JavaScript IndexedDB

Summary: in this tutorial, you'll learn about the IndexedDB and how to use it to persistently store data inside the browser.

What is indexedDB

IndexedDB is a **large-scale object store** built into the web browser.

IndexedDB allows you to persistently store the data using key-value pairs.

The values can be any JavaScript type including boolean, number, string, undefined, null, date, object, array, regex, blob, and files.

Why indexedDB

IndexedDB allows you to create web applications that can work both online and offline.

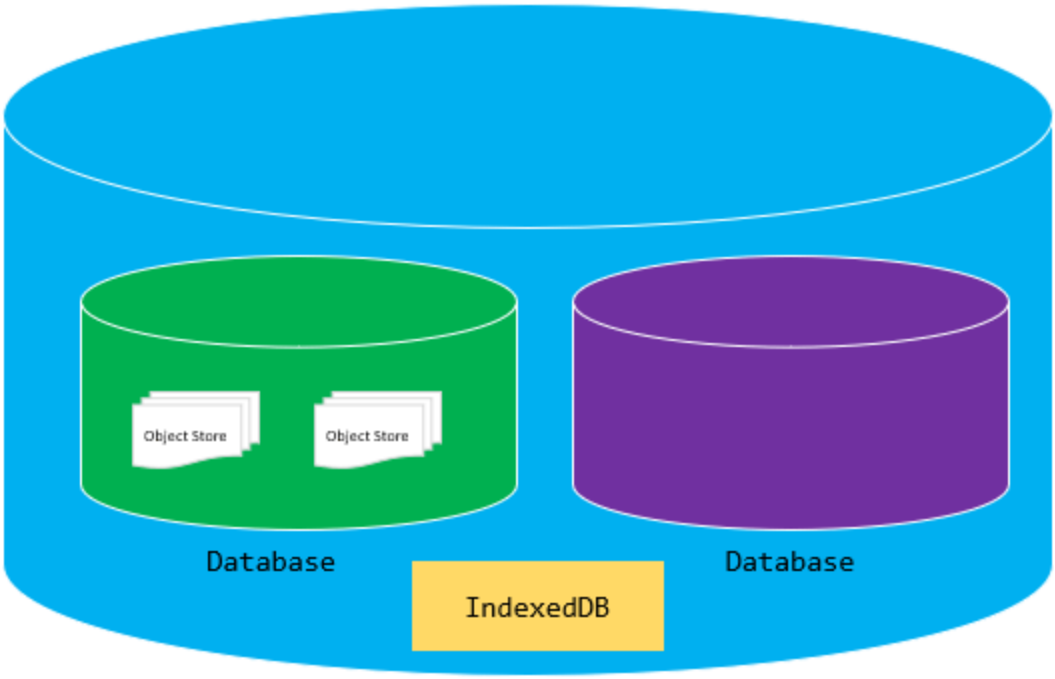
It's useful for applications that store a large amount of data and don't need a constant internet connection.

For example, Google Docs uses the IndexedDB to store the cached documents in the browser and synchronizes with the server periodically. This improves performance and enhances user experiences.

You'll also find other types of applications that heavily use IndexedDB, such as online notepads, quizzes, to-do lists, code sandboxes, and CMS.

IndexedDB structure

The following picture illustrates the structure of the IndexedDB:



Databases

A database is the highest level of IndexedDB, which contains one or more object stores.

The IndexedDB can have one or more databases. Generally, you'll create one database per web application.

Object stores

An object store is a bucket for storing the data and associated indexes. It's conceptually equivalent to the tables in SQL databases.

An object store contains the records stored as key-value pairs.

Indexes

Indexes allow you to query data by the properties of the objects.

Technically, you create indexes on object stores, which are called parent object stores.

For example, if you store the contact information, you may want to create indexes on email, first name, and last name to query the contacts by these properties.

Basic IndexedDB concepts

The following briefly introduces the basic concepts in the IndexedDB:

1) IndexedDB databases store key-value pairs

Unlike localStorage and sessionStorage, the values stored in the IndexedDB can be complex structures like objects and blobs.

Keys can be the properties of these objects or binary objects.

For quick searching and sorting, you can create indexes using any property of the objects.

2) IndexedDB is transactional

Every read from and write to the IndexedDB databases happens in a transaction.

The transactional model ensures data integrity in case users open the web application in multiple tabs/windows and perform the read from and write to the same database.

3) IndexedDB API is mostly asynchronous

IndexedDB operations are asynchronous. They use DOM events to notify you when an operation completes and the result is available.

4) IndexedDB is a NoSQL system

The IndexedDB is a NoSQL system, meaning it uses queries that return a cursor, which then you can use to iterate through the result set.

5) IndexedDB follows the same-origin policy

An origin consists of the domain, protocol, and port of a URL where the code executes. For example `https://www.javascripttutorial.net` :

- **domain**: javascripttutorial.net
- **protocol**: https
- **port**: 443

The `https://www.javascripttutorial.net/dom/` and `https://www.javascripttutorial.net/` are the same origin because they have the same domain, protocol, and port.

However, the `http://www.javascripttutorial.net/` and `https://www.javascripttutorial.net/` aren't the same origin since they have different protocols and ports:

	https://www.javascripttutorial.net	**http://www.javascripttutorial.net**
Protocol	https	http
Port	443	80

IndexedDB adheres to the same-origin policy, meaning each origin has its own set of databases. One origin cannot access databases from another origin.

Basic IndexedDB operations

The following describes the basic operations of the IndexedDB databases such as

- Opening a connection to a database.
- Inserting an object into the object store.
- Reading data from the object store.
- Using a cursor to iterate over a result set.
- Deleting an object from the object store.

Before opening a connection to a database in the IndexedDB, let's create the project structure first.

1) Create the project structure

First, create a new folder called `indexeddb` folder. Inside the `indexeddb` folder, create another subfolder called `js` .

Second, create the `index.html` in the `indexeddb` folder, `app.js` in the `js` folder.

Third, place the `<script>` tag that links to the `app.js` file in the `index.html` file like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>IndexedDB</title>
</head>
<body>
    <script src="js/app.js"></script>
</body>
</html>
```

In app.js, you'll place all the JavaScript code in an IIFE.

```
(function () {
   // all the code will be here
   // ...
})();
```

1) Check if the IndexedDB is supported

The following code checks if a web browser supports the IndexedDB:

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

Since most modern web browsers support the IndexedDB, this may not be necessary anymore.

2) Open a database

To open a connection to a database, you use the `open()` method of the `window.indexedDB` :

```
const request = indexedDB.open('CRM', 1);
```

The `open()` method accepts two arguments:

- The database name (`CRM`)
- The database version (`1`)

The `open()` method returns a request object which is an instance of the `IDBOpenDBRequest` interface.

When you call the `open()` method, it can succeed or fail. To handle each case, you can assign the corresponding event handler as follows:

```
request.onerror = (event) => {
    console.error(`Database error: ${event.target.errorCode}`);
};

request.onsuccess = (event) => {
    // add implementation here
};
```

3) Create object stores

When you open the database for the first time, the `onupgradeneeded` event will trigger.

If you open the database for the second time with a version higher than the existing version, the `onupgradeneeded` event also triggers.

For the first time, you can use the `onupgradeneeded` event handler to initialize the object stores and indexes.

For example, the following `onupgradeneeded` event handler creates the `Contacts` object store and its index.

```
// create the Contacts object store and indexes
request.onupgradeneeded = (event) => {
    let db = event.target.result;

    // create the Contacts object store
    // with auto-increment id
```

```
    let store = db.createObjectStore('Contacts', {
        autoIncrement: true
    });

    // create an index on the email property
    let index = store.createIndex('email', 'email', {
        unique: true
    });
};
```

How it works.

- First, get the `IDBDatabase` instance from the `event.target.result` and assign it to the `db` variable.
- Second, call the `createObjectStore()` method to create the `Contacts` object store with the `autoincrement` key. It means that the IndexedDB will generate an auto-increment number starting at one as the key for every new object inserted into the `Contacts` object store.
- Third, call the `createIndex()` method to create an index on the `email` property. Since the email is unique, the index should also be unique. To do so, you specify the third argument of the `createIndex()` method `{ unique: true }`.

4) Insert data into object stores

Once you open a connection to the database successfully, you can manage data in the `onsuccess` event handler.

For example, to add an object to an object store, you follow these steps:

- First, open a new transaction.
- Second, get an object store.
- Third, call the `put()` method of the object store to insert a new record.
- Finally, close the connection to the database once the transaction is completed.

The following `insertContact()` function inserts a new contact into the `Contacts` object store:

```
function insertContact(db, contact) {
    // create a new transaction
    const txn = db.transaction('Contacts', 'readwrite');

    // get the Contacts object store
    const store = txn.objectStore('Contacts');
    //
    let query = store.put(contact);

    // handle success case
    query.onsuccess = function (event) {
        console.log(event);
    };

    // handle the error case
    query.onerror = function (event) {
        console.log(event.target.errorCode);
    }

    // close the database once the
    // transaction completes
    txn.oncomplete = function () {
        db.close();
    };
}
```

To create a new transaction, you call the `transaction()` method of the `IDBDatabase` object.

You can open a transaction in one of two modes: `readwrite` or `readonly`.

The `readwrite` mode allows you to read data from and write data to the database, while the `readonly` mode allows reading data from the database.

It's best to open a `readonly` transaction when you only need to read data from a database.

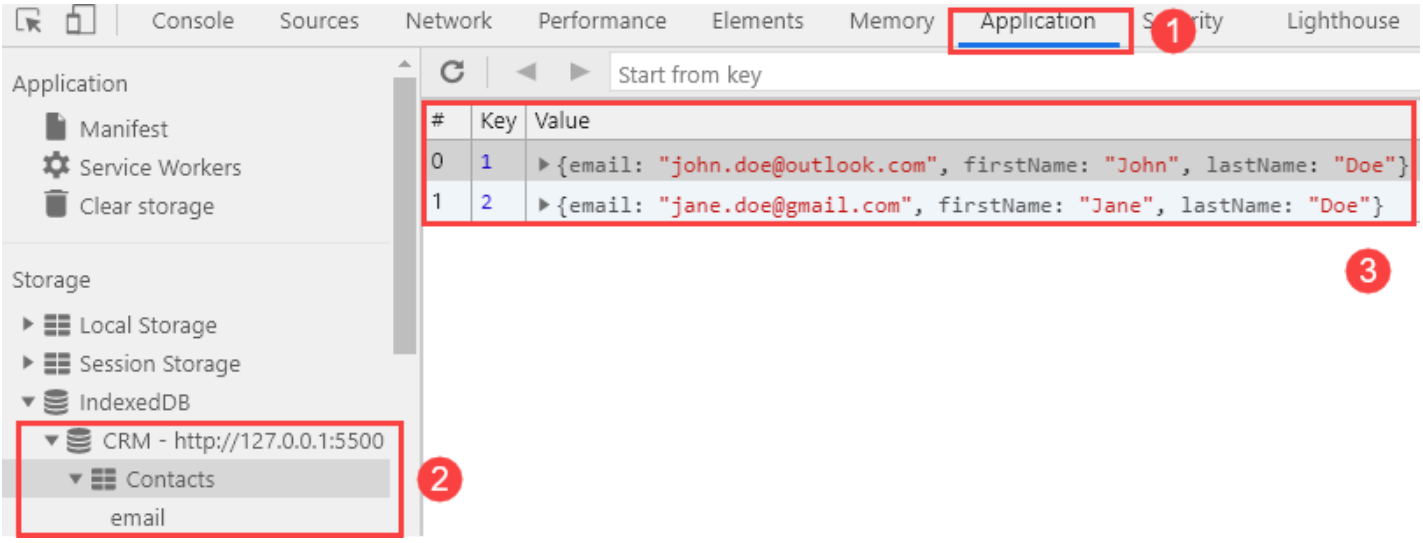
After defining the `insertContact()` function, you can call it in the `onsuccess` event handler of the request to insert one or more contacts like this:

```
request.onsuccess = (event) => {
      const db = event.target.result;

      insertContact(db, {
          email: 'john.doe@outlook.com',
          firstName: 'John',
          lastName: 'Doe'
      });

      insertContact(db, {
          email: 'jane.doe@gmail.com',
          firstName: 'Jane',
          lastName: 'Doe'
      });
};
```

Now, if you open the `index.html` file in the web browser, the code in the app.js will execute to:

- Create the `CRM` database in the IndexedDB.
- Create the `Contacts` object store in the `CRM` database.
- Insert two records into the object store.

If you open the devtools on the web browser, you'll see the CRM database with the `Contacts` object store. And in the `Contacts` object store, you'll see the data there as shown in the following picture:



5) Read data from the object store by key

To read an object by its key, you use the `get()` method of the object store. The following `getContactById()` function finds a contact by an id:

```
function getContactById(db, id) {
    const txn = db.transaction('Contacts', 'readonly');
    const store = txn.objectStore('Contacts');

    let query = store.get(id);

    query.onsuccess = (event) => {
        if (!event.target.result) {
            console.log(`The contact with ${id} not found`);
        } else {
            console.table(event.target.result);
        }
    };

    query.onerror = (event) => {
        console.log(event.target.errorCode);
    }

    txn.oncomplete = function () {
        db.close();
    };
};
```

When you call the get() method of the object store, it returns a query that will execute asynchronously.

Because the query can succeed or fail, you need to assign the `onsuccess` and `onerror` handlers to handle each case.

If the query succeeds, you'll get the result in the event.target.result. Otherwise, you'll get an error code via event.target.errorCode.

The following code closes the connection to the database once the transaction completes:

```
txn.oncomplete = function () {
   db.close();
};
```

The database connection is closed only when all the transactions are completed.

The following calls the `getContactById()` in the `onsuccess` event handler to get the contact with id 1:

```
request.onsuccess = (event) => {
    const db = event.target.result;
    getContactById(db, 1);
};
```

Output:

(index)	Value
email	"john.doe@outlook.com"
firstName	"John"
lastName	"Doe"

6) Read data from the object store by an index

The following defines a new function called `getContactByEmail()` that uses the email index to query data:

```
function getContactByEmail(db, email) {
    const txn = db.transaction('Contacts', 'readonly');
    const store = txn.objectStore('Contacts');

    // get the index from the Object Store
    const index = store.index('email');
    // query by indexes
    let query = index.get(email);

    // return the result object on success
    query.onsuccess = (event) => {
```

```
        console.log(query.result); // result objects
    };

    query.onerror = (event) => {
        console.log(event.target.errorCode);
    }

    // close the database connection
    txn.oncomplete = function () {
        db.close();
    };
}
```

How it works.

- First, get the email index object from the `Contacts` object store.
- Second, use the index to read the data by calling the `get()` method.
- Third, show the result in the `onsuccess` event handler of the query.

The following illustrates how to use the `getContactByEmail()` function in the `onsuccess` event handler:

```
request.onsuccess = (event) => {
    const db = event.target.result;
    // get contact by email
    getContactByEmail(db, 'jane.doe@gmail.com');
};
```

Output:

(index)	Value
email	"jane.doe@gmail.com"
firstName	"Jane"
lastName	"Doe"

7) Read all data from an object store

The following shows how to use a cursor to read all the objects from the `Contacts` object store:

```
function getAllContacts(db) {
    const txn = db.transaction('Contacts', "readonly");
    const objectStore = txn.objectStore('Contacts');

    objectStore.openCursor().onsuccess = (event) => {
        let cursor = event.target.result;
        if (cursor) {
            let contact = cursor.value;
            console.log(contact);
            // continue next record
            cursor.continue();
        }
    };
    // close the database connection
    txn.oncomplete = function () {
        db.close();
    };
}
```

The `objectStore.openCursor()` returns a cursor used to iterate over an object store.

To iterate over the objects in an object store using the cursor, you need to assign an `onsuccess` handler:

```
objectStore.openCursor().onsuccess = (event) => {
   //...
};
```

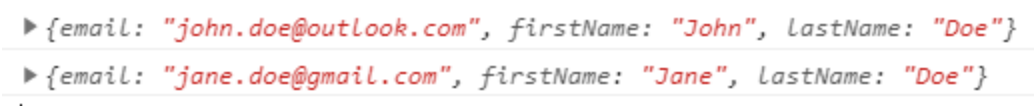
The `event.target.result` returns the cursor. To get the data, you use the `cursor.value` property.

The `cursor.continue()` method advances the cursor to the position of the next record in the object store.

The following calls the `getAllContacts()` in the `onsuccess` event handler to show all data from the `Contacts` object store:

```
request.onsuccess = (event) => {
    const db = event.target.result;
    // get all contacts
    getAllContacts(db);
};
```

Output:



8) Delete a contact

To delete a record from the object store, you use the `delete()` method of the object store.

The following function deletes a contact by its id from the `Contacts` object store:

```
function deleteContact(db, id) {
    // create a new transaction
    const txn = db.transaction('Contacts', 'readwrite');

    // get the Contacts object store
    const store = txn.objectStore('Contacts');
    //
    let query = store.delete(id);

    // handle the success case
    query.onsuccess = function (event) {
        console.log(event);
    };

    // handle the error case
    query.onerror = function (event) {
        console.log(event.target.errorCode);
    }

    // close the database once the
    // transaction completes
```

```
    txn.oncomplete = function () {
        db.close();
    };
}
```

You can call the `deleteContact()` function in the `onsuccess` event handler to delete the contact with id 1 as follows:

```
request.onsuccess = (event) => {
    const db = event.target.result;
    deleteContact(db, 1);
};
```

If you run the code, you'll find that the contact with id 1 will be deleted.

Put it all together

The following shows the complete app.js file:

```
(function () {
    // check for IndexedDB support
    if (!window.indexedDB) {
        console.log(`Your browser doesn't support IndexedDB`);
        return;
    }

    // open the CRM database with the version 1
    const request = indexedDB.open('CRM', 1);

    // create the Contacts object store and indexes
    request.onupgradeneeded = (event) => {
        let db = event.target.result;

        // create the Contacts object store
        // with auto-increment id
        let store = db.createObjectStore('Contacts', {
            autoIncrement: true
        });
```

```
    // create an index on the email property
    let index = store.createIndex('email', 'email', {
        unique: true
    });
};

// handle the error event
request.onerror = (event) => {
    console.error(`Database error: ${event.target.errorCode}`);
};

// handle the success event
request.onsuccess = (event) => {
    const db = event.target.result;

    // insert contacts
    // insertContact(db, {
    //     email: 'john.doe@outlook.com',
    //     firstName: 'John',
    //     lastName: 'Doe'
    // });

    // insertContact(db, {
    //     email: 'jane.doe@gmail.com',
    //     firstName: 'Jane',
    //     lastName: 'Doe'
    // });


    // get contact by id 1
    // getContactById(db, 1);


    // get contact by email
    // getContactByEmail(db, 'jane.doe@gmail.com');

    // get all contacts
    // getAllContacts(db);
```

```
    deleteContact(db, 1);

};

function insertContact(db, contact) {
    // create a new transaction
    const txn = db.transaction('Contacts', 'readwrite');

    // get the Contacts object store
    const store = txn.objectStore('Contacts');
    //
    let query = store.put(contact);

    // handle success case
    query.onsuccess = function (event) {
        console.log(event);
    };

    // handle the error case
    query.onerror = function (event) {
        console.log(event.target.errorCode);
    }

    // close the database once the
    // transaction completes
    txn.oncomplete = function () {
        db.close();
    };
}


function getContactById(db, id) {
    const txn = db.transaction('Contacts', 'readonly');
    const store = txn.objectStore('Contacts');

    let query = store.get(id);

    query.onsuccess = (event) => {
        if (!event.target.result) {
            console.log(`The contact with ${id} not found`);
```

```
        } else {
            console.table(event.target.result);
        }
    };

    query.onerror = (event) => {
        console.log(event.target.errorCode);
    }

    txn.oncomplete = function () {
        db.close();
    };
};

function getContactByEmail(db, email) {
    const txn = db.transaction('Contacts', 'readonly');
    const store = txn.objectStore('Contacts');

    // get the index from the Object Store
    const index = store.index('email');
    // query by indexes
    let query = index.get(email);

    // return the result object on success
    query.onsuccess = (event) => {
        console.table(query.result); // result objects
    };

    query.onerror = (event) => {
        console.log(event.target.errorCode);
    }

    // close the database connection
    txn.oncomplete = function () {
        db.close();
    };
}

function getAllContacts(db) {
    const txn = db.transaction('Contacts', "readonly");
```

```
    const objectStore = txn.objectStore('Contacts');

    objectStore.openCursor().onsuccess = (event) => {
        let cursor = event.target.result;
        if (cursor) {
            let contact = cursor.value;
            console.log(contact);
            // continue next record
            cursor.continue();
        }
    };
    // close the database connection
    txn.oncomplete = function () {
        db.close();
    };
}


function deleteContact(db, id) {
    // create a new transaction
    const txn = db.transaction('Contacts', 'readwrite');

    // get the Contacts object store
    const store = txn.objectStore('Contacts');
    //
    let query = store.delete(id);

    // handle the success case
    query.onsuccess = function (event) {
        console.log(event);
    };

    // handle the error case
    query.onerror = function (event) {
        console.log(event.target.errorCode);
    }

    // close the database once the
    // transaction completes
    txn.oncomplete = function () {
```

```
            db.close();
        };

    }
})();
```

Summary

- The IndexedDB is a large-scale object stored in web browsers.
- The IndexedDB stores data as key-value pairs. The values can be any data including simple and complex ones.
- The IndexedDB consists of one or more databases. Each database has one or more object stores. Typically, you create a database in the IndexedDB per web application.
- The IndexedDB is useful for web applications that don’t require a constant internet connection, especially those that work both online and offline.