webOS: Palm's Game-Changing Mobile Operating System – Part 2

Extend your knowledge of webOS further by building a complete application with widgets, services and more!

Frank W. Zammetti fzammetti@etherient.com http://www.zammetti.com

In the last issue of JSMag, part 1 of this article introduced the basis of Palm's webOS. You learned how to set up your development environment for webOS and created a very simple starter application. In this second part we'll expand upon that new-found knowledge by building the first half of a simple but realistic application, namely a note-taking application. This segment of the webOS series will expose us to some of the database abilities of the platform and begin to show us some Mojo framework dialog calls as well.

From "Hello, World!" to "Note to self: Say Hello, World!"

Last month, in the first part of this article, you were introduced to webOS by way of the oldest trick in the programmers' book, the "Hello, World!" application. It was by no stretch of the imagination anything complex, but it succinctly demonstrated the basics: from the concepts of stages and scenes to the Mojo framework and all stops in between. We also looked at a typical webOS application directory structure and, of course, the development tools to bring it all together.

This month, we're going to create a more robust application that touches on a few more areas of webOS development. This isn't going to be an Earth-shatteringly complex application and I wouldn't expect it to win any awards, but it will quite nicely build upon part 1 and get you further down the webOS trail.

Specifically, this application will allow the user to create and store notes. It's a place where you can enter some notes, store them locally, and be able to see a list of the notes you've created and view any of them if you wish (as well as being able to delete them too, of course). The Palm Pre, and all webOS-based devices, ship with a pretty similar note application that uses a caulk board metaphor, which looks very nice on the screen, but some find a little annoying to use and prefer a simple list of notes instead. So, we'll be giving those people what they want here!

Note: I am assuming from here on out that you've actually read part 1 from last month's issue of JSMag. If you haven't, you will probably be lost here, and you may want to go back to read it before proceeding here with part 2.

Designing the application

When you are designing a web site, a common way to do so is to get some folks into a room with a lot of whiteboards in it. Then, all present can begin to scribble out a rough outline of the pages that will make up the site and the flow through them. Some people actually do this with construction paper, pushpins and string, but either way it's the same basic idea: a low-fidelity model of the pages that make up the site, and how the user will navigate through them.

A webOS application, being essentially a web application that just happens to run on a local device, can be modeled the same way. For complex applications this is almost a must, but for something as simple as the application presented here it probably isn't.

That being said, let's go ahead and do it anyway! Figure 1 illustrates this technique. I used a tool called Balsamiq Mockups to generate this diagram. This is a very nice Adobe AIR-based application that allows you to create diagrams much like Microsoft Visio or similar products, but do so with what looks like hand-drawn graphics. I like this because it gives you a bit of a middle-ground: it's low-fidelity enough to not lock you into things (which is frequently the case with prototypes that are too detailed and "real"-looking), but it looks a bit better than true hand-drawn storyboards do, especially if you have to describe an application to folks in the boardroom!

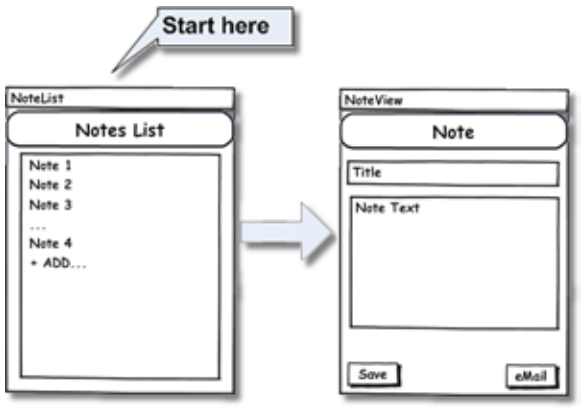

Figure 1. A mockup of the scenes (and flow through them) in this application

As I mentioned, it's a pretty simple application consisting of two scenes. The first, NoteList, will literally just present a list of notes the user has created. Here, the user will be able to add a new note, open an existing note or delete an existing note. As we'll see shortly, the List widget is used for this, which gives us a nice, concise way to provide all this functionality in a way that is standard for webOS applications.

The second scene, NoteView, is where the user views and enters notes. It consists of two TextField widgets for data entry, and a command menu at the bottom where we have icons for saving the note as well as e-mailing it. This could have been done with simple Button widgets as well, but the command menu has some advantages, as we'll see later on.

The user moves from the NoteList scene to the NoteView scene by tapping a note in the List to edit it, or by tapping the Add item at the bottom. They move from the NoteView scene back to the NoteList scene by doing the webOS-standard back swipe gesture, or as a result of saving a note, which brings them back automatically.

Erecting the skeleton

The first step to building this application is to create a skeleton, and that is done in the exact same manner as described in last month's part 1 article.

If you are going to build this application as you read through the article, please go do that now. It's a simple matter of using the Mojo Application wizard in Eclipse. Name the application "NoteTaker" and you can leave the rest of the application configuration options as they are by default, or set them to whatever you like.

The other thing you'll need to do is to add the two scenes. This too was covered in part 1, so please refer back to the last issue if you need a refresher. The two scenes you'll be adding should be named NoteList and NoteView, which will generate the corresponding NoteList-assistant.js and NoteView-assistant.js files in the app/assistants directory, as well as the files NoteList-view.html and NoteView-view.html in the app/views/NoteList and app/views/NoteView directories. There are actually two more file you'll need to add, but I'll hold off telling you about them until later when we're within the proper discussion context.

I'll assume from here on out that you've accomplished this step of setting up the basic application skeleton, or that you're just walking through the existing code that you've downloaded already.

Global Scope is bad, m'kay?

Usually in a webOS application you'll want to follow the same rules for being a good JavaScript citizen that you do everywhere else, and one of the main tenants along those lines is not to pollute global scope any more than you have to. Most of what you do in webOS happens within the confines of a given scene assistant, or perhaps a stage or application assistant, so to large extent your code will be confined to a given context fairly naturally.

However, what happens when you want to share something between scenes? There are a couple of approaches you can take. One is to pass the data explicitly from one scene to another. When you use the `pushScene()` method to show a scene, you can pass any information you like as part of that call. In part 1, we saw this method being used just by passing it the name of a scene, but that's only one way of using that method. Another way is like this:

```
this.controller.pushScene("sceneName", "abc", 123);
```

Any additional arguments after the first one, the name of the scene to push in this case, will be passed to the scene assistant's constructor, where you can do what you need to with them.

Another approach to sharing data (or common functions) is to make them members of the stage or app assistant. Those are accessible from any scene, so they are a definite possibility.

A third approach is to simply create a new arbitrary JS file, add it to your sources.json file so Mojo will load it automatically (you could also load it via a `<script>` tag in index.html), and put your common stuff there. Now, if you do it this way, this is where you have to take care to not pollute global scope. You can accomplish this goal by simply creating an object and putting all your common stuff there. That way, it's only a single object in global scope.

This third approach is the one I prefer and is the one I've used in this application, and this is where we get into one of those two remaining files that need to be added to the project. You'll need to create a file named "NoteTaker.js" in the app directory, and then add the appropriate entry to sources.json to load it. There is no wizard for doing this so it's a completely manual step, but thankful

a simple one! You have only to right-click the app directory in the Project Explorer, click "New", and then "File". Enter the filename and click Finish. Then open up sources.json, copy one of the elements in the existing JSON, modify the path as appropriate and save it. That's it!

In this new NoteTaker.js file, create an object like so: `var NoteTaker = {};`. Everything else described in this section is going to be a member of this object, and the first such member is: `db : null`. I haven't actually told you this yet, but this application will be using a local relational database to store notes in, and this field will store a reference to that database for use throughout the application.

The second field to add to this object is: `currentNote : null`.

When the user wants to edit an existing note, they will indicate this on the NoteList scene by selecting it. What we'll have to work with will be an object with the details of the note. The NoteView scene will be pushed, and it will need access to the selected note object, which this field will point to (see, there was a reason to be talking about how to share stuff between scenes after all!)

Now, I mentioned a moment ago that we'll be using a database in this application, so how about we go take a quick look at what webOS has to offer in this area?

A database in every pot!

In a webOS application, you have a number of choices for how to store data: being a web application you can use cookies; there is also something called the Depot that is a basic but useful ORM (Object-Relational Mapping) solution, and then there's the database support specified in HTML5.

The HTML5 specification includes database capabilities, and most browsers that implement this have so far used the SQLite engine (I'm in fact not aware of any that currently don't use SQLite). Here, webOS is no exception either.

SQLite is a small, relatively simple and efficient relational database engine that provides most of what you think of when you talk about relational databases. You get your usual tables, keys, indexes and such and even a few more advanced things like triggers.

Interestingly, the Depot that I mentioned here is, in fact, a Mojo API that is a layer of abstraction sitting above the HTML5 SQLite relational database. It allows you to think more in terms of objects than SQL statements. I'm a little more old-school though and I still prefer straight SQL, most of the time at least. In my book I cover both approaches, but for this application it frankly isn't much easier either way, so I just fell back on my old-school ways and went with the "bare metal" SQL approach, so to speak. That being said, a little later we'll look at what is takes to use Depot, if for no other reason than you'll be able to make a more informed decision on which to use in your own applications.

Anyway, when it comes to HTML5, a given application in webOS land can be associated with any number of databases, but each application can only access data in a database they created. One database is typically enough, but there's no limitation.

Initializing the database

Going into the code for working with the HTML5 database capabilities in webOS, we begin with initializing the database. This will be handled by a function that is a member of our `NoteTaker` object, `initDB()`, and is shown in Listing 1.

```
initDB : function(inCallback) {

  NoteTaker.db = openDatabase("jsmag_NoteTaker", "",
    "jsmag_NoteTaker", 65536);
  NoteTaker.execSQL({
    sql : "CREATE TABLE IF NOT EXISTS notes " +
      "(title TEXT PRIMARY KEY NOT NULL, note TEXT); GO;",
    params : [ ],
    callback : inCallback
  });

}
```

Listing 1: Initializing the database.

We'll see where this is called from a little later, but let's talk about it in isolation for now. The first thing we see is a call to the `openDatabase()` function, which is a global-scope function that accepts four arguments: the database name (which can be anything you like), the database version (more on this after the break!), the display name (not currently used anywhere) and the estimated size in bytes of the database. The version argument is used to support updating of your application. If you update the app, it can check to see what version of the database is already on the device, if any, and do any data porting and structural changes that are necessary (although it's more likely to just create a new database with a new version number and then copy the data over from the old version, but that's an app-specific decision). The `openDatabase()` function returns a handle to the database, which we store in afore coded `db` field of the `NoteTaker` object. Note too that if the database does not yet exist, it will be created by this function call, which makes our lives a lot easier!

Once the database is opened (and/or created), the next step is to execute a SQL statement to create the one table we're going to need, assuming it doesn't already exist. Our notes table has a very simple structure: a `title` field and a `note` field, both `text` type fields, with the title being the primary key (and so has to be unique and cannot be `null`). I'm assuming you have at least some SQL knowledge and that SQL statement there is relatively self-explanatory.

The way this SQL is going to be executed is by calling the `execSQL()` method of the `NoteTaker` object, which is what we're going to look at in just a little bit. Take note of the `params` and `callback` attributes of the object being passed to `execSQL()`. I'd be willing to bet you can guess what they're all about already, but it will become crystal clear as we check out `execSQL()` itself. Before that though, let's take a quick look at the lower level code that interacts with the HTML5 database in webOS. In fact, this is the code that `execSQL()` will wrap around.

Queries to the left of me, queries to the right!

When you want to execute a SQL statement against an HTML5 database in webOS, you'll use a bit of code similar to Listing 2.

```
db.transaction((function (inTransaction) {
  inTransaction.executeSql("xxx", [ ],
    function(inTransaction, inResultSet) {
      // Process results of SQL statement
    },
    function(inTransaction, inError) {
      // Handle error somehow
    }
  );
}));
```

Listing 2: Executing a SQL statement

Everything you do against the database is done within the confines of a transaction, which will automatically be committed (or rolled back) based on the outcome. The `transaction()` method of the opened database object (`db` here) is the way to do this. This method gets a reference to a transaction from the database, and then calls the function you pass into `transaction()`. This method gets a reference to that transaction object, on which it will then call `executeSql()` one or more times. Yes, you can execute as many SQL statements as like within the bounds of a single transaction, which is exactly the way transactions are meant to work!

The `executeSql()` method, in turn, accepts four arguments. The first is the SQL statement to execute ("`xxx`" in Listing 2). The second is an array of values to insert into the SQL statement. You see, the SQL statement can contain question marks, which will dynamically be replaced with the values from the array passed as the second argument. So, for example, the SQL statement: `select * from people where name=?`, using this value array: `[ "Mike" ]`, will result in this SQL statement being executed: `select * from people where name='Mike'`.

Note that proper quoting of values is handled for you automatically; you just feed it the values and off you go.

Of course, if your SQL statement doesn't need any data inserted, as is the case when we're initializing the database and creating the notes table if needed, you can pass an empty array and everything will work fine.

The third argument to `executeSql()` is a function to be called when the SQL statement executes successfully. This is where you'll be able to get the results of a select and do whatever you need to with it. The fourth argument is a function to be called in the case of an error.

So, now that we know the basics of how to interact with the database, let's get back to `execSQL()`, shown in Listing 3, and see what it's all about.

```
execSQL : function(inCallParams) {

    Mojo.Log.info("sql = " + inCallParams.sql + " ---- params
= " +
    inCallParams.params);

  try {

    NoteTaker.db.transaction((function (inTransaction) {
      inTransaction.executeSql(inCallParams.sql,
inCallParams.params,
        function(inTransaction, inResultSet) {
          var results = [ ];
          if (inResultSet.rows) {
            for (var i = 0; i < inResultSet.rows.length; i++)
{
                results.push(inResultSet.rows.item(i));
            }
          }
          inCallParams.callback(results, inCallParams);
        },
        function(inTransaction, inError) {
          Mojo.Controller.errorDialog(
            "execSQL error: " + inError.code + " - " +
inError.message
          );
        }
      );
    }));

  } catch (e) {
    Mojo.Controller.errorDialog("execSQL exception: " + e);
  }
}
```

Listing 3: the execSQL() function

The whole point to this method is to provide a thin abstraction layer above the basic interaction code we've just looked at. We want to ensure that we can have some common things always present. For example, `execSQL()` provides a common way to handle errors (as well as exceptions, which is a different situation). It also allows us to deal with the HTML5-specific `ResultSet` that will be returned and instead provide the caller a slightly simpler data structure (so that if we wanted to store the data some other way later, so long as that approach returned an array of simple objects too, our application wouldn't have to change, just this method's implementation). It also allows us to have some common logging for every SQL execution to make debugging easier.

Note: another file you can optionally add to your application, in the root directory, is "framework-config.json". As the name implies, this allows you to configure some Mojo framework settings, one of which

is logging. If you create this file, and add the code `{ "logLevel" : 99 }` *to it, this will enable logging. Then, if you're running the application in the emulator, you can use your favorite SSH client, such as PuTTY, to connect to the emulator (localhost, port 5522, username root, no password). Then, if you go into /var/home/root and type* `log xxx.yyy.xxx` *where xxx.yyy.zzz is the ID from the* `appinfo.json` *file, you'll be able to see the output of any* `Mojo.Log.xxx()` *method invocations, where xxx is a log level like* `info` *or* `error`*.*

You should see the same general structure to the code that we just talked about, but here we have an implementation for the success callback. Two things get passed to this function: the transaction and a `ResultSet`. I only want my application code to deal with simple arrays though, so here I iterate over the results, where each row of data returned by the query is an element of the `inResultSet.rows` array. A call to its `item()` method retrieves the next row of data, which is a plain old object with attributes matching the columns in my select SQL, that is added to the results array, and then returned.

If any errors occur, the second function passed to `executeSql()` gets executed, and this just pops up an alert message to the user (not very robust error handling, I admit!). `Mojo.Controller.errorDialog()` is one of a handful of popup dialogs that Mojo makes available to us, this one being about the simplest. It just accepts an error message to display and nothing more. The result of this is shown in Figure 2.

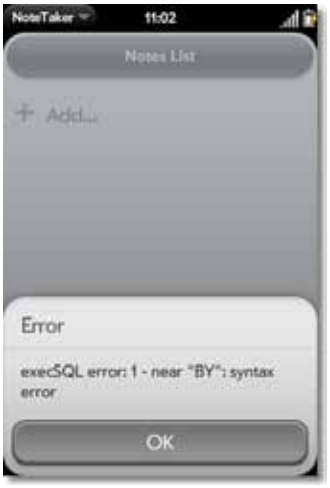

Figure 2. The error dialog.

All of this code is wrapped in a `try...catch` to account for any unexpected problems that may occur, such as calling `transaction()` on the `db` reference. In the case of an error or exception the code tries to give the user some information about what went wrong (not that there's much they can do about it, but it's better than a mystery).

A quick detour into Depot-land

Although it's not used in this application, I think it's a good idea to see the other main storage facility available to you in webOS development: the Depot API.

The Depot API is found in the `Mojo.Depot` package in the Mojo framework. Depot is a relatively simple wrapper around the HTML5 active records database facility that allows you to think in terms of objects instead of typical database structures such as tables, rows and records. Using `Mojo.Depot` is pretty simple, as the example in Listing 4 demonstrates.

```
var get2 = function() {
 depot.simpleGet("myObject",
  function(inObject) {
   $("divOutput").innerHTML += "Should be empty because " +
    "object was removed: " + Object.toJSON(inObject) +
"<br>";
  },
  function() { $("divOutput").innerHTML += "Get2 failed<br>";
}
 );
};

var remove = function() {
 depot.removeSingle("defaultbucket", "myObject",
  function() {
   $("divOutput").innerHTML += "Object removed<br>";
   get2();
  },
  function() { $("divOutput").innerHTML += "Remove
failed<br>"; }
 );
};

var get1 = function() {
 depot.simpleGet("myObject",
  function(inObject) {
   $("divOutput").innerHTML += Object.toJSON(inObject) +
"<br>";
   remove();
  },
  function() { $("divOutput").innerHTML += "Get failed<br>";
}
 );
};

var add = function() {
 depot.simpleAdd("myObject",
  { firstName : "Burt", lastName : "Reynolds" },
  function() {
   $("divOutput").innerHTML += "Add success<br>";
   get1();
  },
  function() { $("divOutput").innerHTML += "Add failed<br>";
}
 );
};

var depot = new Mojo.Depot({ name : "myDepot" },
 function() {
  $("divOutput").innerHTML += "Create success<br>";
  add();
 },
 function() { $("divOutput").innerHTML += "Create
failed<br>"; }
);
```

Listing 4: a Depot example

This perhaps looks a little more complex than it actually is, but that's partly a result of the asynchronous nature of the calls made to `Mojo.Depot`. The complexity can also be attributed to the fact that each of the functions you see is one link in a chain formed by the execution of the callbacks attached to the asynchronous calls.

Each call to Depot has a success and failure callback, which are functions that will be called in those respective situations. Remember, the flow of the code in the larger application as a whole won't stop when you make these calls (the very definition of an asynchronous call!). Because of this, we can't simply do a `Mojo.Depot.simpleAdd()` call followed by a `Mojo.Depot.simpleGet()` call because the second call very well might execute before the first completes, and that would be a Very Bad Thing™ indeed! So, in light of this, we create a couple of functions, assign them to variables, and then use those variables within each of the callbacks to execute them at the appropriate times (that's the chain I was referring to earlier).

It all really starts with the instantiation of a `Mojo.Depot` object near the bottom. The argument passed to the constructor defines the depot, and the only required attribute is `name`, although others can be passed as well:

- `version`: This is the version number used for the HTML5 database (defaults to 1).
- `displayName`: This is the name that would be shown in a UI to the user (this is not currently used).
- `estimatedSize`: This is the size you think the database will wind up being. Although the documentation doesn't say, my supposition is that this will allow the database to be pre-allocated and make it a bit more efficient.
- `replace`: If true, any existing data will be erased, and the depot will be re-created.
- `filters`: This is an array of strings that objects in the depot can use as filters.

The second argument passed in the call to `Mojo.Depot()` is the function to call when the creation is successful, and in that case we output a message to `divOutput` in the scene HTML indicating the creation was success. Following the notification of the success, the `add()` method is called. If the create fails, the function passed as the third argument is executed, and a message indicating the failure is output (this is the case for all of the depot method calls, so I won't mention the failure callback again).

The `add()` method then uses the `Mojo.Depot.simpleAdd()` method to add an object under the key `myObject`. As you can see, we're dealing with a JSON-defined object, not SQL or database structures, which is the whole point of `Mojo.Depot`. In the success callback, after a message is output, the `get1()` method is called.

The `get1()` method calls the `Mojo.Depot.simpleGet()`, passing it the key to retrieve. The success callback is passed the retrieved object, which we then output to our `divOutput` using the `Object.toJSON()` method that has been added to the `Object` prototype by the Prototype library.

After that, the `remove()` method is called, which calls the `Mojo.Depot.removeSingle()` to remove the object from the depot. This method accepts the key of the object to remove, `myObject` here, but it first takes something called a bucket. Buckets are places you can stash objects within the depot. Unfortunately, it doesn't seem possible to add objects to any bucket other than the default bucket, which means that's where we need to remove the object from. The value `defaultbucket` is the value of a hidden field in `Mojo.Depot` that names the default bucket. Unfortunately, since this isn't exposed as a public field, we have no choice but to hard-code the value and hope it doesn't change!

Note: I think the `Mojo.Depot` *package is a perfect example of the sorts of rough edges that development for webOS has at this point. The Palm documentation for this package has some flat-out incorrect information, and I was only able to make this code work by examining the actual Mojo source code. I expect this situation will improve as things proceed, but for now this is the sort of problem you'll sometimes encounter when programming for webOS.*

The callback for the `Mojo.Depot.removeSingle()` method makes a call to `get2()`, which tries to retrieve the same object. The message output to the `<div>` shows null, indicating that the object really was removed from the depot.

There is also a `Mojo.Depot.removeAll()` method that, as its name implies, removes all objects from the opened depot. Also available is a `Mojo.Depot.addMultiple()` method and a `Mojo.Depot.getMultiple()` method for adding and getting multiple items at one time. See the documentation for examples of their usage (hint: it's not a whole lot different from the example code shown here!).

One more detour, because I'm hungry: cookies

As mentioned earlier, you also have the capability to use cookies for data storage as well. Cookies are a ubiquitous and well-known data storage mechanism in the world of web applications, and webOS provides support for them, just as one would expect from a web technology-based operating system! The `Mojo.Model.Cookie` package is actually a class that you'll instantiate to represent a cookie and it's the gateway into all your junk-food needs!

For example, say we want to store the name of our user in a cookie. We might name the cookie `userName`. We first have to create the cookie, and then write a value to it. The code for that can be seen in Listing 5.

```
var userNameCookie = new Mojo.Model.Cookie("userName");
userNameCookie.put("tyra_banks");
var cookieValue = userNameCookie.get();
userNameCookie.delete();
```

Listing 5: a cookie example

Shown in the Listing 5 code is also how you retrieve the value of the cookie, as well as how you delete it. The interesting thing to note here is that the call to the `get()` method can return `null` if the cookie doesn't yet exist. You see, the instantiation of the cookie really just gives you a façade to work with; the actual cookie isn't created and stored until a call to `put()` occurs. Therefore, your code will need to deal with the situation when `get()` returns `null`. This might mean throwing an exception, or simply writing out a default value to the cookie, whatever makes sense within the flow of your application.

Likewise, after the call to `delete()` completes, the cookie isn't actually deleted, its value is simply cleared, so you need to account for that situation as well.

I frankly found this API design a little more confusing than it needs to be. It feels to me like there's an unnecessary level of abstraction in there. But, personal feelings aside, it's not at all complicated and basically does what you'd expect.

Like Depot, our note taking application won't make use of cookies, but at least now you know about the three ways you can store data locally in a webOS application. As of this writing, there is no direct file access available to a webOS application, so these truly are the only ways to store data locally at the moment.

Next Time

Next month, in part 3 of this webOS series, we'll finish up our note taking application and look more closely at scenes and widgets. We'll build some menu functionality and look more in depth at some of the services and dialog offerings that are available from the Mojo framework. Until next time ...

Frank W. Zammetti has been developing software for over a quarter century in a wide variety of technologies for myriad platforms. Frank has been a professional developer/architect for one of the five largest financial institutions in the U.S. for the past thirteen years. Frank holds a large number of professional certifications spanning a variety of technological areas and is a contributor to, founder and leader of a number of well-known open-source projects. Frank has authored a number of programming-related books and articles and regularly presents at various conferences and user group meetings. Frank founded a small software development company, Etherient, focused on mobile and cloud-based computing products for various platforms.

JsDoc *Continued from page 10*

tags and command line options you may find you can document even complex relationships in your code quite easily. Getting help as you learn is easy too, the JsDoc Toolkit wiki and user's group is a convenient place to find out more.

Resources cited in this article

- http://code.google.com/p/jsdoc-toolkit/
- http://jsdoc.sourceforge.net/
- http://groups.google.com/group/jsdoc-2
- http://java.sun.com/j2se/javadoc/writingdoccomments/index.html
- http://code.google.com/p/jgrousedoc/
- http://docs.aptana.com/docs/index.php/ScriptDoc_comprehensive_tag_reference
- http://developer.yahoo.com/yui/yuidoc/
- http://code.google.com/closure/compiler/docs/js-for-compiler.html

---

Michael, having previously worked as a zookeeper and then a science museum teacher for the New York City Public Schools, has been building professional web sites for various major media companies for more than ten years now. To add to his confusion he still considers himself a New Yorker, even though he now works for the BBC in London, UK. You can follow his rants on zoology, object-based teaching, expatriation, or even, occasionally, web development at @micmath on twitter.