**HTML5 INDEXEDDB AND WEB**

**WORKERS**

**RAHUL KRISHNAN**

**SHAKEEL MUHAMMED**

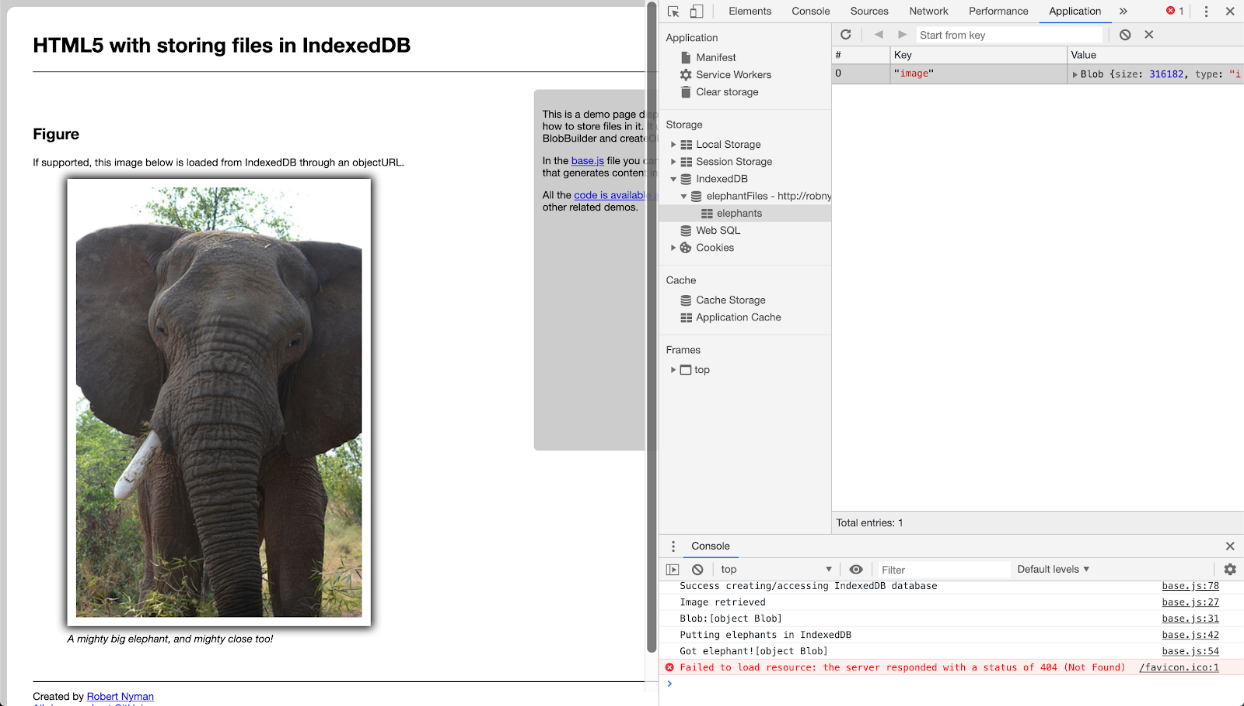
**SANNA MAHAJAN**

**SRIYA NAIR**

**IndexedDB**

* API for client-side storage of significant amounts of structured data.(Incl. BLOBS, files etc)
* Uses indexes to enable high-performance searches of this data.
* Libraries for easily implementing IndexedDB:
  + **localForage:** A Polyfill (code that implements a feature on web browsers that do not support the feature) providing a simple name:value syntax for client-side data storage, which uses IndexedDB in the background, but falls back to WebSQL and then to localStorage in browsers that don't support IndexedDB.
  + **Dexie.js**: A wrapper for IndexedDB that allows much faster development with a simple syntax.
  + **ZangoDB:** A MongoDB-like interface for IndexedDB that supports most of the familiar features of MongoDB.
  + **JsStore:** An IndexedDB wrapper with a syntax like SQL.
* Sites to see indexedDB in action:
  + <http://robnyman.github.io/html5demos/indexeddb/>
  + <http://mdn.github.io/to-do-notifications/>

Go to Inspect->Application->IndexedDB to see the database.



**Concepts behind IndexedDB**

IndexedDB is a transactional database system, like an SQL-based RDBMS. However, unlike SQL-based RDBMSes, which use fixed-column tables, IndexedDB is a JavaScript-based object-oriented database. IndexedDB lets you store and retrieve objects that are indexed with a key.IndexedDB follows a same-origin policy. So while you can access stored data within a domain, you cannot access data across different domains.

* **IndexedDB databases store key-value pairs.** The values can be complex structured objects, and keys can be properties of those objects.
* **IndexedDB is built on a transactional database model.** Everything you do in IndexedDB always happens in the context of a transaction. 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. You don't "store" a value into the database, or "retrieve" a value out of the database through synchronous means. 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.
* **IndexedDB is object-oriented.** IndexedDB is not a relational database with tables representing collections of rows and columns.
* **IndexedDB does not use Structured Query Language (SQL).** It uses NoSQL, However there are libraries using IndexedDB that support SQL.
* **IndexedDB adheres to a same-origin policy.** An origin is the domain, application layer protocol, and port of a URL of the document where the script is being executed.

**Interfaces of IndexedDB**

* **IDBEnvironment**

Provides access to IndexedDB functionality. It is implemented by the window and worker objects. This interface isn't part of the 2.0 specification.

* **IDBFactory**

Provides access to a database. This is the interface implemented by the global object indexedDB and is therefore the entry point for the API.

* **IDBOpenDBRequest**

Represents a request to open a database.

* **IDBDatabase**

Represents a connection to a database. It's the only way to get a transaction on the database.

* **IDBTransaction**

Represents a transaction. You create a transaction on a database, specify the scope (such as which object stores you want to access), and determine the kind of access (read only or readwrite) that you want.

* **IDBRequest**

Generic interface that handles database requests and provides access to results.

* **IDBObjectStore**

Represents an object store that allows access to a set of data in an IndexedDB database, looked up via primary key.

* **IDBIndex**

Also allows access to a subset of data in an IndexedDB database, but uses an index to retrieve the record(s) rather than the primary key. This is sometimes faster than using IDBObjectStore.

* **IDBCursor**

Iterates over object stores and indexes.

* **IDBCursorWithValue**

Iterates over object stores and indexes and returns the cursor's current value.

* **IDBKeyRange**

Defines a key range that can be used to retrieve data from a database in a certain range.

**Why client-side storage?**

There are several reasons to use client-side storage.   
First, you can make your app work when the user is offline, possibly syncing data back once the network is connected again.   
Second, it's a performance booster; you can show a large corpus of data as soon as the user clicks on to your site, instead of waiting for it to download again.   
Third, it's an easier programming model, with no server infrastructure required.   
Of course, the data is more vulnerable and the user can't access it from multiple clients, so you should only use it for non-critical data

**Other Methods of storing data in the browser  
  
WebStorage->LocalStorage-**  simply provides a key-value mapping, e.g. localStorage["name"] = username;. Unfortunately, present implementations only support string-to-string mappings, so you need to serialise and deserialise other data structures. You can do so using JSON.stringify()and JSON.parse().

**Drawbacks of LocalStorage**

* Poor performance for large/complex data, when using the synchronous API (which is the most widely supported mode).
* Poor performance when searching large/complex data, due to lack of indexing. (Search operations have to manually iterate through all items.)
* Poor performance when storing and retrieving large/complex data structures, because it's necessary to manually serialize and deserialize to/from string values. The major browser implementations only support string values (even though the spec says otherwise).
* Need to ensure data consistency and integrity, since data is effectively unstructured.

**WebSQL**- gives you all the power - and effort - of a structured SQL relational database.

**Drawbacks of WebSQL**

* Deprecated.
* Steep learning curve, requiring knowledge of relational databases and SQL.
* Diminishes agility, as database schema must be defined upfront, with all records in a table matching the same structure.

**WEB WORKERS**

**WHAT IS WEB WORKERS?**

If you want to do intensive tasks using JavaScript that is time consuming and require hefty calculations, the web page will freeze up and interrupts the user until the job is completed, as JavaScript code runs in the foreground. We usually don’t notice, because modern browsers are so fast. But throughout the lifecycle of a web page, this difference can become quite noticeable.

Web Worker is a JavaScript code that runs in the background, without affecting the performance of the page. You can continue to do whatever you want: clicking, selecting things, etc., while the web worker runs in the background. Unlike JavaScript the Web worker doesn't interrupt the user and the web page remain responsive because they are running tasks in the background.

**WEB WORKERS AND DOM**

Since web workers are in external files, they do not have access to the following JavaScript objects:

* The window object
* The document object
* The parent object

**HOW WEB WORKERS WORK?**

Workers are initialized with the URL of a JavaScript file, which contains the code the worker will execute. This code sets event listeners and communicates with the script that spawned it from the main page.

Syntax:

var worker = new Worker(forLoop.js);

If your application has multiple supporting JavaScript files, you can import them using importScripts() method which takes file name(s) as argument separated by comma as follows –

importScripts(“helper.js”, “anotherHelper.js”);

Once the Web Worker is spawned, communication between web worker and its parent page is done using the postMessage() method. Depending on the browser/version, postMessage() can accept either a string or JSON object as its single argument.

worker.postMessage(message);

**CHECK WEB WORKER SUPPORT**

Check whether the user’s browser supports a web worker before creating one.

if (typeof(Worker) !== "undefined") {  
  // Yes! Web worker support!  
  // *Some code.....*  
}

else {  
  // Sorry! No Web Worker support..  
}

The earliest browser versions which support Web Workers are:

* IE: 10
* Firefox: 3.5
* Chrome: 4
* Safari: 4
* Opera: 10.6

**CREATION OF WEB WORKER**

* <!DOCTYPE HTML>
* <html>
* <head>
* <title>Big for loop</title>
* <script>
* var worker = new Worker('bigLoop.js');
* worker.onmessage = function (event) {
* alert("Completed " + event.data + "iterations" );
* };
* function sayHello() {
* alert("Hello sir...." );
* }
* </script>
* </head>
* <body>
* <input type = "button" onclick = "sayHello();" value = "Say Hello"/>
* </body>
* </html>

TERMINATION OF A WEB WORKER

terminate() method used to terminate a web worker, and free browser/computer resources.

Syntax: w.terminate();

A terminated Web Worker will no longer respond to messages or perform any additional computations. We cannot restart a worker; instead, we can create a new worker using the same URL.