you can pass nextunique (or prevunique if you're going backwards) as the direction parameter. When nextunique or prevunique is used, the entry with the lowest key is always the one returned.

index.openKeyCursor(**null**, ***IDBCursor***.nextunique).onsuccess = **function**(event) {  
 **var** cursor = event.**target**.**result**;  
 **if** (cursor) {  
 *// Do something with the entries.* cursor.continue();  
 }  
};

# Version changes while a web app is open in another tab

When your web app 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 app 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 (an onblocked event is fired until they are closed or reloaded). Here's how it works:

**var** openReq = mozIndexedDB.open(**"MyTestDatabase"**, 2);  
  
openReq.**onblocked** = **function**(event) {  
 *// If some other tab is loaded with the database, then it needs to be closed  
 // before we can proceed.  
 alert*(**"Please close all other tabs with this site open!"**);  
};  
  
openReq.**onupgradeneeded** = **function**(event) {  
 *// All other databases have been closed. Set everything up.* db.createObjectStore(*/\* ... \*/*);  
 *useDatabase*(db);  
};  
  
openReq.**onsuccess** = **function**(event) {  
 **var** db = event.**target**.**result**;  
 *useDatabase*(db);  
 **return**;  
};  
  
**function** *useDatabase*(db) {  
 *// Make sure to add a handler to be notified if another page requests a version  
 // change. We must close the database. This allows the other page to upgrade the database.  
 // If you don't do this then the upgrade won't happen until the user closes the tab.* db.onversionchange = **function**(event) {  
 db.close();  
 *alert*(**"A new version of this page is ready. Please reload!"**);  
 };  
  
 *// Do stuff with the database.*}