IndexedDB: basic concepts

INTRODUCTION

IndexedDB is very different from SQL databases, but don't be afraid if you've only used SQL databases: IndexedDB might seem complex at first sight, but it really isn't.

Let's quickly look at the main concepts of IndexedDB, as we will go into detail later on:

- IndexedDB stores and retrieves objects which are indexed by a "key".
- Changes to the database happen within transactions.
- IndexedDB follows a same-origin policy. So while you can access stored data within a domain, you cannot access data across different domains.
- It makes extensive use of an asynchronous API: most processing will be done in callback functions - and we mean *LOTS of callback functions*!

DETAILED OVERVIEW

IndexedDB databases store key-value pairs. The values can be complex structured objects (hint: imagine JSON objects), and keys can be properties of those objects. You can create indexes that use any property of the objects for quick searching, as well as sorted enumeration.

Example of data (we reuse some sample from this MDN tutorial):

```
// This is what our customer data looks like.
const customerData = [
{ ssn: "444-44-
4444", name: "Bill", age: 35,email: "bill@company.com" },
{ ssn: "555-55-
5555", name: "Donna", age: 32,email: "donna@home.org" }
];
```

Where customerData is an array of "customers", each customer having several properties: ssn for the social security number, a name, an age and an email.

IndexedDB is built on a transactional database model. Everything you do in IndexedDB always happens in the context of a transaction. The IndexedDB API provides lots of objects that represent indexes, tables, cursors, and so on, but each is tied to a particular transaction. Thus, you cannot execute commands or open cursors outside a transaction.

Example of a transaction:

```
// Open a transaction for reading and writing on the DB
"customer"
var transaction =db.transaction(["customers"], "readwrite");
// Do something when all the data is added to the database.
transaction.oncomplete = function(event) {
   alert("All done!");
};
transaction.onerror = function(event) {
   // Don't forget to handle errors!
};
// Use the transaction to add data...
var objectStore =transaction.objectStore("customers");

for (var i in customerData) {
   var request =objectStore.add(customerData[i]);
   request.onsuccess = function(event) {
      // event.target.result == customerData[i].ssn
 };
}
```

Transactions have a well-defined lifetime, so attempting to use a transaction after it has completed throws exceptions.

Transactions also auto-commit and cannot be committed manually.

This transaction model is really useful when you consider what might happen if a user opened two instances of your web app in two different tabs simultaneously. Without transactional operations, the two instances could stomp all over each others modifications.

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 in the database, or "retrieve" a value out of the database through synchronous means. Instead, you "request" that a database operation happens. You are 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. This sounds a little complicated at first, but there are some sanity measures baked in. After all, you are a JavaScript programmer, aren't you? ;-)

So look at previous extracts of code: `transaction.oncomplete`, `transaction.onerror, request.onsuccess`, etc...

IndexedDB uses requests all over the place. Requests are objects that receive the success or failure DOM events that were mentioned previously. They have `onsuccess` and `onerror` properties, and you can call `addEventListener()` and `removeEventListener()` on them. They also have `readyState`, `result`, and `errorCode`properties that tell you the status of the request.

The `result` property is particularly magical, as it can be many different things, depending on how the request was generated (for example, an `IDBCursor` instance, or the key for a value that you just inserted into the database). We will see this in detail in the next page "Using IndexedDB".

IndexedDB uses DOM events to notify you when results are available. DOM events always have a `type` property (in IndexedDB, it is most commonly set to "`success`" or "`error`"). DOM events also have a `target` property that shows where the event is headed. In most cases, the `target` of an event is the `IDBRequest` object that was generated as a result of doing some database operation. Success events don't bubble up and they can't be cancelled. Error events, on the other hand, do bubble, and can be cancelled. This is quite important, as error events abort whatever transactions they're running in, unless they are cancelled.

IndexedDB is object-oriented. IndexedDB is not a relational database, which has tables with collections of rows and columns. This important and fundamental difference affects the way you design and build your applications, it is an Object Store!

In a traditional relational data store, you would have a table that stores a collection of rows of data and columns of named types of data. IndexedDB, on the other hand, requires you to create an object store for a type of data and simply persist JavaScript objects to that store. Each object store can have a collection of indexes (corresponding to the properties of the JavaScript object you store in the store) that makes it efficient to query and iterate across.

IndexedDB does not use Structured Query Language (SQL). It uses queries on an index that produces a cursor, which you use to iterate across the result set. If you are not familiar with NoSQL systems, read the Wikipedia article on NoSQL.

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. Each origin has its own associated set of databases. Every database has a name that identifies it within an origin. Think of it as "an application, a Database".

What defines "same origin" does not include the port or the protocol. For example, an app in a page with this URL`http://www.example.com/app/` and an app in a page with this URL `http://www.example.com/dir/` both can access the same IndexedDB database because they have the same origin (example.com), `http://www.example.com:8080/dir/ (different port)` or `https://www.example.com/dir/`(different protocol), cannot be considered of the same origin (port and protocol are different from `http://www.example.com`)