PouchDB iBook

The Cloud Widget

Imagine an ibook Widget that supports updates, push notifications, real-time data syncing, and chat/discussion support. Well, thanks to CouchDB, and the development of other noSQL databases, the super cloud widget I dreamed about before starting this project could be a reality. Which is why this project came to light: to develop, test, rinse and repeat the possibility of having an iBook widget that could push and receive remote updates to and from a centralized database, and even remotely sync session data from the widget back to the database in real-time.

To accomplish such a feat, I would need my usual go-to programming toolkit: (1) My trusty Macbook Pro, (2) My iPad Mini with iBooks installed (3) Sublime Text 2 and it's endless access to packages and coding goodies, (4) Google Chrome and Safari with their amazing webkit devtools, and (5) iBooks Author.

The Star I Shot For

In attempts to provide myself with a feasible and attainable goal, a stable and simple project idea was born. PouchDB is a drop-in javascript driven API for mobile devices to interact with IndexedDB or WebSQL and essentially store information client side and interface with an external CouchDB server for syncing. PouchDB natively supports CouchDB's replication and syncing capabilities, so these capabilities quickly became evolved into the framework I would need to build my cloud widget.

It turns out, conveniently, that a live chat environment built on PouchDB would provide the concept and testing to prove almost anything possible for the framework. The idea is simple: Open an iBook widget, enter your name, and start chatting with other individuals who are also reading the iBook. This straightforward functionality would prove the concept of pushing, receiving, and syncing information from a remote noSQL database inside of a widget.

Beginning Development

When building HTML widgets for an iBook, I find that there is no better development environment than Sublime Text coupled with the browser. Chrome and Safari in this instance, would later give me the ability to easily debug and inspect my tests and various build cycles.

After referencing Apples iBook widget creation documentation, I quickly started a new project and started with testing CouchDB's awesome and unique features through PouchDB. Using very simple API callouts, it was easy to create a local database and start saving JSON documents for later use. I remembered the feeling I got the first time I used a noSQL stack, and the reason why I hate traditional SQL databases!

Hurdles and Limitations

Having such easy access to a JSON database, you can quickly get carried away with the size of the database and the size of your JSON documents. One example of this is storing a couple of high quality photos as base 64 data inside of the database (which is exactly what I did to test the size restraints of each browsers indexedDB/WebSQL maximum size. It turns out that each browser does have it's own different limitations and even changes periodically depending on the build and version of the browser. Currently the latest version of the most popular browsers supper the following:

Browser	Database Type	Storage Limits
Chrome	IndexedDB	Available HD / 2 / 5
Safari	WebSQL	50MB
Firefox	IndexedDB	∞
iOS Safari	WebSQL	5MB + 45MB

Finding the limit on iOS Safari was immediately scary, as it is easy to exceed 5MB in any frequently used database right off the bat. But after research, I found it possible to expand per requesting the user for more storage space (up to 50MB). From then on, I inferred from previous knowledge that the Safari driven UIWebView supplied within the iBook would be no different in it's size limitations. So, I prepared for the worst as usual!

Controller Time!

The app structure I had come up with was essentially a modified version of the PouchDB's todo list syncing app. I had already stripped out the todo list functionality and built in a chat-room like interface, just something simple and easy to get the idea across.

Behind this simple interface is also a simple, controller. The username is stored in a local Javascript variable that is reset on each page load, and the rest of the app is bolted on to a couple of PouchDB API's.

On loading, the database is initiated locally with a blank slate, or (assuming replication could be turned on, the contents that were previously synced or entered). In a typical case, the database would contain one document for each entry in the chat room, which each contains a unique ID, revision ID, and message. The message contains a a string with the username and text content of the message.

As a user submits messages to the chat room, the database size is constantly measured for changes in the database from any source. In theory, when a change is detected, the new contents of the database will be replicated to the client, showing the message from another user. When a user submits his own message in response, the local database gains the entry, and replicates the new database back to the server, resulting in the same continuous function.

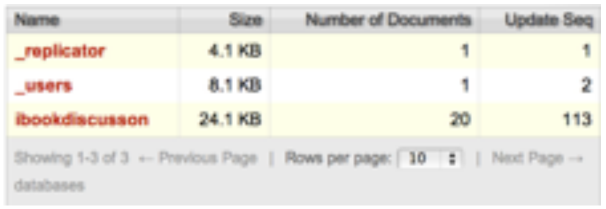
Do you remember our database limitation? 5MB was our goal, and this could be scary if one is looking at a chat room that could quickly grow over that size with a large number of users or simply over a large time span. To save myself local storage size (which could cause the iBook to inflate) and also the hassle of paying for extra storage space in the cloud, I strapped a function to the syncing action which constantly monitors the number of documents (or entries) in our chat database. If the number ever exceeds the size defined by the developer (in my own case, 10 documents), the database will be trimmed and then replicated. The same function includes a setting to limit the amount of entries displayed to the user, but still hold on to the remaining messages as a log. By turning off replication and syncing temporarily, the app was ready to test in the browser.

Browser Success!

It's amazing what can be done so quickly with these awesome frameworks. But, despite my project working in the browser, I needed to remember my original goal to get it working inside of an ibook **and** syncing to the cloud. As I expected, implementing the first stage webSQL database in the widget structure would be clear and feasible. So, feeling fairly straightforward, I dropped my app into the widget and and let 'er fly! Fortunately, the app worked the first time! But I was still chatting only with myself. Time to get to the cloud!

Iriscouch & “The Cloud”

Our next task was getting to the cloud, which if you know anything about CouchDB is supposed to be extremely easy. Replication to and from a database is built in and supports a continuous flow between the two or three or how ever many slave databases you want to incorporate into the mix. There is obviously some wariness with scalability that comes with this approach, but it would work just fine for my purposes. After dealing with what seemed like hours of painstaking SSH sessions and server configuration to install couchDB on my shared hosting server (I wouldn't recommend it anyway), I happily resorted to using IrisCouch.com's cloud hosted CouchDB environment. They offer free accounts, and a comfortable limit of resources to use for development and testing before you have to start paying for the service. Signup was quick, and integration with PouchDB was strikingly fast and easy.



Name	Size	Number of Documents	Update Seq
_replicator	4.1 KB	1	1
_users	8.1 KB	1	2
ibookdiscusson	24.1 KB	20	113

Showing 1-3 of 3 ← Previous Page | Rows per page: 10 | Next Page →

databases

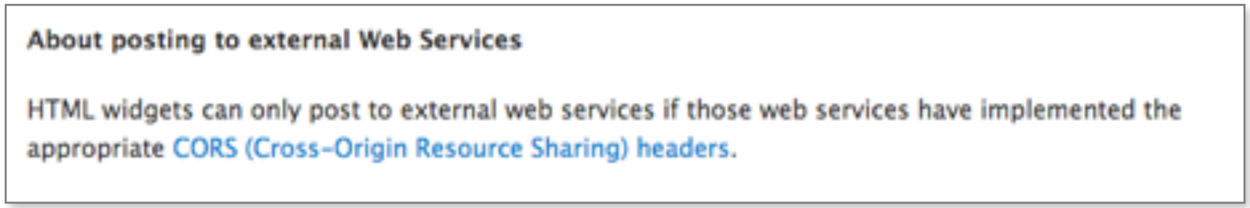
DB structure on Iris Couch's Futon interface

Before I could even read through all of the documentation, I started implementing the code. After getting familiar with Futon and mowing through some existing example projects from PouchDB's website, I did notice some performance issues with IrisCouch.com. Mostly related to latency or what appeared to be my database entering a sleep status until woken up. After talking to support, the issues was cleared up and my CouchDB was ready to roll.

Thinking it would be a breeze, I skipped the browser, and dropped in my widget with the few changes I had made.

Dashcode Debugging

Enter Dashcode, Apples widget building suite that just happened to supply me with the debugging information I needed to find the problem that immediately arose. After seeing my widget freeze after opening in my iBook, I opened up the widget in Dashcode, clicked run, and watched the console spit problems. Both jQuery and PouchDB got plenty of red flags running through my widget. For now, the only one I worried about was PouchDB, because previously I had learned that jQuery is widely known to throw errors in Dashcode's environment but still function correctly in the live widget. Displaying as **SecurityError: DOM Exception 18**, the error caused pouchDB to completely crash

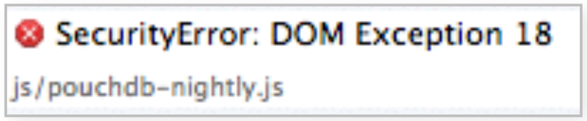


About posting to external Web Services

HTML widgets can only post to external web services if those web services have implemented the appropriate CORS (Cross-Origin Resource Sharing) headers.

Apple's iBook Widget Documentation on CORS

as soon as the database attempts to initialize. It can be a fairly ambiguous error when looking for details as I soon found out, but the Security 18 error is known to many browsers to be a cross-origin request being denied because the domains of the requests do not match. The inability to allow these XHR's (xmlHTTPRequests) function within the ibook widget would essentially break the code and also the concept.



SecurityError: DOM Exception 18

js/pouchdb-nightly.js

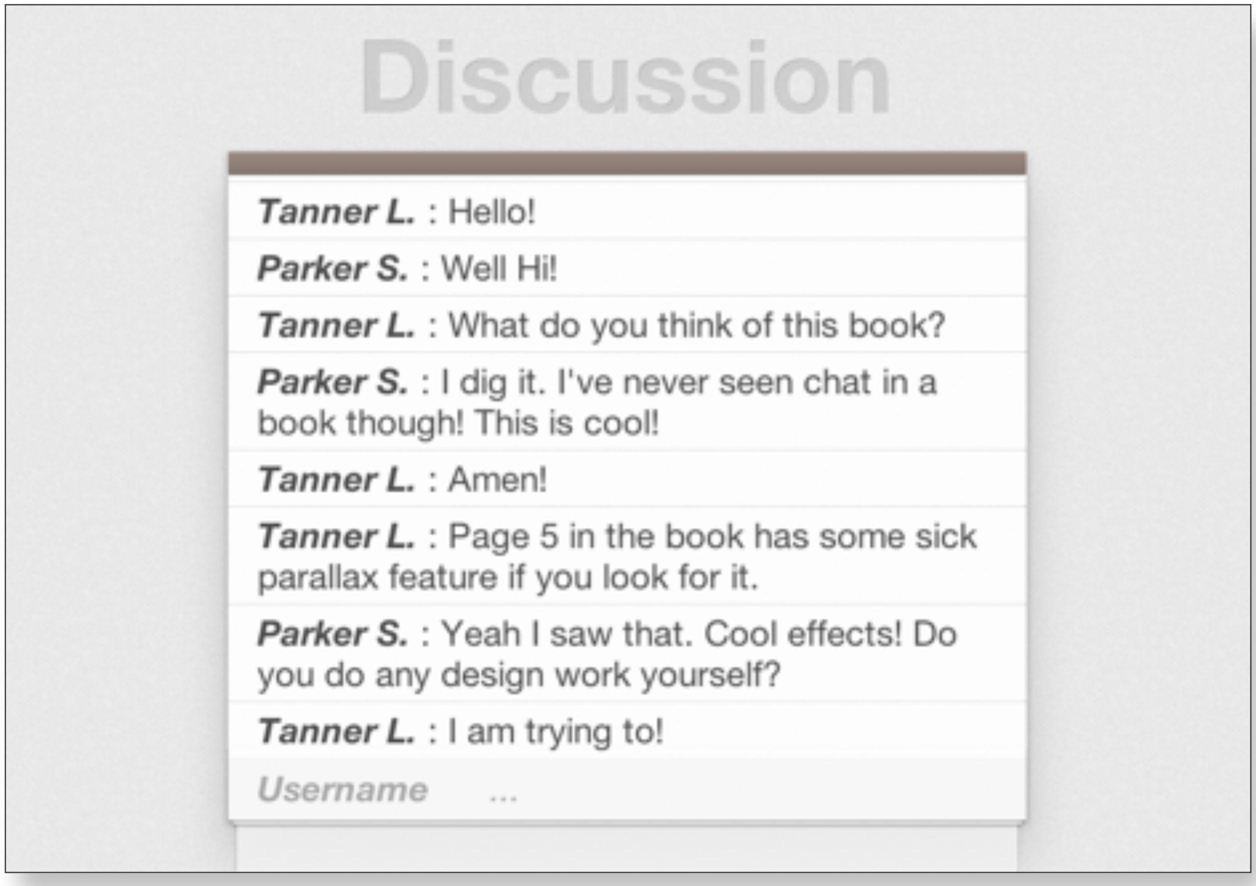
I did remember though, seeing Apple's iBook Widget documentation making careful notes that explain cross domain restrictions and XHR (shown below).

CORS Will Set You Free

According to the policy at enable-cors.org, cross-origin requests must have matching domain headers to authorize any request. For example, when working with AngularJS, (a very popular MVC framework for javascript) inside of an iBook, a CORS error is thrown when attempting to use XHR's because there is no domain to reference in a local file system (in which case, both domain headers return are null). I an into this in a previous iBook project, but later discovered that the good people of AngularJS prepared for this specific scenario and offered a way to bypass XHR restrictions by embedding scripts in the main HTML file for use with an associated ID. Until similar bypasses and tweaking happen on PouchDB's part, I can't expect the same luxury in it's framework.

The Solution

So our initial defeat came from a dubious cross domain error. Of course, as stated in Apple's iBook widget design documentation, CORS is a required implementation for external requests to function properly. But then, why would CORS be an issue if the database was only local? Technically, if using PouchDD merely as a storage API, then there shouldn't be any external calls at all. Come to find out, that upon testing the widget **without** server syncing support and replication, PouchDB functioned perfectly on our iPad iBook, but did not function at all on Mavericks iBook preview. Then after reading PouchDB's documentation in it's entirety I found the solution. They clearly state that in order to use replication in your PouchDB server, you must enable CORS support in your CouchDB server via curl, no matter where it is hosted. They even provided a quick way to accomplish this via command line as shown below.



Discussion

Tanner L. : Hello!

Parker S. : Well Hi!

Tanner L. : What do you think of this book?

Parker S. : I dig it. I've never seen chat in a book though! This is cool!

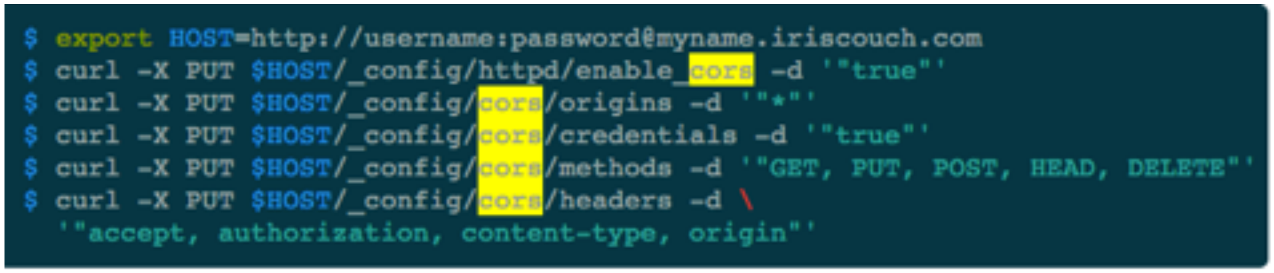
Tanner L. : Amen!

Tanner L. : Page 5 in the book has some sick parallax feature if you look for it.

Parker S. : Yeah I saw that. Cool effects! Do you do any design work yourself?

Tanner L. : I am trying to!

The completed PouchDB iBook Chat widget



```
$ export HOST=http://username:password@myname.iriscouch.com
$ curl -X PUT $HOST/_config/httpd/enable_cors -d '"true"'
$ curl -X PUT $HOST/_config/cors/origins -d '"*"'
$ curl -X PUT $HOST/_config/cors/credentials -d '"true"'
$ curl -X PUT $HOST/_config/cors/methods -d '"GET, PUT, POST, HEAD, DELETE"'
$ curl -X PUT $HOST/_config/cors/headers -d \
  '"accept, authorization, content-type, origin"'
```

Enabling CORS support on Iris Couch's CouchDB.

After enabling CORS and without even restarting the IrisCouch server, my databases immediately began syncing in realtime across the browser and more importantly, the iBook widget.

Conclusion

My findings in this project were quite revealing: To the obvious developer, the most straightforward way to access cross-origin, or cross-file information through an iBook is to have the exact file referenced via source through the main HTML document. The most common working examples of this use iframes and local scripts that are self contained with no external or un-referenced dependencies.

But more interestingly, one may use cross-origin javascript requests by implementing CORS on their own web server or database server. With frameworks such as PouchDB, and noSQL databases, this is definitely the “cooler” way of accessing and writing data to external services with offline changes and cloud syncing.
I also think it was highly significant when, at first failing, I assumed that Apple had indeed implemented strict security settings in it’s iBooks’ widget environment for good reason, to potentially stop content from being changed after submission and approval from apple’s iBook store. But, on closer investigation, they comply completely with the CORS policy that every other browser and service does that ensures secure post and get methods for cross-origin requests.

Finding this out was a discovery on my own, and it is sad to find that right now many forums on the internet (some of them right on Apple’s support) think that external XHR is restricted by Apple in a widget, but I can tell you that’s not true.

I think Apple has done the right thing here. They, of course, would not want to encourage developers to violate iBook Store policies, but at the same time have left the capability there to do amazing things inside of widgets.

I personally would love to see some of dashcode’s widget debugging features come to iBooks Author. Or better yet, allow widgets to be debugged through Safari’s remote webkit debugging interface. A good use for this would be finding out why this awesome PouchDB widget does not function in the Mavericks iBook Previewer.