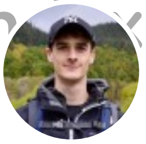
Create Your Own To-Do App with HTML5 and IndexedDB



Matt West
writes on June 10, 2013

IndexedDB is a client-side web technology that allows developers to build applications that are able to store data locally. Unlike LocalStorage, which enables the storage of simple key/value pairs, IndexedDB supports the storage of structured data. This enables developers to build more complex applications.

In this blog post you are going to learn about IndexedDB by building a simple todo list application.

Note: You can download all of the code used in this tutorial here.

Building the Application View

Before you start writing the JavaScript code that will power your application you first need to set up a new page to display the todo items.

Create a new file called `index.html` that contains the following HTML code.

Note: You will need to serve this HTML file from a local development server in order to have access to IndexedDB. If you don't already have a local development server installed you might want to try XAMPP.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Todo List App</title>

  <link rel="stylesheet" href="style.css">
</head>
<body>
  <div id="page-wrapper">
    <!-- Form for new Todo Items -->
    <form id="new-todo-form" method="POST" action="#">
      <input type="text" name="new-todo" id="new-todo" placehold
er="Enter a todo item..." required>
    </form>

    <!-- Todo Item List -->
    <ul id="todo-items"></ul>
  </div>

  <script src="db.js"></script>
  <script src="app.js"></script>
</body>
</html>
```

Now that you have your HTML file setup lets create a simple stylesheet for your app. Create a new file called `style.css` and add to it the following CSS code. This file should be created in the same folder as your `index.html` file.

```
* { -moz-box-sizing: border-box; -webkit-box-sizing: border-box;
 box-sizing: border-box; }

body, html {
  padding: 0;
  margin: 0;
}

body {
  font-family: Helvetica, Arial, sans-serif;
  color: #545454;
  background: #F7F7F7;
}

#page-wrapper {
  width: 550px;
  margin: 2.5em auto;
  background: #FFF;
  box-shadow: 0 1px 3px rgba(0,0,0,0.2);
  border-radius: 3px;
}

#new-todo-form {
  padding: 0.5em;
  background: #0088CC;
  border-top-left-radius: 3px;
  border-top-right-radius: 3px;
}

#new-todo {
  width: 100%;
  padding: 0.5em;
  font-size: 1em;
  border-radius: 3px;
  border: 0;
}

#todo-items {
  list-style: none;
  padding: 0.5em 1em;
  margin: 0;
}

#todo-items li {
  font-size: 0.9em;
  padding: 0.5em;
  background: #FFF;
  border-bottom: 1px solid #EEE;
  margin: 0.5em 0;
}

input[type="checkbox"] {
  margin-right: 10px;
}
```

Now that you have the barebones of your app setup lets move on to writing some code that will handle saving, retrieving and deleting todo items from the database.

Creating The Database Module

In order to make your code more maintainable and reusable you are going to create a JavaScript *module* that will contain all of the code that handles interactions with the database. A *module* is an encapsulated piece of code that has a specific responsibility.

Create a new file in your project folder called `db.js` and add to it the following code.

```
var todoDB = (function() {
  var tDB = {};
  var datastore = null;
```

```
    // TODO: Add methods for interacting with the database here.

    // Export the tDB object.
    return tDB;
  }());
```

Here you have created the beginnings of your JavaScript module. The first and last lines create a new module called `todoDB`. You then create an empty JavaScript object called `tDB`. This will be used to store all of the methods in the module that you want to be accessible from outside the scope of the module. It is possible to create variables and methods that are only accessible within a module. You then create a `datastore` variable that will be used to store a reference to the database. Notice that this variable has not been created as part of the `tDB` object. This means that the variable will not be accessible outside of the module scope.

Note: For more information on JavaScript modules and scopes check out this great article by Ben Cherry. http://www.adequatelygood.com/JavaScript-Module-Pattern-In-Depth.html

---

Now that you have your module setup it's time to start writing the code that will interact with the IndexedDB database.

Add the following method below your declaration of the `datastore` variable.

```
  /**
   * Open a connection to the datastore.
   */
  tDB.open = function(callback) {
    // Database version.
    var version = 1;

    // Open a connection to the datastore.
    var request = indexedDB.open('todos', version);

    // Handle datastore upgrades.
    request.onupgradeneeded = function(e) {
      var db = e.target.result;
      e.target.transaction.onerror = tDB.onerror;

      // Delete the old datastore.
      if (db.objectStoreNames.contains('todo')) {
        db.deleteObjectStore('todo');
      }

      // Create a new datastore.
      var store = db.createObjectStore('todo', {
        keyPath: 'timestamp'
      });
    };

    // Handle successful datastore access.
    request.onsuccess = function(e) {
      // Get a reference to the DB.
      datastore = e.target.result;

      // Execute the callback.
      callback();
    };

    // Handle errors when opening the datastore.
    request.onerror = tDB.onerror;
  };
```

Note: This method takes an argument named `callback`. Database transactions are asynchronous meaning that the browser will not wait for the database request to finish before moving on to the next bit of code it needs to execute. This means that we need to specify a *callback* function that will be executed once the request has finished in order to make use of the data.

The `open` method is responsible for opening a new connection to the database. You start by declaring a variable (`version`) that stores the database version. This is needed in order to keep track of database upgrades. You might want to upgrade the database if you needed to add new object stores (think of these like database tables) or change the key for an object store.

You then open a connection to the database using the `indexedDB.open` method. The first parameter specifies the object store that you want to access and the second paramter specifies the database version. If the object store does not exist or the version has changed the `onupgradeneeded` event will be triggered, we will look at this next.

The next step in your code is to create an event listener for the `onupgradeneeded` event we just looked at. Here you first get a reference to the database from the event data (`e.target.result`) and store this in a variable called `db` . You then check to see if the object store exists and if it does you delete it. After that you create a new object store using the `createObjectStore` method, passing in the name of the object store (`todo`) and a JavaScript object that contains some settings. In this settings object you have specified that the key that your todo items should be stored under will be a property called `timestamp` . We will come back to this later.

The `onsuccess` event listener will get a reference to the database from the event data (`e.target.result`) and use this to set the `datastore` variable. It then executes the `callback` function. You will see the importance of this callback function later in this tutorial.

Next you are going to create a method that will be responsible for fetching all the todo items from the database. Copy the following code below your `tDB.open` method.

```
/**
 * Fetch all of the todo items in the datastore.
 */
tDB.fetchTodos = function(callback) {
  var db = datastore;
  var transaction = db.transaction(['todo'], 'readwrite');
  var objStore = transaction.objectStore('todo');

  var keyRange = IDBKeyRange.lowerBound(0);
  var cursorRequest = objStore.openCursor(keyRange);

  var todos = [];

  transaction.oncomplete = function(e) {
    // Execute the callback function.
    callback(todos);
  };

  cursorRequest.onsuccess = function(e) {
    var result = e.target.result;

    if (!!result == false) {
      return;
    }

    todos.push(result.value);

    result.continue();
  };

  cursorRequest.onerror = tDB.onerror;
};
```

At the beginning of the `fecthTodos` method you first create a new variable `db` and set this to the `datastore` variable you initialized earlier.

You then create a new IDBTransaction using this `db` variable and assign this to a variable called `transaction`. This transaction will handle the interaction with the database.

Using the `objectStore` method on the transaction you get a reference to the `todo` object store and save this reference in a new variable called `objStore`.

Next you create a `IDBKeyRange` object that specifies the range of items in the object store that you want to retrieve. In your case you want to get all of the items so you set the lower bound of the range to 0. This will select all keys from 0 up.

Now that you have a key range you can create a cursor that will be used to cycle through each of the todo items in the database. This is assigned to a new variable called `cursorRequest`.

You then create an empty array (`todos`) that will be used to store the todo items once they have been fetched from the database.

The `transaction.oncomplete` event handler is used to execute the callback function once all of the todo items have been fetched. The `todos` array will be passed into this callback as a parameter.

The `cursorRequest.onsuccess` event handler is triggered for each item that is returned from the database. Here you first check to see if the result contains a todo item and then if it does you add that item to the `todos` array. The `result.continue()` method is then called which will move the cursor on to the next item in the database.

Finally, you declare an error handler that should be used if the cursor encounters a problem.

Now it's time to write a method that will handle adding new todo items to the database. Copy the following code into your `todoDB` module.

```
/**
 * Create a new todo item.
 */
tDB.createTodo = function(text, callback) {
  // Get a reference to the db.
  var db = datastore;

  // Initiate a new transaction.
  var transaction = db.transaction(['todo'], 'readwrite');
```

```
    // Get the datastore.
    var objStore = transaction.objectStore('todo');

    // Create a timestamp for the todo item.
    var timestamp = new Date().getTime();

    // Create an object for the todo item.
    var todo = {
      'text': text,
      'timestamp': timestamp
    };

    // Create the datastore request.
    var request = objStore.put(todo);

    // Handle a successful datastore put.
    request.onsuccess = function(e) {
      // Execute the callback function.
      callback(todo);
    };

    // Handle errors.
    request.onerror = tDB.onerror;
  };
```

In this method you do the same setup for creating a database transaction as you did before. You then generate a timestamp. This will be used as the key for the todo item.

Next you create an object (`todo`) with two properties, `text` and `timestamp` . The text property is set using the text parameter passed into the method and the timestamp is set using the `timestamp` variable you just created.

To save the todo item you call the `put` method on the object store, passing in the `todo` object.

Finally you setup event handlers for `onsuccess` and `onerror` . If the todo item is successfully saved you execute the callback function, passing in the new todo item as a parameter.

The final method that is needed for the database module is a way of deleting todo items. Copy the following code into your module.

```
  /**
   * Delete a todo item.
   */
  tDB.deleteTodo = function(id, callback) {
    var db = datastore;
    var transaction = db.transaction(['todo'], 'readwrite');
    var objStore = transaction.objectStore('todo');

    var request = objStore.delete(id);
    request.onsuccess = function(e) {
      callback();
    }

    request.onerror = function(e) {
      console.log(e);
    }
  };
```

This method takes an `id` for the item that is to be deleted and a callback function that will be executed if the request is successful.

After doing the standard setup to get a reference to the object store you use the object store's `delete` method to remove the todo item from the database.

You setup an `onsuccess` event listener that will execute the callback function and an `onerror` handler that will log any errors to the console.

That's the database module done! Next you are going to write the app code that will handle displaying todos on the screen and taking input for new todo items.

Creating the App Code

Create a new file called `app.js` and save this in the same folder as your `index.html` file. This new file will contain all of the code that handles interactions with the app UI.

Add the following code to your `app.js` file. Any code that you put between the curly braces here will be executed when the page loads.

```
window.onload = function() {
  // TODO: App Code goes here.
};
```

Now open a connection to the database by calling the `todoDB.open` method that you created earlier. You have access to `todoDB` here because the `db.js` file is loaded before `app.js` .

Pass in `refreshTodos` as the callback. You will write the `refreshTodos` method shortly.

```
// Display the todo items.
todoDB.open(refreshTodos);
```

Now get references to the new todo item form and text input field.

```
// Get references to the form elements.
```

```
var newTodoForm = document.getElementById('new-todo-form');
var newTodoInput = document.getElementById('new-todo');
```

Your next task is to setup an event listener for when the form is submitted.

```
// Handle new todo item form submissions.
newTodoForm.onsubmit = function() {
  // Get the todo text.
  var text = newTodoInput.value;

  // Check to make sure the text is not blank (or just spaces).
  if (text.replace(/ /g,'') != '') {
    // Create the todo item.
    todoDB.createTodo(text, function(todo) {
      refreshTodos();
    });
  }

  // Reset the input field.
  newTodoInput.value = '';

  // Don't send the form.
  return false;
};
```

Here you first get the text for the new todo item by accessing the `value` property on the text input. To prevent blank todo items from being added to the database you do a quick check to see if the text you gathered is more than just whitespace. You then issue a command to `todoDB.createTodo` passing in the text for the new todo item as well as a callback function that will execute `refreshTodos` to update the UI when the new item has been saved.

Finally you clear the text input and return `false` so that the form does not cause a new HTTP request.

Now lets write that `refreshTodos` method. This will fetch all of the todo items from the database and display them in the todos list. Copy the following code into `app.js` .

```
// Update the list of todo items.
function refreshTodos() {
  todoDB.fetchTodos(function(todos) {
    var todoList = document.getElementById('todo-items');
    todoList.innerHTML = '';

    for(var i = 0; i < todos.length; i++) {
      // Read the todo items backwards (most recent first).
      var todo = todos[(todos.length - 1 - i)];

      var li = document.createElement('li');
      li.id = 'todo-' + todo.timestamp;
      var checkbox = document.createElement('input');
```

```
        checkbox.type = "checkbox";
        checkbox.className = "todo-checkbox";
        checkbox.setAttribute("data-id", todo.timestamp);

        li.appendChild(checkbox);

        var span = document.createElement('span');
        span.innerHTML = todo.text;

        li.appendChild(span);

        todoList.appendChild(li);

        // Setup an event listener for the checkbox.
        checkbox.addEventListener('click', function(e) {
          var id = parseInt(e.target.getAttribute('data-id'));

          todoDB.deleteTodo(id, refreshTodos);
        });
      }

    });
  }
```

Here you execute the `todoDB.fetchTodos` method with a callback which gets passed an array of todo items.

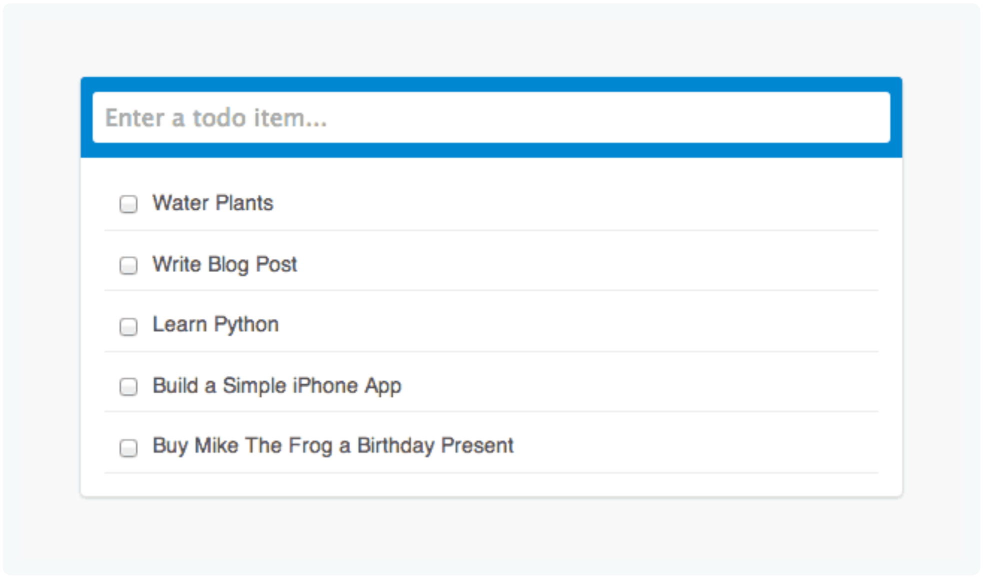
Inside this callback you first get a reference to the todo items list and then make sure that this element has no HTML content.

You then loop through each of the todo items in reverse order so that the most recent todo items are displayed at the top of the list. For each todo item you create a new `<li>` element that contains a checkbox for marking the todo as complete and a `<span>` element that contains the todo item text. The checkbox has a special attribute called `data-id` that contains the timestamp for the todo item. After creating each `<li>` you append it to the `todoList`.

Finally you setup an event listener on each checkbox that will be triggered when the user clicks to complete an item. Inside this event listener you first get the todo items id from the `data-id` attribute on the checkbox. You then execute the `todoDB.deleteTodo` method, passing in the todo item id and specifying `refreshTodos` as the callback function.

You're done! If you load up the `index.html` file in your web browser you should now be able to add todo items to the list and also mark them as complete.

Final Thoughts


Enter a todo item...
Water Plants
Write Blog Post
Learn Python
Build a Simple iPhone App
Buy Mike The Frog a Birthday Present

The Finished Todo List App

IndexedDB allows developers to create a whole new level of client-side applications. In this post you have learned the basics of how to add and remove data from an IndexedDB database. If you're feeling adventurous you might want to try building on your app to allow editing of todo items, or maybe you want to keep completed todo items but have them marked as 'done' instead of just deleting them.

How do you plan to use IndexedDB in your projects? Let us know in the comments below.

Useful Links

- Can I use IndexedDB
- MDN IndexedDB Documentation
- Indexed Database API (W3C Spec)

app code html5 indexeddb javascript web app

33 Responses to “Create Your Own To-Do App with HTML5 and IndexedDB”

Victor Assis on April 3, 2018 at 12:08 pm said:

that’s awesome … helps a lot! thanks Matt 😀

sangeeta soni on July 7, 2017 at 1:23 am said:

im using indexeddb in my project.I have created the database .But now i want to search in data bse that value im searching is not a key.
I have created teo pages
1. taking value from user through a drop down list nad passing it to other page
2.on second page i taking value from url saving it to in a var

i want to search that value(name) in data base and retrive all vlues which have same name.
but my key path is id .
can u tell me how can i do it

Jahaber Sathik on August 17, 2015 at 12:21 am said:

Dear Matt,

It is indeed a good article especially like a person who would like to code as a beginner, I would like to implement edit and delete option and also to store in local database or web server. But I’m a beginner could you please help me for the same?

Thanks and regards
Sathik

> **Matt West on August 17, 2015 at 2:00 am said:**
>
> Hi Stahik,
>
> The current implementation includes a delete option. For an example of how to add edit functionality I’d recommend looking at this example project:
> http://todomvc.com/examples/vanillajs/
> https://github.com/tastejs/todomvc/tree/gh-pages/examples/vanillajs
>
> If you’d like to store the items on a server you will need to have a database like MySQL and some backend scripts that handle storing the data. If you’re interested in pursuing this further you may want to check out this tutorial:
> http://code.tutsplus.com/tutorials/how-to-code-a-fun-to-do-list-with-php-and-ajax–net-3170
>
> **nine on December 6, 2016 at 2:01 am said:**
>
> why don’t use simply :

> todoDB.createTodo(text, refreshTodos) ;
> instead of
> // Create the todo item.
> todoDB.createTodo(text, function(todo) {
> refreshTodos();
> });
>
> you return a todo we do nothing with ?

Sanjay Kadam on September 5, 2013 at 6:55 am said:

That's great… app

Tuan Jinn Nguyen on September 3, 2013 at 10:15 am said:

Nice one. However, I opened it using chrome (ver 29.0.1547.57 m), and Firefox (23.0.1) from local hard drive and it works like a charm. But the results are different, Chrome data shows what I entered in chrome while Firefox remains its own data. I see only 1 .DS_Store file. How does that happen, can you explain?

> **Matt West on September 3, 2013 at 10:24 am said:**
>
> Hi Tuan,
>
> Each browser uses it's own datastore so data added in Chrome will not be visible in Firefox and vice-versa.
>
> > **Tuan Jinn Nguyen on September 3, 2013 at 10:27 am said:**
> >
> > Thanks for your quick reply. My apology if the question is a bit stupid, but I was searching for more logical explanation myself, for instance: the data gotta be stored in some physical file – the database meta file (i was kind of thinking it would be: the .DS_Store file. But it seems I am wrong. I would feel more comfortable to pick up indexeddb as a solution for my app (using appjs) if what I thought were true.
> >
> > > **Matt West on September 5, 2013 at 2:54 am said:**
> > >
> > > Chrome looks to store the database files (on mac) in:
> > > /Users/{user}/Library/Application Support/Google/Chrome/Default/IndexedDB
> > >
> > > I'm not sure about other browsers.
> >
> > **Tuan Jinn Nguyen on September 3, 2013 at 10:32 am said:**
> >
> > well stupid me :D. .DS_Store is for Mac OS. But then the question would be how long would this indexeddb remains with the browser?
> >
> > > **Matt West on September 5, 2013 at 2:41 am said:**
> > >
> > > IndexedDB is a persistent storage technology so

it should remain there until the user clears it out.

Willem Mulder on September 30, 2013 at 5:36 am said:

Really, have you seen that for yourself? Because https://developers.google.com/chrome/whitepapers/storage#tempo lists it as Temporary storage, but that might also have to do with the fact that a browser is allowed to delete data when the disk gets full…

Matt West on September 30, 2013 at 5:45 am said:

Hi Willem!

Thanks for correction. You are right.

If you're building chrome apps/extensions you can have persistent IndexedDB if you use the `unlimitedStorage` permission.

There was talk of allowing persistent IndexedDB in any browser but as far as I'm aware nobody has implemented it yet.

Willem Mulder on September 30, 2013 at 9:05 am said:

Do you know what talk that was?

I'm really curious how I could have a really large (say 100Mb) local persistent DB. Is the only way to serialize the data and export/import it using the Files API?

Matt West on September 30, 2013 at 5:57 pm said:

It's referenced on developers.google.com:

"It is available only to apps that use the Files System API, but will eventually be available to other offline APIs like IndexedDB and Application Cache."

Yep, the Filesystem APIs are probably going to be the best way of guaranteeing persistent storage for the time being. Unless you're creating a chrome app/extension.

Willem Mulder on October 1, 2013 at 7:15 am said:

Thanks! 🙂

Tuan Jinn on September 3, 2013 at 10:05 am said:

Awesome stuff!!!!!!!

Ramesh on August 7, 2013 at 6:36 am said:

Ok .. i able to run on Web Server.. But, Big question, i have is ? why you implemented this to run on Web server.. Not running from harddrive.. !!

> Matt West on August 17, 2015 at 1:54 am said:
>
> Hi Ramesh,
>
> IndexedDB requires that the website is being served over http:// and not file:// as local files are. Installing software like XAMPP would allow you to test your sites on your local machine. See: https://www.apachefriends.org/index.html
>
> > John Mansfield on March 23, 2016 at 3:43 am said:
> >
> > IndexedDb seems to work fine from file:// now
> > (at least on Chrome v.49)

Ramesh on August 7, 2013 at 6:03 am said:

Uncaught Error: NotFoundError: DOM IDBDatabase Exception 8 … line no 60 db.js

sajay on June 28, 2013 at 2:59 am said:

Super Stuff Matt:D

confused on June 27, 2013 at 2:23 pm said:

When i open my index.html file all that show up is the text box to input todo items. i inputed todo items but don't see a list like your screenshot above?

> Matt West on June 28, 2013 at 4:45 am said:
>
> Are you serving the file from a local development server? Something like MAMP or XAMPP.
>
> IndexedDb won't work if the file is just opened form your hard drive.
>
> > confused on June 28, 2013 at 2:55 pm said:
> >
> > no. and that's my stupid mistake. forgive my naive-ness. I literally followed the instructions verbatim.
> >
> > > Matt West on June 29, 2013 at 4:41 am said:
> > >
> > > Good to hear you got it sorted 🙂

Tuan Jinn Nguyen on September 3, 2013 at 10:12 am said:

Hold on, that aint right. I opened it using chrome (ver 29.0.1547.57 m) from local hard drive and it works like a charm. And firefox (23.0.1) too. But the results are different

Joey Waddell on June 13, 2013 at 2:18 pm said:

this is awesome!

> Matt West on June 14, 2013 at 2:47 am said:
>
> Glad you enjoyed it 🙂

Hemanth Malli on June 11, 2013 at 8:06 am said:

I'm visiting your blog for the first time and its amazing. I loved it!! Keep going …

Stacy on June 11, 2013 at 4:45 am said:

WOW! amazing guide) thanks

> Matt West on June 11, 2013 at 7:34 am said:
>
> Thanks for reading Stacy 🙂