Working with IndexedDB

- What is IndexedDB? {: #what }
- IndexedDB terms {: #terms }
- Checking for IndexedDB support {: #support }
- Opening a database {: #open }
- Working with object stores {: #stores }
  - Creating object stores
  - Defining primary keys
  - Defining indexes
- Working with data {: #data }
  - Creating data
  - Reading data
  - Updating data
  - Deleting data
- Getting all the data {: #get }
  - Using the getAll() method
  - Using cursors
  - Working with ranges and indexes {: #ranges }
- Using database versioning {: #versioning }
- Further reading {: #resources }
  - IndexedDB Documentation
  - Data storage limits
- Appendix
  - Comparison of IndexedDB API and IndexedDB Promised library

---

This text guides you through the basics of the IndexedDB API. We are using Jake Archibald's IndexedDB Promised library, which is very similar to the IndexedDB API, but uses promises rather than events. This simplifies the API while maintaining its structure, so anything you learn using this library can be applied to the IndexedDB API directly.

What is IndexedDB? {: #what }

IndexedDB is a large-scale, NoSQL storage system. It lets you store just about anything in the user's browser. In addition to the usual search, get, and put actions, IndexedDB also supports transactions. Here is the definition of IndexedDB on MDN:

> IndexedDB is a low-level API for client-side storage of significant amounts of structured data, including files/blobs. This API uses indexes to enable high performance searches of this data. While DOM Storage is useful for storing smaller amounts of data, it is less useful for storing larger amounts of structured data. IndexedDB provides a solution.

Each IndexedDB database is unique to an origin (typically, this is the site domain or subdomain), meaning it cannot access or be accessed by any other origin. Data storage limits are usually quite large, if they exist at all, but different browsers handle limits and data eviction differently. See the Further reading section for more information.

IndexedDB terms {: #terms }

Database : The highest level of IndexedDB. It contains the object stores, which in turn contain the data you would like to persist. You can create multiple databases with whatever names you choose.

Object store : An individual bucket to store data. You can think of object stores as being similar to tables in traditional relational databases. Typically, there is one object store for each *type* (not JavaScript data type) of data you are storing. For example, given an app that persists blog posts and user profiles, you could imagine two object stores. Unlike tables in traditional databases, the actual JavaScript data types of data within the store do not need to be consistent (for example, if there are three people in the `people` object store, their age properties could be `53`, `'twenty-five'`, and `unknown`).

Index : A kind of object store for organizing data in another object store (called the reference object store) by an individual property of the data. The index is used to retrieve records in the object store by this property. For example, if you're storing people, you may want to fetch them later by their name, age, or favorite animal.

Operation : An interaction with the database.

Transaction : A wrapper around an operation, or group of operations, that ensures database integrity. If one of the actions within a transaction fails, none of them are applied and the database returns to the state it was in before the transaction began. All read or write operations in IndexedDB must be part of a transaction. This allows for atomic read-modify-write operations without having to worry about other threads acting on the database at the same time.

Cursor : A mechanism for iterating over multiple records in a database.

Checking for IndexedDB support {: #support }

IndexedDB is almost universally supported, with the one exception being Opera Mini. It's still a good idea to feature-detect support just in case. The easiest way is to check the `window` object:

```
if (!('indexedDB' in window)) {
  console.log("This browser doesn't support IndexedDB");
  return;
}
```

You simply place this function at the beginning of our scripts and you're ready to use IndexedDB.

Opening a database {: #open }

With IndexedDB, you can create multiple databases with any names you choose. To open a database, you use:

```
idb.open(name, version, upgradeCallback);
```

This method returns a promise that resolves to a database object. When using `idb.open`, you provide a name, version number, and an optional callback to set up the database.

Here is an example of `idb.open()` in context:

```
// Check for support.
if (!('indexedDB' in window)) {
  console.log("This browser doesn't support IndexedDB.");
  return;
```

```
}


const dbPromise = idb.open('test-db1', 1);
```

You place the check for IndexedDB support at the top of the anonymous function. This exits out of the function if the browser doesn't support IndexedDB. Then you call `idb.open()` to open a database named `"test-db1"`. In this example, the optional callback function has been left out to keep things simple.

Working with object stores {: #stores }

Creating object stores

A database typically contains one or more object stores. Object stores can be thought of as similar to tables in SQL databases and should contain objects of the same *type* (not JavaScript data type). For example, for a site persisting user profiles and notes, you can imagine a `people` object store containing `person` objects, and a `notes` object store. A well structured IndexedDB database should have one object store for each type of data that needs to be persisted.

To ensure database integrity, object stores can only be created and removed in the callback function in `idb.open()`. The callback receives an instance of `UpgradeDB`, a special object in the IDB Promised library that is used to create object stores. Call the `createObjectStore()` method on `UpgradeDB` to create the object store:

```
upgradeDb.createObjectStore('storeName', options);
```

This method takes the name of the object store as well as a parameter object that lets you define various configuration properties for the object store.

Below is an example of the `createObjectStore()` method:

```
// Check for support.
if (!('indexedDB' in window)) {
  console.log("This browser doesn't support IndexedDB.");
  return;
}


const dbPromise = idb.open('test-db2', 1, function (upgradeDb) {
```

```
  console.log('Creating a new object store.');
  if (!upgradeDb.objectStoreNames.contains('firstOS')) {
    upgradeDb.createObjectStore('firstOS');
  }
});
```

Again, you first check the browser for IndexedDB support. This time you include the callback function in `idb.open()` in order to create the object store. The browser throws an error if you try to create an object store that already exists in the database so you wrap the `createObjectStore()` method in an `if` statement that checks if the object store exists. Inside the `if` block you call `createObjectStore()` on the `UpgradeDB` object to create an object store named `"firstOS"`.

Defining primary keys

When you define object stores, you can define how data is uniquely identified in the store using the primary key. You can define a primary key by either defining a key path, or by using a key generator.

A *key path* is a property that always exists and contains a unique value. For example, in the case of a `people` object store, you could choose the email address as the key path.

```
upgradeDb.createObjectStore('people', { keyPath: 'email' });
```

This example creates an object store called `"people"` and assigns the `email` property as the primary key.

You could also use a key generator, such as `autoIncrement`. The key generator creates a unique value for every object added to the object store. By default, if you don't specify a key, IndexedDB creates a key and stores it separately from the data.

```
upgradeDb.createObjectStore('notes', { autoIncrement: true });
```

This example creates an object store called `"notes"` and sets the primary key to be assigned automatically as an auto incrementing number.

```
upgradeDb.createObjectStore('logs', { keyPath: 'id', autoIncrement: true });
```

This example is similar to the previous example, but this time the auto incrementing value is assigned to a property called `"id"`.

Choosing which method to use to define the key depends on your data. If your data has a property that is always unique, you can make it the keypath to enforce this uniqueness. Otherwise, using an auto incrementing value makes sense.

Let's look at an example:

```
// Check for support.
if (!('indexedDB' in window)) {
  console.log("This browser doesn't support IndexedDB.");
  return;
}

const dbPromise = idb.open('test-db3', 1, function (upgradeDb) {
  if (!upgradeDb.objectStoreNames.contains('people')) {
    upgradeDb.createObjectStore('people', { keyPath: 'email' });
  }
  if (!upgradeDb.objectStoreNames.contains('notes')) {
    upgradeDb.createObjectStore('notes', { autoIncrement: true });
  }
  if (!upgradeDb.objectStoreNames.contains('logs')) {
    upgradeDb.createObjectStore('logs', { keyPath: 'id', autoIncrement: true });
  }
});
```

This code creates three object stores demonstrating the various ways of defining primary keys in object stores.

Defining indexes

Indexes are a kind of object store used to retrieve data from the reference object store by a specified property. An index lives inside the reference object store and contains the same data, but uses the specified property as its key path instead of the reference store's primary key. Indexes must be made when you create your object stores and can also be used to define a unique constraint on your data.

To create an index, call the `createIndex()` method on an object store instance:

```
objectStore.createIndex('indexName', 'property', options);
```

This method creates and returns an index object. The `createIndex()` method takes the name of the new index as the first argument and the second argument refers to the property on the data you want to index. The final argument lets you define two options that determine how the index operates: `unique` and `multiEntry`. If `unique` is set to `true`, the index does not allow duplicate values for a single key. Next, `multiEntry` determines how `createIndex()` behaves when the indexed property is an array. If it's set to `true`, `createIndex()` adds an entry in the index for each array element. Otherwise, it adds a single entry containing the array.

Here is an example:

```
// Check for support.
if (!('indexedDB' in window)) {
  console.log("This browser doesn't support IndexedDB.");
  return;
}

const dbPromise = idb.open('test-db4', 1, function (upgradeDb) {
  if (!upgradeDb.objectStoreNames.contains('people')) {
    const peopleOS = upgradeDb.createObjectStore('people', { keyPath: 'email' });
    peopleOS.createIndex('gender', 'gender', { unique: false });
    peopleOS.createIndex('ssn', 'ssn', { unique: true });
  }
  if (!upgradeDb.objectStoreNames.contains('notes')) {
    const notesOS = upgradeDb.createObjectStore('notes', { autoIncrement: true });
    notesOS.createIndex('title', 'title', { unique: false });
  }
  if (!upgradeDb.objectStoreNames.contains('logs')) {
    const logsOS = upgradeDb.createObjectStore('logs', { keyPath: 'id', autoIncrement: true
});
  }
});
```

In this example, the `"people"` and `"notes"` object stores have indexes. To create the indexes, you first assign the result of `createObjectStore()` (which is an object store object) to a variable so you can call `createIndex()` on it.

Indexes are updated every time you write data to the reference object store. More indexes mean more work for IndexedDB.

Working with data {: #data }

This section describes how to create, read, update, and delete data. These operations are all asynchronous, using promises where the IndexedDB API uses requests. This simplifies the API. Instead of listening for events triggered by the request, you can simply call `.then()` on the database object returned from `idb.open()` to start interactions with the database or `await` its creation.

All data operations in IndexedDB are carried out inside a transaction. Each operation has this form:

1. Get database object
2. Open transaction on database
3. Open object store on transaction
4. Perform operation on object store

A transaction can be thought of as a safe wrapper around an operation or group of operations. If one of the actions within a transaction fails, all of the actions are rolled back. Transactions are specific to one or more object stores, which you define when you open the transaction. They can be read-only or read and write. This signifies whether the operations inside the transaction read the data or make a change to the database.

Creating data

To create data, call the `add()` method on the object store and pass in the data you want to add. The `add()` method has an optional second argument that lets you define the primary key for the individual object on creation, but it should only be used if you have not specified the key path in `createObjectStore()`. Here is a simple example:

```
someObjectStore.add(data, optionalKey);
```

The data parameter can be data of any type: a string, number, object, array, and so forth. The only restriction is if the object store has a defined keypath, the data must contain this property and the value must be unique. The `add()` method returns a promise that resolves once the object has been added to the store.

Each `add()` call occurs within a transaction, so even if the promise resolves successfully, it doesn't necessarily mean the operation worked. Remember, if one of the actions in the transaction fails, all of the operations in the transaction are rolled back. To be sure that the add operation was carried out, you need to check if the whole transaction has completed using the `transaction.complete()` method. This is a promise that resolves when the transaction completes and rejects if the transaction errors. Note that this method doesn't actually close the transaction. The transaction completes on its own. You must perform this check for all "write" operations, because it is your only way of knowing that the changes to the database have actually been carried out.

Let's look at an example of the `add` method:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readwrite');
    const store = tx.objectStore('store');
    const item = {
      name: 'sandwich',
      price: 4.99,
```

```
      description: 'A very tasty sandwich',
      created: new Date().getTime(),
    };
    store.add(item);
    return tx.complete;
  })
  .then(function () {
    console.log('Added item to the store!');
  });
```

First, you get the database object. You call `.then()` on `dbPromise`, which resolves with the database object, and pass this object to the callback function in `.then()`. Because `dbPromise` (from `idb.open()`) is a promise, you can safely assume that when `.then()` executes, the database is open and all object stores and indexes are ready for use.

The next step is to open a transaction by calling the `transaction()` method on the database object. This method takes a list of object stores, which defines the scope of the transaction (if it is a single store name, you can pass it directly instead of passing an array of names, as you did in the example where you only wanted the `"store"` object store). The `transaction()` method also has an optional second argument for the mode, which can be `readonly` or `readwrite`. This option is `read-only` by default.

You can then open the `"store"` object store on this transaction and assign it to the `store` variable. Now when you call `store.add()`, the add operation occurs within the transaction. Finally, you return `tx.complete()` and log a success message once the transaction has completed.

Reading data

To read data, call the `get()` method on the object store. The `get()` method takes the primary key of the object you want to retrieve from the store. Here is a basic example:

```
someObjectStore.get(primaryKey);
```

As with `add()`, the `get()` method returns a promise and must happen within a transaction.

Let's look at an example of the `get()` method:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readonly');
    const store = tx.objectStore('store');
    return store.get('sandwich');
```

```
  })
  .then(function (val) {
    console.dir(val);
  });
```

Once again, you start the operation by getting the database object and creating a transaction. Note that this time it is a read-only transaction because you are not writing anything to the database inside the transaction (that is, using `put()`, `add()`, or `delete()`). You then open the object store on the transaction and assign the resulting object store object to the `store` variable. Finally, you return the result of `store.get()` and log this object to the console.

If you try to get an object that doesn't exist, the success handler still executes, but the result is `undefined`.

Updating data

To update data, call the `put()` method on the object store. The `put()` method is very similar to the `add()` method and can be used instead of `add()` to create data in the object store. Like `add()`, `put()` takes the data and an optional primary key:

```
someObjectStore.put(data, optionalKey);
```

Again, this method returns a promise and occurs inside a transaction. As with `add()`, you need to be careful to check `transaction.complete()` if you want to be sure that the operation was actually carried out.

Here is an example using the `put` method:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readwrite');
    const store = tx.objectStore('store');
    const item = {
      name: 'sandwich',
      price: 99.99,
      description: 'A very tasty, but quite expensive, sandwich',
      created: new Date().getTime(),
    };
    store.put(item);
    return tx.complete;
  })
  .then(function () {
```

```
    console.log('Item updated!');
  });
```

To update an existing item in the object store, use the `put()` method on an object containing the same primary key value as the object in the store. Here, the assumption is that the `keyPath` for the store object store is the `"name"` property and that you are updating the price and description of the `"sandwich"` object. The database interaction has the same structure as the create and read operations: get the database object, create a transaction, open an object store on the transaction, and perform the operation on the object store.

Deleting data

To delete data, call the `delete()` method on the object store.

```
someObjectStore.delete(primaryKey);
```

Once again, this method returns a promise and must be wrapped in a transaction. Here is a simple example:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readwrite');
    const store = tx.objectStore('store');
    store.delete(key);
    return tx.complete;
  })
  .then(function () {
    console.log('Item deleted.');
  });
```

The structure of the database interaction is the same as for the other operations. Note that you again check that the whole transaction has completed by returning the `tx.complete()` method to be sure that the deletion was carried out.

Getting all the data {: #get }

So far you have only retrieved objects from the store one at a time. You can also retrieve all of the data (or a subset) from an object store or index using either the `getAll()` method or using cursors.

Using the getAll() method

The simplest way to retrieve all of the data is to call the `getAll()` method on the object store or index, like this:

```
someObjectStore.getAll(optionalConstraint);
```

This method returns all the objects in the object store matching the specified key or key range (see Working with ranges and indexes), or all objects in the store if no parameter is given. As with all other database operations, this operation happens inside a transaction. Here is a short example:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readonly');
    const store = tx.objectStore('store');
    return store.getAll();
  })
  .then(function (items) {
    console.log('Items by name:', items);
  });
```

Here you are calling `getAll()` on the `"store"` object store. This returns all of the objects in the store ordered by the primary key.

Using cursors

Another way to retrieve all of the data is to use a cursor. A cursor selects each object in an object store or index one by one, letting you do something with the data as it is selected. Cursors, like the other database operations, work within transactions.

You create the cursor by calling the `openCursor()` method on the object store, like this:

```
someObjectStore.openCursor(optionalKeyRange, optionalDirection);
```

This method returns a promise that resolves with a `cursor` object representing the first object in the object store or `undefined` if there is no object. To move on to the next object in the object store, you call `cursor.continue()`. This returns a promise that resolves with the next object, or `undefined` if there are no more objects. You put this inside a loop to move through all of the entries in the store one by one. The optional key range in the `openCursor()` method limits the iteration to a subset of the objects in the store. The direction option can be `next` or `prev`, specifying forward or backward traversal through the data.

The next example uses a cursor to iterate through all the items in the "store" object store and log them to the console:

```
dbPromise
  .then(function (db) {
    const tx = db.transaction('store', 'readonly');
    const store = tx.objectStore('store');
    return store.openCursor();
  })
  .then(function logItems(cursor) {
    if (!cursor) {
      return;
    }
    console.log('Cursored at:', cursor.key);
    for (const field in cursor.value) {
      console.log(cursor.value[field]);
    }
    return cursor.continue().then(logItems);
  })
  .then(function () {
    console.log('Done cursoring.');
  });
```

As usual, you start by getting the database object, creating a transaction, and opening an object store. You call the `openCursor()` method on the object store and pass the cursor object to the callback function in `.then()`. This time you name the callback function `"logItems"` so you can call it from inside the function and make a loop. The line `if (!cursor) {return;}` breaks the loop if the promise returned by `store.openCursor()` resolves with `undefined`, or if the promise returned by `cursor.continue()`resolves with `undefined` (indicating that there are no more objects).

The cursor object contains a `key` property that represents the primary key for the item. It also contains a `value` property that represents the data. At the end of `logItems()`, you return `cursor.continue().then(logItems)`. The `cursor.continue()` method returns a promise that resolves to a cursor object representing the next item in the store or `undefined` if there are no more objects. This result is passed to the callback function in `.then()`, which you have chosen to be `logItems()`, so that the function loops. Consequently, `logItems()` continues to call itself until no objects remain.

Working with ranges and indexes {: #ranges }

You can get all the data in a couple of different ways, but what if you want only a subset of the data based on a particular property? This is where indexes come in. Indexes let you fetch the data in an object store by a property other than the primary key. You can create an index on any property (which becomes the keypath for the index), specify a range on that property, and get the data within the range using the `getAll()` method or a cursor.

You define the range using the `IDBKeyRange` object. This object has five methods that are used to define the limits of the range:

- `upperBound()`
- `lowerBound()`
- `bound()` (which means both)
- `only()`
- `includes()`

As expected, the `upperBound()` and `lowerBound()` methods specify the upper and lower limits of the range.

```
IDBKeyRange.lowerBound(indexKey);
```

Or

```
IDBKeyRange.upperBound(indexKey);
```

They each take one argument which is the index's keypath value of the item you want to specify as the upper or lower limit.

The `bound()` method is used to specify both an upper and lower limit, and takes the lower limit as the first argument:

```
IDBKeyRange.bound(lowerIndexKey, upperIndexKey);
```

The range for these functions is inclusive by default, but can be specified as exclusive by passing `true` as the second argument (or the third and fourth in the case of `bound()`, for the lower and upper limits respectively). An inclusive range includes the data at the limits of the range. An exclusive range does not.

Let's look at an example. For this demo, you have created an index on the `"price"` property in the `"store"` object store. You have also added a small form with two inputs for the upper and lower limits of the range. Imagine you are passing in the lower and upper bounds to the function as floating point numbers representing prices:

```
function searchItems(lower, upper) {
  if (lower === '' && upper === '') {
    return;
```

```
  }

  let range;
  if (lower !== '' && upper !== '') {
    range = IDBKeyRange.bound(lower, upper);
  } else if (lower === '') {
    range = IDBKeyRange.upperBound(upper);
  } else {
    range = IDBKeyRange.lowerBound(lower);
  }

  dbPromise
    .then(function (db) {
      const tx = db.transaction(['store'], 'readonly');
      const store = tx.objectStore('store');
      const index = store.index('price');
      return index.openCursor(range);
    })
    .then(function showRange(cursor) {
      if (!cursor) {
        return;
      }
      console.log('Cursored at:', cursor.key);
      for (const field in cursor.value) {
        console.log(cursor.value[field]);
      }
      return cursor.continue().then(showRange);
    })
    .then(function () {
      console.log('Done cursoring.');
    });
}
```

The code first gets the values for the limits and checks if the limits exist. The next block of code decides which method to use to limit the range based on the values. In the database interaction, you open the object store on the transaction as usual, then you open the `"price"` index on the object store. The `"price"` index allows you to search for the items by price. You open a cursor on the index and pass in the range. The cursor now returns a promise representing the first object in the range, or `undefined` if there is no data within the range. The `cursor.continue()` method returns a cursor representing the next object and so on through the loop until you reach the end of the range.

Using database versioning {: #versioning }

When you call `idb.open()`, you can specify the database version number in the second parameter. If this version number is greater than the version of the existing database, the upgrade callback executes, allowing you to add object stores and indexes to the database.

The browser throws an error if you try to create object stores or indexes that already exist in the database. You can wrap the calls to `createObjectStore` in `if` statements checking if the object store already exists using `upgradeDb.objectStoreNames.contains('objectStoreName')`. You can also use a `switch` statement on the `oldVersion` property as in the next example.

The `UpgradeDB` object has a special `oldVersion` property, which indicates the version number of the database existing in the browser. You can pass this version number into a `switch` statement to execute blocks of code inside the upgrade callback based on the existing database version number. Here's an example:

```
const dbPromise = idb.open('test-db7', 2, function (upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      upgradeDb.createObjectStore('store', { keyPath: 'name' });
    case 1:
      const peopleStore = upgradeDb.transaction.objectStore('store');
      peopleStore.createIndex('price', 'price');
  }
});
```

In the example you have set the newest version of the database at 2. When this code first executes, and since the database doesn't yet exist in the browser, `upgradeDb.oldVersion` is 0 and the `switch` statement starts at `case 0`. In the example, this results in a `"store"` object store being added to the database. Usually, in `switch` statements, there is a `break` after each case, but this is deliberately not used here. This way, if the existing database is a few versions behind (or if it doesn't exist), the code continues through the rest of the `case` blocks until it has executed all the latest changes. So in the example, the browser continues executing through `case 1`, creating a `"price"` index on the `"store"` object store. Once this has finished executing, the database in the browser is at version 2 and contains a `"store"` object store with a `"price"` index.

Let's say you now want to create a `"description"` index on the `"store"` object store. You need to update the version number and add a case, like this:

```
const dbPromise = idb.open('test-db7', 3, function (upgradeDb) {
  switch (upgradeDb.oldVersion) {
    case 0:
      upgradeDb.createObjectStore('store', { keyPath: 'name' });
    case 1:
      const storeOS = upgradeDb.transaction.objectStore('store');
      storeOS.createIndex('price', 'price');
    case 2:
      const storeOS = upgradeDb.transaction.objectStore('store');
      storeOS.createIndex('description', 'description');
  }
});
```

Assuming the database you created in the previous example still exists in the browser, when this executes `upgradeDb.oldVersion` is 2. The `case 0` and `case 1` are skipped, and the browser executes the code in `case 2`, which creates a `"description"` index. Once all this has finished, the browser has a database at version 3 containing a `"store"` object store with `"price"` and `"description"` indexes.

Further reading {: #resources }

IndexedDB Documentation

- Using IndexedDB - MDN
- Basic Concepts Behind IndexedDB - MDN
- Indexed Database API - W3C

Data storage limits

- Storage for the web
- Browser storage limits and eviction criteria

Appendix

Comparison of IndexedDB API and IndexedDB Promised library

The IndexedDB Promised library sits on top of the IndexedDB API, translating its requests into promises. The overall structure is the same between the library and the API and, in general, the actual syntax for the database operations is the same and they will act the same way. But there are a few differences because of the differences between requests and promises, which are covered here.

All database interactions in the IndexedDB API are requests and have associated `onsuccess` and `onerror` event handlers. These are similar to the `.then()` and `.catch()` promise functions. The `indexedDB.open()` method in the raw API also gets a special event handler, `onupgradeneeded`, which is used to create the object stores and indexes. This is equivalent to the upgrade callback in `idb.open()` in the Promised library. In fact, if you look through the Promised library, you will find the upgrade callback is just a convenient wrapper for the `onupgradeneeded` event handler.

Here's an example of the IndexedDB API. This example opens a database, adds an object store, and adds one item to the object store:

```
let db;


const openRequest = indexedDB.open('test_db', 1);


openRequest.onupgradeneeded = function (e) {
  db = e.target.result;
  console.log('running onupgradeneeded');
  if (!db.objectStoreNames.contains('store')) {
    const storeOS = db.createObjectStore('store', { keyPath: 'name' });
  }
};
openRequest.onsuccess = function (e) {
  console.log('running onsuccess');
  db = e.target.result;
  addItem();
};
openRequest.onerror = function (e) {
  console.log('onerror!');
  console.dir(e);
};
```

```
function addItem() {
  const transaction = db.transaction(['store'], 'readwrite');
  const store = transaction.objectStore('store');
  const item = {
    name: 'banana',
    price: '$2.99',
    description: 'It is a purple banana!',
    created: new Date().getTime(),
  };

  const request = store.add(item);

  request.onerror = function (e) {
    console.log('Error', e.target.error.name);
  };
  request.onsuccess = function (e) {
    console.log('Woot! Did it');
  };
}
```

This code does something very similar to previous examples in this tutorial except that it doesn't use the Promised library. You can see that the structure of the database interaction hasn't changed. Object stores are created on the database object in the upgrade event handler, and items are added to the object store in the same transaction sequence you've seen before. The difference is that this is done with requests and event handlers rather than promises and promise chains.

Here is a short reference of the differences between the IndexedDB API and the IndexedDB Promised library.

Action	IndexedDB Promised	IndexedDB API
Open database	`idb.open(name, version, upgradeCallback)`	`indexedDB.open(name, version)`
Upgrade database	Inside upgradeCallback	`request.onupgradeneeded`
Success	`.then`	`request.onsuccess`
Error	`.catch`	`request.onerror`

□□□ #3
□ □□□ □□□ 2026-10-31 21:58:15 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:20 UTC