TODO with Indexed DB

IndexedDB (currently version 2) provides client-sided database storage for web applications. From the specification:

> This document defines APIs for a database of records holding simple values and hierarchical objects. Each record consists of a key and some value. Moreover, the database maintains indexes over records it stores. An application developer directly uses an API to locate records either by their key or by using an index. A query language can be layered on this API. An indexed database can be implemented using a persistent B-tree data structure.

I've always found it hard to reason through IndexedDB, when it would be a good case to use it and whether it would be used on its own or in addition to other APIs.

This time I will try using raw IDB to create a TODO list browser-side. The idea is that this app will not use external dependencies, they will all be hosted on the same server as the application.

Project Structure

The idea of the Todo list is to create a local copy of a Todo database and let the user work with it in the local computer without having to be online to interact with it.

This is not meant for production. The very fact that it'll work on an individual computer means that we need to be at that computer to make changes to the database.

But it's an interesting exercise to learn about IDB, see how it works and see what the shortcomings (other than locality) may be.

Initializing the database

The first thing I do is to create a feature detection function that I will use shortly to make sure the browser supports IndexedDB before moving forward.

```
function idbOK() {
  return 'indexedDB' in window;)
}
```

We check if the browser supports IndexedDB and if it doesn't then we bail early and log it to console. I might change the idbOK function to do the check and the bailout there.

```
// No support? Go in the corner and pout.
if (!idbOK()) {
  console.log('Indexed DB not supported');
}
```

Now we can start working on creating the database and setting up the store and the indices that we want to use in the database.

Because the database doesn't exist the upgrade is needed and it's only during the onupgradeneeded event that you can make changes to the databbase like setting it up for the first time.

We first check if there is a `todos` object store, if there isn't one then we create it.

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

todosdb.onupgradeneeded = function(e) {
  const db = e.target.result;
  console.log('running onupgradeneeded');

  if (!db.objectStoreNames.contains('todos')) {
    console.log('making a new object store');
    let store = db.createObjectStore('todos', {
        keyPath: 'id',
        autoIncrement: true,
    });
```

We set up indices for different fields in the database. These indices will help us

when querying information later on.

```
    let titleIndex = store.createIndex('by_title', 'title', {'unique': fa
    let startIndex = store.createIndex('by_start', 'startDate');
    let completeIndex = store.createIndex('by_completed', 'complete');
  }
};
```

All functions related to IDB trigger success (`onsuccess`) and error (`onerror`) events.

```
todosdb.onsuccess = function(e) {
  console.log('running onsuccess');
  db = e.target.result;
  // console.dir(db.objectStoreNames);
};

todosdb.onerror = function(e) {
  console.log('onerror!');
  console.error(e);
};
'onerror!'
```

I've moved this constant and event outside of functions to make sure that it works when calling a function.

```
const addTodoButton = document.querySelector('#addTodo');
addTodoButton.addEventListener('click', addTodo);
```

Adding Todos

Now that we've created the database we can add new todos.

The first part is to capture the values from the form we have in the web page we use to access the IDB content.

```
function addTodo() {
  const title = document.getElementById('title').value;
  const description = document.getElementById('description').value;
  const dateStarted = document.getElementById('dateStarted').value;
  const dateEnded = document.getElementById('dateEnded').value;
  const completed = document.getElementById('dateStarted').value;
```

We then create a transaction and store objects. This tells IDB what objectStore what we want to work with and that we want to work with a transaction. A transaction is an atomic unit of work, either it all succeeds and gets pushed to IDB or things fail and nothing gets pushed.

```
  // Get a transaction
  const transaction = db.transaction(['todos'], 'readwrite')'todos'// Ask
  const store = transaction.objectStore('todos');
```

Next we define a todo object matching the elements we captured from the form with the names that we want to use in the database.

```
  // Define a todo
  const todo = {
    title: title,
    description: description,
    dateStarted: dateStarted,
    dateEnded: dateEnded,
    completed: completed,
  };
```

Lastly we perform the addition using `store.add` and give success and error handlers in case we waant to do something special.

```
  // Perform the add
  let request = store.add(todo);

  request.onerror = function(e) {
    console.log('Error', e.target.er'Error'e);
```

```
    // some type of error handler
  };

  request.onsuccess = function(e) {
    console.log('Woot! Did it');
  };
}
'Woot! Did it'
```

Show all records

Showing all the records combines retrieving them, creating a template for each instance and attaching the template to the page.

We first open the database and, on success, we create a transaction and select the object store and open a cursort to the object store.

```
function showAll() {
  let dbrequest = window.indexedDB.open('todosdb');

  dbrequest.onsuccess = function(e) {
    let transaction = db.transaction(['todos'], 'readonly');
    let request = transaction.objectStore('todos').openCursor();
```

We iterate over the content of the cursor. If it's empty then we return since we don't have any data to display.

```
    request.onsuccess = function(e) {
      let cursor = request.result || e.result;

      if (!cursor) {
        return;
      }
```

If the cursor is not empty we use it to iterate over the items in the database, popullate the template defined in `todoContent` and attach it to the existing

document.

Some of these values take advantrage of the fact that we can use Javascript values inside the parameters; I've used ternary if operators to make sure that the dates valuess don't return undefined if there is nothing there but ooutput a blank space instead.

```
      let container = document.getElementById('todos-listing');

      // Define the content for each individual entry
      let todoContent = `<h2>${cursor.value.title}</h2>
      <div class="todo-desc">
        ${cursor.value.description ? cursor.value.descri"todo-desc"`<h2>$
      <div class="todo-desc">
        ${cursor.value.description ? cursor.value.description : ''''}
      </div>
      <p>Date Started:
        ${cursor.value.dateStarted ? cursor.value.dateStarted : ''''}</p>
      <p>Date Finished:
        ${cursor.value.dateCompleted ? cursor.value.dateCompleted x: ''''
      <p>Completed: ${cursor.value.completed}`      cursor.continue();
    };

    request.onerror = function(e) {
      console.log('Error', e.target.error.name);
      // some typ'Error'// some type of error handler
    };
  };
}
```

This is the basic functionality corresponding to Create and Display in a CRUD application. Update and is more complicated as it requires creating a form where the user can edit the content of the todo and delete requires some way for the user to confirm that they want to delete the content.

Those two methods along with a way to update the database in place will come in another series of posts