CODE > WEB DEVELOPMENT

Working With IndexedDB

by Raymond Camden   13 Sep 2013

Difficulty: Intermediate   Length: Quick   Languages: English

Web Development   IndexedDB   JavaScript

One of the more interesting developments in web standards lately is the Indexed Database (IndexedDB for short) specification. For a fun time you can read the spec yourself. In this tutorial I'll be explaining this feature and hopefully giving you some inspiration to use this powerful feature yourself.

Overview

> *As a specification, IndexedDB is currently a Candidate Recommendation.*

In a nutshell, IndexedDB provides a way for you to store large amounts of data on your user's browser. Any application that needs to send a lot of data over the wire could greatly benefit from being able to store that data on the client instead. Of course storage is only part of the equation. IndexedDB also provides a powerful indexed based searching API to retrieve the data you need.

You may wonder how IndexedDB differs from other storage mechanisms?

Cookies are extremely well supported, but have legal implications and limited

Are you a developer? Try out the HTML to PDF API

storage space. Also - they are sent back and forth to the server with every request, completely negating the benefits of client-side storage.

Local Storage is also very well supported, but limited in terms of the total amount of storage you can use. Local Storage doesn't provide a true "search" API as data is only retrieved via key values. Local Storage is great for "specific" things you may want to store, for example, preferences, whereas IndexedDB is better suited for Ad Hoc data (much like a database).

Before we go any further though, let's have an honest talk about the state of IndexedDB in terms of browser support. As a specification, IndexedDB is currently a Candidate Recommendation. At this point the folks behind the specification are happy with it but are now looking for feedback from the developer community. The specification may change between now and the final stage, W3C Recommendation. In general, the browsers that support IndexedDB now all do in a fairly consistent manner, but developers should be prepared to deal with prefixes and take note of updates in the future.

As for those browsers supporting IndexedDB, you've got a bit of a dilemma. Support is pretty darn good for the desktop, but virtually non-existent for

mobile. Let's see what the excellent site CanIUse.com says:

IndexedDB - Working Draft

Method of storing data client-side, allows indexed database queries. Previously known as WebSimpleDB API.

	IE	Firefox	Chrome	Safari	Opera	iOS Safari	Android Browser	Blackberry Browser	Chrome for Android
Show all versions							2.1		
							2.2		
						3.2	2.3		
						4.0-4.1	3.0		
	8.0					4.2-4.3	4.0		
	9.0	21.0	27.0	5.1		5.0-5.1	4.1	7.0	
Current	10.0	22.0	28.0	6.0	15.0	6.0-6.1	4.2	10.0^{webkit}	28.0
Near future	11.0	23.0	29.0	7.0	16.0	7.0			
Farther future		24.0							

Notes | Known issues (1) | Resources (5) | Feedback   Edit on GitHub

Partial support in BB10 refers to an outdated specification being implemented. Code targeting the current state of the specification might not work.

Chrome for Android does support the feature, but very few people are currently using that browser on Android devices. Does the lack of mobile support imply you shouldn't use it? Of course not! Hopefully all our readers are familiar with the concept of progressive enhancement. Features like IndexedDB can be added to your application in a manner that won't break in non-supported browsers. You could use wrapper libraries to switch to WebSQL on mobile, or simply skip storing data locally on your mobile clients. Personally I believe the ability to cache large blocks of data on the client is important enough to use now even without mobile support.

Let's Get Started

We've covered the specification and support, now let's look at using the feature. The very first thing we should do is check for IndexedDB support. While there are tools out there that provide generic ways to check for browser features, we can make this much simpler since we're just checking for one particular thing.

```
1   document.addEventListener("DOMContentLoaded", function(){
```

```
2
3    if("indexedDB" in window) {
4        console.log("YES!!! I CAN DO IT!!! WOOT!!!)";
5    } else {
6        console.log("I has a sad.");
7    }
8
9  },false);
```

The code snippet above (available in `test1.html` if you download the zip file
attached to this article) uses the `DOMContentLoaded` event to wait for the page
to load. (Ok, that's kind of obvious, but I recognize this may not be familiar to
folks who have only used jQuery.) I then simply see if indexedDB exists in the
`window` object and if so, we're good to go. That's the simplest example, but
typically we would probably want to store this so we know later on if we can
use the feature. Here's a slightly more advanced example (`test2.html`).

```
1    var idbSupported = false;
2
3    document.addEventListener("DOMContentLoaded", function(){
4
5        if("indexedDB" in window) {
6            idbSupported = true;
7        }
8
9    },false);
```

All I've done is created a global variable, `idbSupported`, that can be used as a flag to see if the current browser can use IndexedDB.

Opening a Database

IndexedDB, as you can imagine, makes use of databases. To be clear, this isn't a SQL Server implementation. This database is local to the browser and only available to the user. IndexedDB databases follow the same rules as cookies and local storage. A database is unique to the domain it was loaded from. So for example, a database called "Foo" created at foo.com will not conflict with a database of the same name at goo.com. Not only will it not conflict, it won't be available to other domains as well. You can store data for your web site knowing that another web site will not be able to access it.

Opening a database is done via the open command. In basic usage you provide a name and a version. The version is very important for reasons I'll cover more later. Here's a simple example:

Are you a developer? Try out the HTML to PDF API

```
1    var openRequest = indexedDB.open("test",1);
```

Opening a database is an asynchronous operation. In order to handle the result of this operation you'll need to add some event listeners. There's four different types of events that can be fired:

- success
- error
- upgradeneeded
- blocked

You can probably guess as to what success and error imply. The upgradeneeded event is used both when the user first opens the database as well as when you change the version. Blocked isn't something that will happen usually, but can fire if a previous connection was never closed.

Typically what should happen is that on the first hit to your site the upgradeneeded event will fire. After that - just the success handler. Let's look at a simple example (`test3.html`).

```
01   var idbSupported = false;
02   var db;
03
04   document.addEventListener("DOMContentLoaded", function(){
05
06       if("indexedDB" in window) {
07           idbSupported = true;
08       }
09
10       if(idbSupported) {
11           var openRequest = indexedDB.open("test",1);
12
13           openRequest.onupgradeneeded = function(e) {
14               console.log("Upgrading...");
15           }
16
17           openRequest.onsuccess = function(e) {
18               console.log("Success!");
19               db = e.target.result;
20           }
21
22           openRequest.onerror = function(e) {
23               console.log("Error");
24               console.dir(e);
25           }
26
27       }
28
29   },false);
```

Once again we check to see if IndexedDB is actually supported, and if it is, we

open a database. We've covered three events here - the upgrade needed event, the success event, and the error event. For now focus on the success event. The event is passed a handler via `target.result`. We've copied that to a global variable called `db`. This is something we'll use later to actually add data. If you run this in your browser (in one that supports IndexedDB of course!), you should see the upgrade and success message in your console the first time you run the script. The second, and so forth, times you run the script you should only see the success message.

Object Stores

So far we've checked for IndexedDB support, confirmed it, and opened a connection to a database. Now we need a place to store data. IndexedDB has a concept of "Object Stores." You can think of this as a typical database table. (It is much more loose than a typical database table, but don't worry about that now.) Object stores have data (obviously) but also a keypath and an optional set of indexes. Keypaths are basically unique identifiers for your data and come in a few different formats. Indexes will be covered later when we start talking about retrieving data.

Are you a developer? Try out the HTML to PDF API

Now for something crucial. Remember the upgradeneeded event mentioned before? You can only create object stores during an upgradeneeded event. Now - by default - this will run automatically the first time a user hits your site. You can use this to create your object stores. The crucial thing to remember is that if you ever need to modify your object stores, you're going to need to upgrade the version (back in that open event) and write code to handle your changes. Lets take a look at a simple example of this in action.

```
01   var idbSupported = false;
02   var db;
03
04   document.addEventListener("DOMContentLoaded", function(){
05
06       if("indexedDB" in window) {
07           idbSupported = true;
08       }
09
10       if(idbSupported) {
11           var openRequest = indexedDB.open("test_v2",1);
12
13           openRequest.onupgradeneeded = function(e) {
14               console.log("running onupgradeneeded");
15               var thisDB = e.target.result;
16
17               if(!thisDB.objectStoreNames.contains("firstOS")) {
18                   thisDB.createObjectStore("firstOS");
```

Are you a developer? Try out the HTML to PDF API

```
19                    }
20
21                }
22
23            openRequest.onsuccess =function(e) {
24                console.log("Success!");
25                db = e.target.result;
26            }
27
28            openRequest.onerror =function(e) {
29                console.log("Error");
30                console.dir(e);
31            }
32
33        }
34
35    },false);
```

This example (`test4.html`) builds upon the previous entries so I'll just focus on what's new. Within the upgradeneeded event, I've made use of the database variable passed to it (`thisDB`). One of the properties of this variable is a list of existing object stores called `objectStoreNames`. For folks curious, this is not a simple array but a "DOMStringList." Don't ask me - but there ya go. We can use the `contains` method to see if our object store exists, and if not, create it. This is one of the few synchronous functions in IndexedDB so we don't have to listen for the result.

To summarize then - this is what would happen when a user visits your site. The first time they are here, the upgradeneeded event fires. The code checks to see if an object store, "firstOS" exists. It will not. Therefore - it is created. Then the success handler runs. The second time they visit the site, the version number will be the same so the upgradeneeded event is not fired.

Now imagine you wanted to add a second object store. All you need to do is increment the version number and basically duplicate the contains/createObjectStore code block you see above. The cool thing is that your upgradeneeded code will support both people who are brand new to the site as well as those who already had the first object store. Here is an example of this (`test5.html`):

```
01   var openRequest = indexedDB.open("test_v2",2);
02
03   openRequest.onupgradeneeded = function(e) {
04       console.log("running onupgradeneeded");
05       var thisDB = e.target.result;
06
07       if(!thisDB.objectStoreNames.contains("firstOS")) {
08           thisDB.createObjectStore("firstOS");
09       }
10
```

```
11    if(!thisDB.objectStoreNames.contains("secondOS")) {
12        thisDB.createObjectStore("secondOS");
13    }
14
15  }
```

Adding Data

Once you've got your object stores ready you can begin adding data. This is - perhaps - one of the coolest aspects of IndexedDB. Unlike traditional table-based databases, IndexedDB lets you store an object as is. What that means is you can take a generic JavaScript object and just store it. Done. Obviously there's some caveats here, but for the most part, that's it.

Working with data requires you to use a transaction. Transactions take two arguments. The first is an array of tables you'll be working with. Most of the time this will be one table. The second argument is the type of transaction. There are two types of transactions: readonly and readwrite. Adding data will be a readwrite operation. Let's start by creating the transaction:

```
1    //Assume db is a database variable opened earlier
2    var transaction = db.transaction(["people"],"readwrite");
```

Note the object store, "people", is just one we've made up in the example above. Our next full demo will make use of it. After getting the transaction, you then ask it for the object store you said you would be working with:

```
1    var store = transaction.objectStore("people");
```

Now that you've got the store you can add data. This is done via the - wait for it - `add` method.

```
1    //Define a person
2    var person = {
3        name:name,
4        email:email,
5        created:new Date()
6    }
7
8    //Perform the add
9    var request = store.add(person,1);
```

Remember earlier we said that you can store any data you want (for the most part). So my person object above is completely arbitrary. I could have used

firstName and lastName instead of just name. I could have used a gender property. You get the idea. The second argument is the key used to uniquely identify the data. In this case we've hard coded it to 1 which is going to cause a problem pretty quickly. That's ok - we'll learn how to correct it.

The add operation is ascynchronous, so lets add two event handlers for the result.

```
1   request.onerror =function(e) {
2       console.log("Error",e.target.error.name);
3       //some type of error handler
4   }
5
6   request.onsuccess =function(e) {
7       console.log("Woot! Did it");
8   }
```

We've got an `onerror` handler for errors and `onsuccess` for good changes. Fairly obvious, but let's see a complete example. You can find this in the file `test6.html`.

>

```html
01   <!doctype html>
02   <html>
03   <head>
04   </head>
05
06   <body>
07
08   <script>
09   var db;
10
11   function indexedDBOk() {
12       return "indexedDB" in window;
13   }
14
15   document.addEventListener("DOMContentLoaded", function() {
16
17       //No support? Go in the corner and pout.
18       if(!indexedDBOk) return;
19
20       var openRequest = indexedDB.open("idarticle_people",1);
21
22       openRequest.onupgradeneeded = function(e) {
23           var thisDB = e.target.result;
24
25           if(!thisDB.objectStoreNames.contains("people")) {
26               thisDB.createObjectStore("people");
27           }
28       }
29
30       openRequest.onsuccess = function(e) {
31           console.log("running onsuccess");
32
```

```
33          db = e.target.result;
34
35          //Listen for add clicks
36          document.querySelector("#addButton").addEventListener("click"
37      }
38
39      openRequest.onerror = function(e) {
40          //Do something for the error
41      }
42
43  },false);
44
45  function addPerson(e) {
46      var name = document.querySelector("#name").value;
47      var email = document.querySelector("#email").value;
48
49      console.log("About to add "+name+"/"+email);
50
51      var transaction = db.transaction(["people"],"readwrite");
52      var store = transaction.objectStore("people");
53
54      //Define a person
55      var person = {
56          name:name,
57          email:email,
58          created:new Date()
59      }
60
61      //Perform the add
62      var request = store.add(person,1);
63
64      request.onerror = function(e) {
```

```
65            console.log("Error",e.target.error.name);
66            //some type of error handler
67        }
68
69      request.onsuccess = function(e) {
70            console.log("Woot! Did it");
71        }
72    }
73    </script>
74
75    <input type="text" id="name" placeholder="Name"><br/>
76    <input type="email" id="email" placeholder="Email"><br/>
77    <button id="addButton">Add Data</button>
78
79    </body>
80    </html>
```

The example above contains a small form with a button to fire off an event to store the data in IndexedDB. Run this in your browser, add something to the form fields, and click add. If you've got your browser dev tools open, you should see something like this.
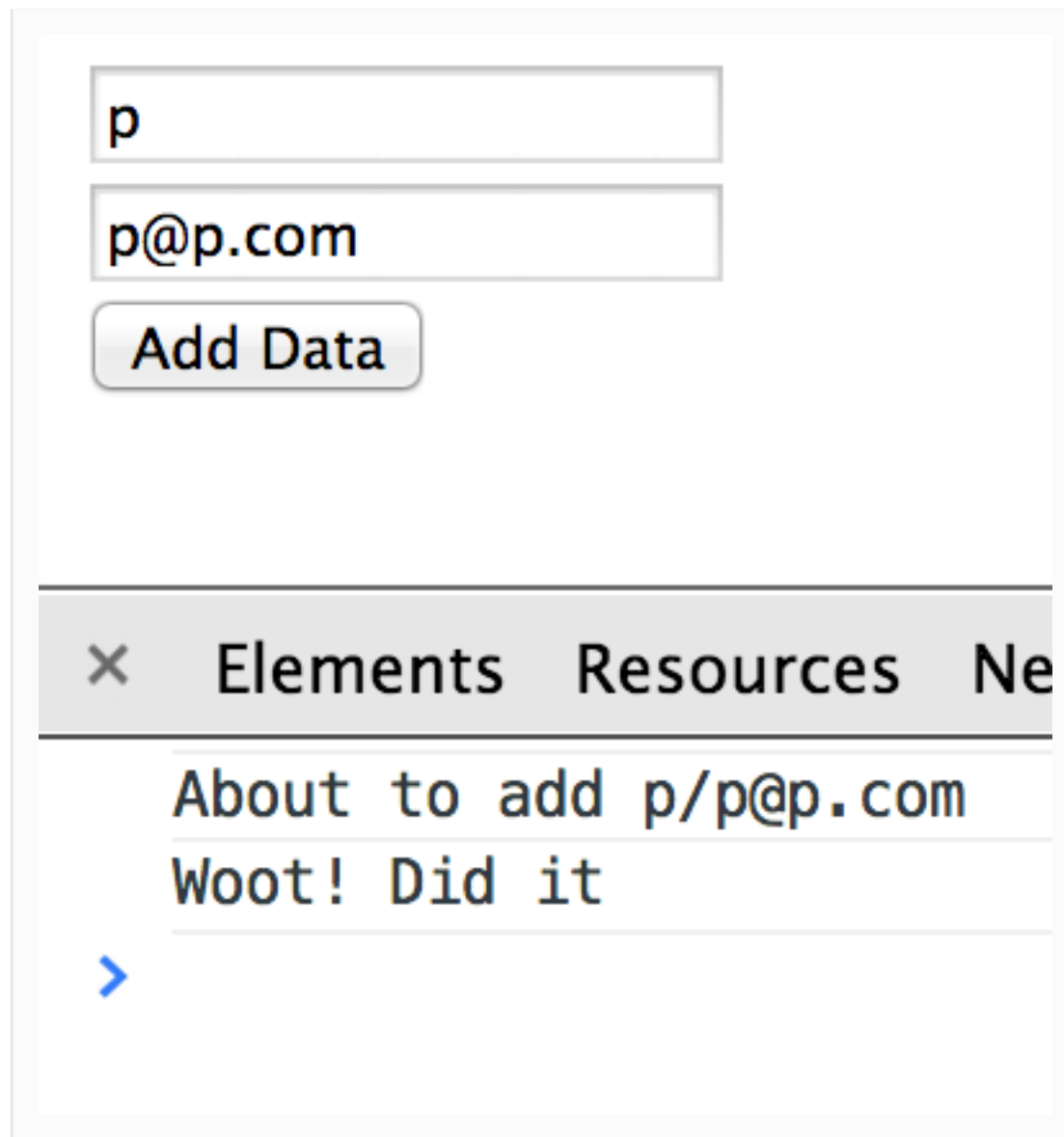
This is a great time to point out that Chrome has an excellent viewer for IndexedDB data. If you click on the Resources tab, expand the IndexedDB section, you can see the database created by this demo as well as the object

just entered.
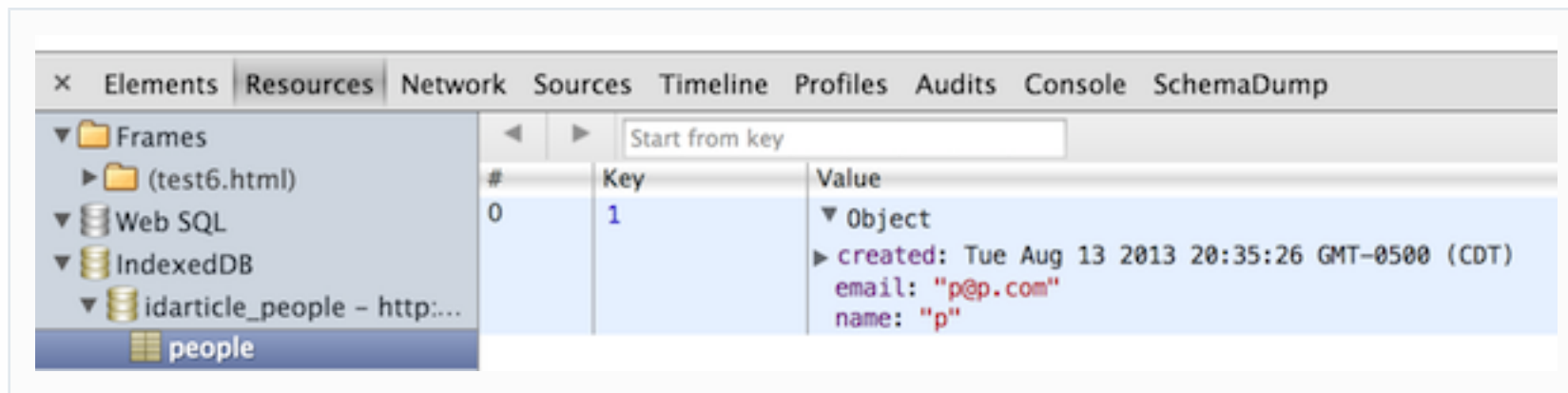
For the heck of it, go ahead and hit that Add Data button again. You should see an error in the console:

```
×    Elements   Resources   Network

About to add p/p@p.com
Woot! Did it
About to add p/p@p.com
Error ConstraintError
>  |
```
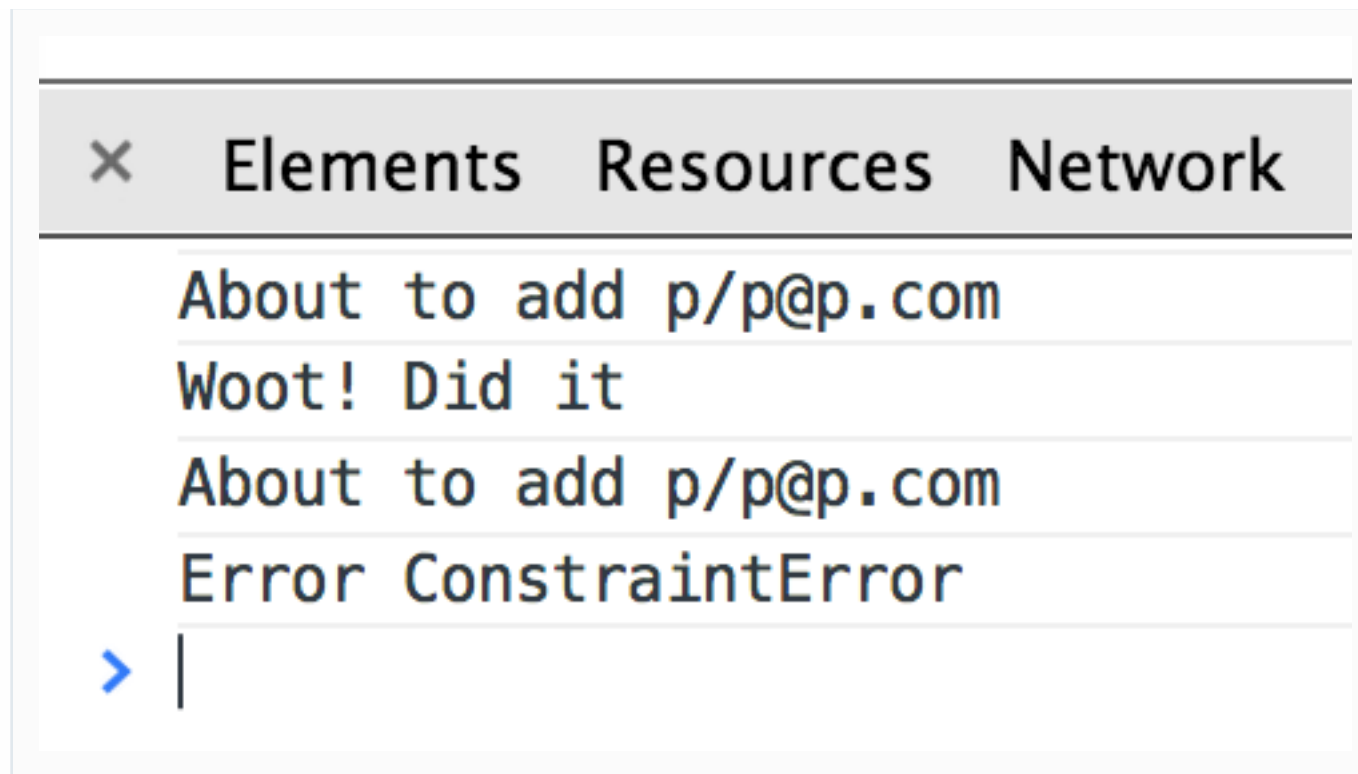
The error message should be a clue. ConstraintError means we just tried to add data with the same key as one that already existed. If you remember, we hard coded that key and we knew that was going to be a problem. It's time to talk keys.

Keys

Are you a developer? Try out the HTML to PDF API

Keys are IndexedDB's version of primary keys. Traditional databases can have tables without keys, but every object store needs to have a key. IndexedDB allows for a couple different types of keys.

The first option is to simply specify it yourself, like we did above. We could use logic to generate unique keys.

Your second option is a keypath, where the key is based on a property of the data itself. Consider our people example - we could use an email address as a key.

Your third option, and in my opinion, the simplest, is to use a key generator. This works much like an autonumber primary key and is the simplest method of specifying keys.

Keys are defined when object stores are created. Here are two examples - one using a key path and one a generator.

```
1   thisDb.createObjectStore("test", { keyPath: "email" });
2   thisDb.createObjectStore("test2", { autoIncrement:true });
```

We can modify our previous demo by creating an object store with an
autoIncrement key:

```
1    thisDB.createObjectStore("people", {autoIncrement true});
```

Finally, we can take the Add call we used before and remove the hard coded
key:

```
1    var request = store.add(person);
```

That's it! Now you can add data all day long. You can find this version in
`test7.html`.

Reading Data

Now let's switch to reading individual pieces of data (we'll cover reading larger
sets of data later). Once again, this will be done in a transaction and will be
asynchronous. Here's a simple example:

```
1   var transaction = db.transaction(["test"], "readonly");
2   var objectStore = transaction.objectStore("test");
3
4   //x is some value
5   var ob = objectStore.get(x);
6
7   ob.onsuccess = function(e) {
8
9   }
```

Note that the transaction is read only. The API call is just a simple get call with the key passed in. As a quick aside, if you think using IndexedDB is a bit verbose, note you can chain many of those calls as well. Here's the exact same code written much tighter:

```
1   db.transaction(["test"], "readonly").objectStore("test").get(X).onsuccess
```

Personally I still find IndexedDB a bit complex so I prefer the 'broken out' approach to help me keep track of what's going on.

The result of the get's onsuccess handler is the object you stored before. Once you have that object you can do whatever you want. In our next demo (`test8.html`) we've added a simple form field to let you enter a key and print

the result. Here is an example:

raymondcamden@gma

Add Data

1 ⬍

Get Data

Key 1

name=Ray
email=raymondcamden@gmail.com
created=Tue Aug 13 2013 21:25:49 GMT-0500 (CDT)

×   Elements   Resources   Network   Sources   Timeline

▶ undefined

About to add Ray/raymondcamden@gmail.com

```
Woot! Did it
▶ Object
```

The handler for the Get Data button is below:

```
01  function getPerson(e) {
02      var key = document.querySelector("#key").value;
03      if(key === "" || isNaN(key)) return;
04
05      var transaction = db.transaction(["people"],"readonly");
06      var store = transaction.objectStore("people");
07
08      var request = store.get(Number(key));
09
10      request.onsuccess = function(e) {
11
12          var result = e.target.result;
13          console.dir(result);
14          if(result) {
15              var s = "&lt;h2>Key "+key+"&lt;/h2>&lt;p>";
16              for(var field in result) {
17                  s+= field+"="+result[field]+"&lt;br/>";
18              }
19              document.querySelector("#status").innerHTML = s;
20          } else {
21              document.querySelector("#status").innerHTML = "&lt;h2>No ma
22          }
```

```
23            }
24      }
```

For the most part, this should be self explanatory. Get the value from the field and run a get call on the object store obtained from a transaction. Notice that the display code simply gets all the fields and dumps them out. In a real application you would (hopefully) know what your data contains and work with specific fields.

Reading More Data

So that's how you would get one piece of data. How about alot of data? IndexedDB has support for what's called a cursor. A cursor lets you iterate over data. You can create cursors with an optional range (a basic filter) and a direction.

As an example, the following code block opens a cursor to fetch all the data from an object store. Like everything else we've done with data this is asynchronous and in a transaction.

```
01   var transaction = db.transaction(["test"], "readonly");
02   var objectStore = transaction.objectStore("test");
03
04   var cursor = objectStore.openCursor();
05
06   cursor.onsuccess = function(e) {
07       var res = e.target.result;
08       if(res) {
09           console.log("Key", res.key);
10           console.dir("Data", res.value);
11           res.continue();
12       }
13   }
```

The success handler is passed a result object (the variable res above). It contains the key, the object for the data (in the value key above), and a continue method that is used to iterate to the next piece of data.

In the following function, we've used a cursor to iterate over all of the objectstore data. Since we're working with "person" data we've called this getPeople:

```
01   function getPeople(e) {
02
03       var s = "";
04
```

```
05      db.transaction(["people"], "readonly").objectStore("people").openCur
06          var cursor = e.target.result;
07          if(cursor) {
08              s += "&lt;h2>Key "+cursor.key+"&lt;/h2>&lt;p>";
09              for(var field in cursor.value) {
10                  s+= field+"="+cursor.value[field]+"&lt;br/>";
11              }
12              s+="&lt;/p>";
13              cursor.continue();
14          }
15          document.querySelector("#status2").innerHTML = s;
16      }
17  }
```

You can see a full demo of this in your download as file `test9.html`. It has an Add Person logic as in the earlier examples, so simply create a few people and then hit the button to display all the data.

So now you know how to get one piece of data as well as how to get all the data. Let's now hit our final topic - working with indexes.

They Call This IndexedDB, Right?

We've been talking about IndexedDB for the entire article but haven't yet actually done any - well - indexes. Indexes are a crucial part of IndexedDB object stores. They provide a way to fetch data based on their value as well as specifying if a value should be unique within a store. Later we'll demonstrate how to use indexes to get a range of data.

First - how do you create an index? Like everything else structural, they must be done in an upgrade event, basically at the same time you create your object store. Here is an example:

```
1    var objectStore = thisDb.createObjectStore("people",
2                       { autoIncrement true });
3    //first arg is name of index, second is the path (col);
4    objectStore.createIndex("name","name", {unique:false});
5    objectStore.createIndex("email","email", {unique:true});
```

In the first line we create the store. We take that result (an objectStore object) and run the `createIndex` method. The first argument is the name for the index and the second is the property that will be indexed. In most cases I think you will use the same name for both. The final argument is a set of options. For now, we're just using one, unique. The first index for name is not unique. The second one for email is. When we store data, IndexedDB will check these indexes and ensure that the email property is unique. It will also do some data handling on the back end to ensure we can fetch data by these indexes.

How does that work? Once you fetch an object store via a transaction, you can then ask for an index from that store. Using the code above, here is an example of that:

```
1    var transaction = db.transaction(["people"],"readonly");
2    var store = transaction.objectStore("people");
3    var index = store.index("name");
4
5    //name is some value
6    var request = index.get(name);
```

First we get the transaction, followed by the store, and then index. As we've said before, you could chain those first three lines to make it a bit more compact if you want.

Once you've got an index you can then perform a `get` call on it to fetch data by name. We could do something similar for email as well. The result of that call is yet another asynchronous object you can bind an onsuccess handler to. Here is an example of that handler found in the file `test10.html`:

```
01    request.onsuccess =function(e) {
02
03        var result = e.target.result;
04        if(result) {
05            var s = "&lt;h2>Name "+name+"&lt;/h2>&lt;p>";
06            for(var field in result) {
07                s+= field+"="+result[field]+"&lt;br/>";
08        }
```

```
09              document.querySelector("#status").innerHTML = s;
10      } else {
11              document.querySelector("#status").innerHTML = "&lt;h2>No match&
12      }
13  }
```

Note that an index `get` call may return multiple objects. Since our name is not unique we should probably modify the code to handle that, but it isn't required.

Now let's kick it up a notch. You've seen using the get API on the index to get a value based on that property. What if you want to get a more broad set of data? The final term we're going to learn today are Ranges. Ranges are a way to select a subset of an index. For example, given an index on a name property, we can use a range to find names that begin with A up to names that begin with C. Ranges come in a few different varieties. They can be "everything below some marker", "everything above some marker", and "something between a lower marker and a higher marker." Finally, just to make things interesting, ranges can be inclusive or exclusive. Basically that means for a range going from A-C, we can specify if we want to include A and C in the range or just the values between them. Finally, you can also request both ascending and descending ranges.

Ranges are created using a toplevel object called IDBKeyRange. It has three methods of interest: `lowerBound`, `upperBound`, and `bound`. `lowerBound` is used to create a range that starts at a lower value and returns all data "above" it. `upperBound` is the opposite. And - finally - `bound` is used to support a set of data with both a lower and upper bound. Let's look at some examples:

```
01   //Values over 39
02   var oldRange = IDBKeyRange.lowerBound(39);
03
04   //Values 40a dn over
05   var oldRange2 = IDBKeyRange.lowerBound(40,true);
06
07   //39 and smaller...
08   var youngRange = IDBKeyRange.upperBound(40);
09
10   //39 and smaller...
11   var youngRange2 = IDBKeyRange.upperBound(39,true);
12
13   //not young or old... you can also specify inclusive/exclusive
14   var okRange = IDBKeyRange.bound(20,40)
```

Once you have a range, you can pass it to an index's openCursor method. This gives you an iterator to loop over the values that match that range. As a practical manner, this isn't really a search per se. You can use this to search content based on the beginning of a string, but not the middle or end. Let's

Are you a developer? Try out the HTML to PDF API

look at a full example. First we'll create a simple form to search people:

```
1   Starting with: <input type="text" id="nameSearch" placeholder="Name"><br/>
2   Ending with: <input type="text" id="nameSearchEnd" placeholder="Name"><br/
3   <button id="getButton">Get By Name Range</button>
```

We're going to allow for searches that consist of any of the three types of ranges (again, a value and higher, a highest value, or the values within two inputs). Now let's look at the event handler for this form.

```
01   function getPeople(e) {
02       var name = document.querySelector("#nameSearch").value;
03
04       var endname = document.querySelector("#nameSearchEnd").value;
05
06       if(name == "" &amp;&amp; endname =="") return;
07
08       var transaction = db.transaction(["people"],"readonly");
09       var store = transaction.objectStore("people");
10       var index = store.index("name");
11
12       //Make the range depending on what type we are doing
13       var range;
14       if(name != "" &amp;&amp; endname !="") {
15           range = IDBKeyRange.bound(name, endname);
16       } else if(name == "") {
17           range = IDBKeyRange.upperBound(endname);
```

Are you a developer? Try out the HTML to PDF API

```javascript
18      } else {
19          range = IDBKeyRange.lowerBound(name);
20      }
21
22      var s = "";
23
24      index.openCursor(range).onsuccess =function(e) {
25          var cursor = e.target.result;
26          if(cursor) {
27              s += "&lt;h2>Key "+cursor.key+"&lt;/h2>&lt;p>";
28              for(var field in cursor.value) {
29                  s+= field+"="+cursor.value[field]+"&lt;br/>";
30              }
31              s+="&lt;/p>";
32              cursor.continue();
33          }
34          document.querySelector("#status").innerHTML = s;
35      }
36
37  }
```

From top to bottom - we begin by grabbing the two form fields. Next we create a transaction and from that get the store and index. Now for the semi-complex part. Since we have three different types of ranges we need to support we have to do a bit of conditional logic to figure out which we'll need. What range we create is based on what fields you fill in. What's nice is that once we have the range, we then simply pass it to the index and open the cursor. That's it! You can find this full example in `test11.html`. Be sure to

enter some values first so you have data to search.

What's Next?

Believe it or not - we've only begun our discussion on IndexedDB. In the next article, we'll cover additional topics, including updates and deletes, array based values, and some general tips for working with IndexedDB.

Raymond Camden

Raymond Camden is a senior developer evangelist for Adobe. His work focuses on web standards, mobile development and ColdFusion. He's a published author and presents at conferences and user groups on a variety of topics. He loves kittens and Star Wars - not necessarily in that order.

Download Attachment

Are you a developer? Try out the HTML to PDF API

pdfcrowd.com

Translations

Envato Tuts+ tutorials are translated into other languages by our community members—you can be involved too!

Translate this post

Powered by native

79 Comments   **Nettuts+**

❤ **Recommend** 7   ↱ **Share**

Join the discussion…

Renan • 3 years ago

Great article, and great explanation about. Thanks a lot!

9 ∧ | ∨ • Reply • Share ›

kuhnlud • 3 years ago

Great! many thank! There isn't a lot of tutorial about Datastorage in Browser! A question.
clear the cache from the Browser, will I clear the DB?

3 ∧ | ∨ • Reply • Share ›

Raymond Camden ➤ kuhnlud • 3 years ago

Last time I tested, clearing the cache did NOT kill IndexedDB data. In fact, there w
user to do it - unless they went into console and it manually. I've added this to my
cover though.

2 ∧ | ∨ • Reply • Share ›

Ian Simmons ➤ Raymond Camden • 3 years ago

Good to know as a developer. As a user who is always cleaning up history
would want to know I had a way to clean up indexedDB. I suppose develop
would need to account for this by having the local data rebuild from the on
delete it locally.

∧ | ∨ • Reply • Share ›

Raymond Camden ➤ Ian Simmons • 3 years ago

Well in theory, if the user killed his db, then the code you write for f
again, right? What I mean is - there is no difference between a firs
intentionally deleted their db store.

Ian Simmons → Raymond Camden • 3 years ago

Right, you put it better but that's the concept I was thinking. Just wa
actually done anything with local storage of any kind but am seriou
Your article is very helpful and informative. Thanks :-)

^ | ∨ • Reply • Share ›

Raymond Camden → Ian Simmons • 3 years ago

If you haven't looked at _anything_ in this space, also check out Lo
simpler, and better for small atomic pieces of data (for example, st
preferences). Here is a presentation I did about a year or so ago. It
storage ideas: http://www.raymondcamden.com/i...

^ | ∨ • Reply • Share ›

kuhnlud → Raymond Camden • 3 years ago

Thank for the answer, And what about localstorage? any chance to
cache clear?

1 ^ | ∨ • Reply • Share ›

Raymond Camden → kuhnlud • 3 years ago

I believe it is the same - you can't clear it via the 'normal' UI but can
(To be clear, my confidence isn't super high on this.)

^ | ∨ • Reply • Share ›

Ian • 3 years ago

Unfortunately IndexedDB doesn't seem to support full-text search or multi-index queries, has rather complicated syntax, so I propose to use a simple and cross-browser wrapper http://aaronpowell.github.io/d...

2 ^ | ˅ • Reply • Share ›

Chris Emerson • 3 years ago

Thanks for this article Raymond - looking forward to part 2. By the way - any chance you' jquery library Parashuram made for IndexedDB usage? I'm still getting my head around tl method to use it but am hoping soon after I get the jist of things I can move on to using th so much easier to read/write (for me anyhow).

https://github.com/axemclion/j...

1 ^ | ˅ • Reply • Share ›

Raymond Camden ➜ Chris Emerson • 3 years ago

Part 2 did launch: http://net.tutsplus.com/tutori...

And nope - haven't seen that library.

^ | ˅ • Reply • Share ›

Janne Leppänen • 3 years ago

Looking awesome so far! Thanks Raymond. :-)
I must be really slow, because it took at least 60 minutes for me to complete this one. Sti case scenarios where indexedDB would shine best. Will applications still need another d IndexedDB, because data is only stored to cache?

1 ^ | ˅ • Reply • Share ›

Raymond Camden → Janne Leppänen • 3 years ago

First off - I have no idea where the 20 minute estimate came from. Blame the edit
far more accurate. ;)

"Will applications still need another database the side of IndexedDB, because dat
depends. :) For example, I wrote a demo once where I used IndexedDB to store a
add WebSQL support. My thinking was that if you support IndexedDB you get the
It didn't break the app, just made it better for modern browsers. It is my demo so I
works well: http://www.raymondcamden.com/i...

1 ∧ | ∨ • Reply • Share ›

ShahzadSeraj • a year ago

how i can get the latest key path from indexedDB

∧ | ∨ • Reply • Share ›

Raymond Camden → ShahzadSeraj • 10 months ago

When you add an object, use the onsuccess handler to get the key.

∧ | ∨ • Reply • Share ›

Mayank Tripathi • 2 years ago

This is new in the HTML 5 specification. It is not the same as a relational database. By us
store a large number of objects locally. This is a new JavaScript API that is offered by HT
a specification for an indexed database that is present within our browser. As I said, it is i
database so it does not have a table, rows and columns like a relational database.

In a relational database, to store data we write a database query, like:

insert in <table_name>([column_1], [column_2]......[column_n]) values ([val_1];[val_2];....

The same as for select, update and delete we have different queries that we also have us
Database. But as I said, an IndexedDB stores objects. So indexedDB has a different way
In this first you create an object to store a type of data; simply stated, we store JavaScrip
have a simple value (like string, date and so on) or hierarchical objects (like JavaScript ol
consists of a key with its corresponding value and each object stores a collection of inde
query and iteration can be fast across objects. In an IndexedDB the query is replaced by
produces a cursor to be used to iterate across the result set.

The IndexedDB API is exposed through the window.indexedDB object. When you work o
following lines of code you should always use, to determine whether or not the browser s

see more

⌃ | ⌄ • Reply • Share ›

Chris Eilander • 2 years ago

I had a problem at the point of using the autoincrement stuff. It gives me the following err

ncaught DataError: Failed to execute 'add' on 'IDBObjectStore': The object store uses ou
key generator and the key parameter was not provided.

⌃ | ⌄ • Reply • Share ›

Raymond Camden ➜ Chris Eilander • 2 years ago

Double check your code. I just ran test7.html again and had no error. I split it up s
CodePen: http://codepen.io/cfjedimaster...

⌃ | ⌄ • Reply • Share ›

Chris Eilander → Raymond Camden • 2 years ago

I did some more changes. It was my
var openRequest = indexedDB.open("idarticlepeople",1); changed the nam
autoincrementing with an existing database that wasn't created with was t

∧ | ∨ • Reply • Share ›

Raymond Camden → Chris Eilander • 2 years ago

Hmm - in my article I tried to use unique names so each one was
forgotten to do so. If so - sorry!

∧ | ∨ • Reply • Share ›

Chris Eilander → Raymond Camden • 2 years ago

No, that was my fault for not checking for that. I followed your tut a
so I could try it out myself instead of just looking at code. Since i al
part i wasn't looking anymore. Sorry for bothering you with this mis

∧ | ∨ • Reply • Share ›

AttitudeMonger • 2 years ago

Hi,

So imagine I have large bodies of text, say movie reviews. I want a database search bas
incomplete phrases in the collection of all reviews. Say, there are 200 reviews. Each revi
id. Some of them contain the text "is the death of duty". Now if I search by the text "death
whose review contains that text somewhere. Can that be implemented? Will it be fast en
some search engines like Lucene offer?

∧ | ∨ • Reply • Share ›

Raymond Camden ➜ AttitudeMonger • 2 years ago

No. So one of the big things missing from the *current* spec is the ability to searc
next version of the spec will address this.

⌃ | ⌄ • Reply • Share ›

AttitudeMonger ➜ Raymond Camden • 2 years ago

Any idea when the next spec is coming?

⌃ | ⌄ • Reply • Share ›

Raymond Camden ➜ AttitudeMonger • 2 years ago

Not for a while I'd guess. :\

⌃ | ⌄ • Reply • Share ›

AttitudeMonger ➜ Raymond Camden • 2 years ago

:\

⌃ | ⌄ • Reply • Share ›

Marcus Ang • 2 years ago

I am wondering if there's already a part 2 of this article **@Raymond Camden**

⌃ | ⌄ • Reply • Share ›

Raymond Camden ➜ Marcus Ang • 2 years ago

Part 2: http://net.tutsplus.com/tutori...

Part 3: http://net.tutsplus.com/tutori...

2 ⌃ | ⌄ • Reply • Share ›

Majid Lotfi • 2 years ago

Thanks, nice explanation.

But I am still confused, do we need IndexedDB to cache data coming from datatabase se
data when we are offline ?

if a company (for example travel agency) has a web site that is using IndexedDB, after th
is in his machine, BUT what about the company server ? how they will know about all the

please bear with me if I misunderstood the concept of why we need this IndexedDB ?
Thanks.

∧ | ∨ • Reply • Share ›

> **Raymond Camden** → Majid Lotfi • 2 years ago
> You don't *need* to use IndexedDB, it is an option. And yes, it would let you store
> offline.
>
> ∧ | ∨ • Reply • Share ›

NaMeLeSs • 2 years ago

Big THANK YOU!

∧ | ∨ • Reply • Share ›

M.A. • 3 years ago

Thank you for this wonderful set of articles on IndexedDB Raymond! I am finding them ve
understand!

variable), but I keep getting "DataCloneError: The object could not be cloned." I have no is
or opening it, but the add step is not working. Do you know what causes this?

Regards,
Mohamad Awada

∧ | ∨ • Reply • Share ›

Raymond Camden ➔ M.A. • 3 years ago

I've never encountered that error before. Is your app online where I can see?

∧ | ∨ • Reply • Share ›

M.A. ➔ Raymond Camden • 3 years ago

Actually, I haven't put it online yet because I am developing it with the offlin
the app with and without the webserver on, and it is still giving me the sam
I uploaded the code onto google drive, and this is the link: https://drive.goo
I am sure it is entering the addStore function (line 175) and that it is catchi
modular form, and I am creating the variable storeinfo in the function as no
the modular form. However, the store.add(storeinfo,1) on line 199 is creati
it on Firefox 27.0.1 and Chrome v32, with the same error on both browser
is not helping much regarding the error...

∧ | ∨ • Reply • Share ›

Raymond Camden ➔ M.A. • 3 years ago

Maybe it's the language - but I'm unable to replicate your issue. Yo
an add immediately - but in my testing, addStore didn't set holder t
add isn't run for me. If you can either simplify your code or tell me

M.A. ➜ Raymond Camden • 3 years ago

Ah I am sorry for not making it clearer. Holder is set to 1 after the b
pressed on the modal form. So addStore is always going to be ent
the button is not pressed, it will be timed out.

⌃ | ⌄ • Reply • Share ›

Raymond Camden ➜ M.A. • 3 years ago

I don't have a modal form - I've got... well a bunch of stuff all over tl
can't replicate this. Can you please make a new demo, MUCH sim
demonstrate this as it is not working for me at all right now.

⌃ | ⌄ • Reply • Share ›

M.A. ➜ Raymond Camden • 3 years ago

I actually found out what was wrong when I was trying to make it si
you. I was passing the input form's object as is, and that was what
DataCloneError. So apparently, IndexedDB can clone Objects but
composed of Objects.
Thank you so much for your time! And again, thank you for the wol
on IndexedDB!

⌃ | ⌄ • Reply • Share ›

Raymond Camden ➜ M.A. • 3 years ago

Man - I wish I had said something. I had an idea that was it, but wa
think I saw something similar with WebSQL once. Anyway, glad yc

baroquedub • 3 years ago

All very useful, thanks. I've a question re. the Chrome browser's viewer for IndexedDB da
article. I've found that it doesn't update in real time. i.e. if you programmatically make cha
reflected in what you see. It always displays the value of when you first opened the 'Reso
tools. Even if you right click and select the refresh option, the data shown still shows the

Can anyone recommend another option? (logging values to the console obviously works
have a real time viewer for all of the database data.

∧ | ∨ • Reply • Share ›

Raymond Camden ➤ baroquedub • 3 years ago

I'm not seeing that. I've got the latest Chrome (32.0.1700.102) and when I click the

1 ∧ | ∨ • Reply • Share ›

baroquedub ➤ Raymond Camden • 3 years ago

Hi Raymond, thanks for the prompt reply.

I've just updated to the latest Chrome and still getting the same problem. H
http://screencast.com/t/SGP3MV...

I go through the process twice - starting in the Resources view to show ho
after a refresh), then the second time (after I refresh the page) my initial vi
only after the update that I open the Resources Indexeddb tab, and hey pr

Strange!

∧ | ∨ • Reply • Share ›

Raymond Camden → baroquedub • 3 years ago

That's the wrong one. I believe it just refreshes the dbs/objectstore
one below. See screenshot.

https://www.evernote.com/shard...

1 ⌃ | ⌄ • Reply • Share ›

baroquedub → Raymond Camden • 3 years ago

You're a star! Many thanks indeed.

⌃ | ⌄ • Reply • Share ›

apple • 3 years ago

Thanks for such wonderfull document.. But how can i delete or update entries?

⌃ | ⌄ • Reply • Share ›

Raymond Camden → apple • 3 years ago

Please see the follow up entries.

⌃ | ⌄ • Reply • Share ›

JSNovice • 3 years ago

Thanks for the article raymond.

I have a doubt in javascript.
I have the below code. How to access the global variable/update the global variable inside
Please clarify

```
function setupEmployee() {
var setupEmpTransaction = db.transaction([storeName], "readwrite");
var empDataStore = setupEmpTransaction.objectStore(storeName);
var emp1 = {ld:101, name: "King", age: "30", destination: "programmer", employer: "cts"};
var emp2 = {ld:102, name: "Queen", age: "30", destination: "programmer", employer: "cts
var emp3 = {ld:103, name: "Bishop", age: "30", destination: "programmer", employer: "cts
var emp4 = {ld:104, name: "Horse", age: "30", destination: "senior associate", employer: "
var emp5 = {ld:105, name: "Rook", age: "30", destination: "senior associate", employer: "
var emp6= {ld:106, name: "Pawn", age: "30", destination: "senior associate", employer: "

var employees = [emp1, emp2, emp3, emp4, emp5, emp6];

for (var i = 0; i < employees.length; i++) {
```

see more

∧ | ∨ • Reply • Share ›

Wasim Khan • 3 years ago

Explained Clearly. Helpful document

∧ | ∨ • Reply • Share ›

Александр Бережник • 3 years ago

Thanks & BIG hello from Ukraine!

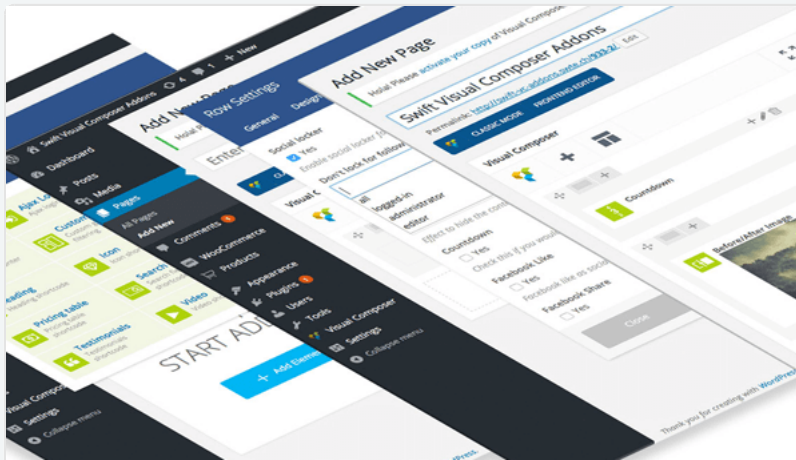∧ | ∨ • Reply • Share ›

Load more comments

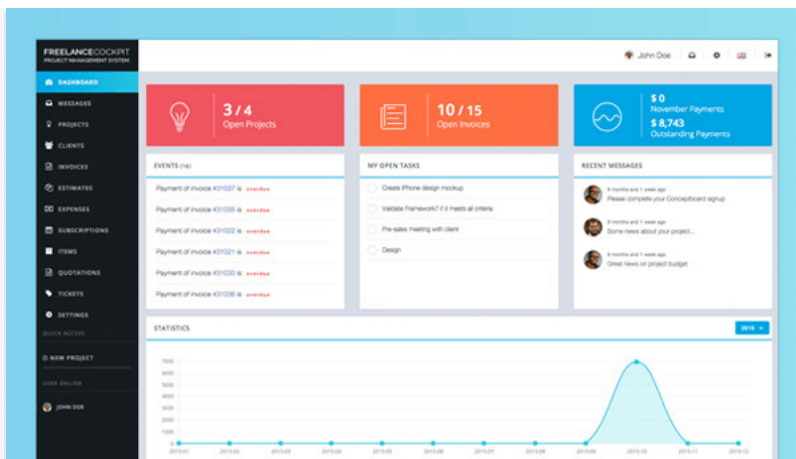Are you a developer? Try out the HTML to PDF API

Looking for something to help kick start your next project?

Envato Market has a range of items for sale to help get you started.

WordPress Plugins

From $4

Are you a developer? Try out the HTML to PDF API pdfcrowd.com

PHP Scripts

From $1

Teaching skills to millions worldwide.

Video

22,497 Tutorials   **900** Courses

Meet Envato

About Envato

Explore our Ecosystem

Careers

Join our Community

Teach at Envato Tuts+

Translate for Envato Tuts+

Forums

Community Meetups

Help and Support

FAQ

Help Center

Terms of Use

About Envato Tuts+

Advertise

Email Newsletters

Get Envato Tuts+ updates, news, surveys & offers.

Email Address

Subscribe

Privacy Policy

Build anything from social networks to file upload systems. Build faster with pre-coded PHP scripts.

Browse PHP on CodeCanyon

Follow Envato Tuts+