

Web Data Management Assignment 2 Writeup

Alex Simes (4415299) and Peter van Buul (1512269)

Introduction

For this assignment, we built a web application around a bibliographical dataset stored in CouchDB. This was our first experience working with CouchDB and we learned a lot. To complete the application, we built a straightforward web application using PHP to connect the front end to CouchDB through curl requests. We also wrote an extensive shell script which fully configures our CouchDB setup and populates it with the documents (both data and views).

Building the Views

Before starting on application development, we built the views we would need. Since the user would query by any combination of Title, Author, Publisher, or Year, we would need to use multi-key views. Our first plan was to use a single view which contained all 4 fields as keys and design our query in a clever way to essentially pass wildcard values to the fields left blank by the user. We discovered that this sort of functionality is in fact not supported by CouchDB, which left us with the option of creating 15 different views, one for each possible combination of search criteria the user might want to use.

This task ended up not taking as long as we anticipated, as there was a lot of repeated code. One issue we ran into was how to quickly reset or transport our database with the views and how to leverage Git so we could pass our view code back and forth. Our solution was to store all our view code in a [json document](#) which we included in a [reset script](#) for our database. We ran into issues with having tabs and newlines inside the fields of our json file, so we ended up having to squish the files into single lines before passing them to CouchDB.

```
tr -d '\n\t' < config_files/all.json > temp.json

curl -X PUT http://admin:admin@127.0.0.1:5984/books/_design/app --data-binary
@temp.json

rm temp.json
```

Having this many views made CouchDB take a few minutes to sort itself out after a full reset, since it had to repopulate all the views. We'll discuss the full reset script and how we queried the views later on.

Meeting Application Requirements

















Searching the Bibliographic Collection

The first thing we tackled was searching through the bibliographic database. We needed to allow searching by any combination of four fields. We provide the user with the search form shown below, in which he can fill in however many fields he likes.

Title:	Journal or Publisher:
<input type="text" value="algorithms"/>	<input type="text"/>
Author:	Year:
<input type="text"/>	<input type="text" value="2008"/>
<input type="button" value="Submit"/>	

Upon hitting submit, the form contents are sent off to a [PHP script](#). This script starts by detecting which fields the user filled in. From this knowledge, it knows which of our pre-made views to query and with what keys. This query is formatted and sent off in our PHP code.

Upon being returned, we needed to do a small massaging step in our data. The way our views are structured, we sometimes had duplicate entries returned. We converted the return JSON to a PHP associative array to squash all duplicates into single rows. After this step, we print the result in an HTML table as shown below.

Title	Author(s)	Year	Source	View PDF	Edit	Delete	Add PDF
Robotic Exploration and Landmark Determination: Hardware-Efficient Algorithms and FPGA Implementations.	K. Sridharan, Panakala Rajesh Kumar	2008	DBLP				
Multi-Objective Evolutionary Algorithms for Knowledge Discovery from Databases		2008	DBLP				
Despeckle Filtering Algorithms and Software for Ultrasound Imaging	Christos P. Loizou, Constantinos S. Pattichis	2008	DBLP				
Bio-inspired Algorithms for the Vehicle Routing Problem		2009	DBLP				

Create/Update/Delete Bibliographical Entries

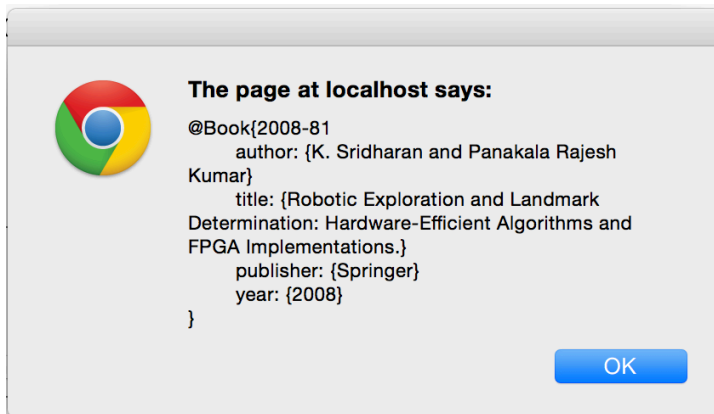
Next, we needed to create a way to add/update/remove entries. You'll notice in the screenshot above that we have links to edit and delete the entries in the search results. The editing link brings the user to a [pre-populated form](#) where they can edit values and submit the changes to CouchDB. The delete link simply passes the document id to a [PHP script](#) which deletes the document using, again, a curl request.

To add new entries, we used an [HTML form](#). This form submits to a [PHP script](#) to process the new data. We used curl requests to query CouchDB at `http://127.0.0.1:5984/_uuids` for a unique ID rather than generate our own.

Producing Bibtex Entries

You'll also notice that the titles in our search results view are clickable. The title can be clicked to provide the user with a Bibtex entry of the document.

This Bibtex entry is generated with a short [javascript function](#) and shown to the user in an alert box, as shown below.



Uploading PDFs as Attachments

Creating an interface for the user to upload PDFs initially gave us some problems. We wanted a way to avoid the server permission issues we struggled with in the first assignment and to simply pass the attachment directly to CouchDB. We came up with an interesting way to do this. We created a [HTML page](#) which we loaded into CouchDB as an attachment. This PHP page is a file upload form and looks for GET params for an ID and revision number. Once the user selects a file and submits, the form POSTS to the correct document in our books database with the form data. This solution is simple and clean, especially considering we add the upload form page to our CouchDB setup in the reset script.

An interesting issue we ran into with coding the form document was the names of the GET parameters we passed to it. CouchDB seemed to get confused if we named them `id` and `rev`, so we went with `doc_id` and `doc_rev` instead. Also, since the CouchDB server can't actually parse PHP, we had to use a javascript trick to get the GET parameters.

Replication

Now we can finally discuss the [reset script](#) we've mentioned so many times above. Using this script, both of us could quickly have our CouchDB databases in the exact same state.

The script starts by dropping and creating our three databases: books, books2, and books_app. The books_app database holds our PDF upload form, discussed above. The books and books2 databases are set up in a master-master configuration. This means they always replicate to each other.

From this point on, we only need add to books and it automatically replicates onto books2. Next, we add all of our views. This is done by removing the tabs and newlines CouchDB was complaining about and doing a PUT to

`http://admin:admin@127.0.0.1:5984/books/_design/app` with the squished file.

Next, we add the PDF upload form to the books_app database. This takes two steps. First we create the handle_upload document, then add the attachment PHP page to it. The reason this was a bit tricky was that when adding an attachment to a document, CouchDB requires the revision number. With some hacky shell scripting we were able to pull the revision number of the handle_upload doc from the curl response to the insertion.

And finally, we unzip the books-json package, insert all books, and clean up the json files afterwards. This complete process leaves us a CouchDB setup that is completely ready for our application!

Using the Log to see Modifications

We ran into a lot of problems trying to implement this last part of the assignment. While we were able to see the logs on our local filesystem, we did not find a way to query CouchDB for the logs and get them back in a format to present to the user. The one thing we found was doing a GET request on `http://127.0.0.1:5984/_log` which dumps out the last few items in the log. However, this was not enough to build the functionality requested.

After some further investigation, we found the 'http://127.0.0.1:5984/books/_changes' that can be used to query the changes made to the books database. By adding the `descending=true` parameter and selecting the top 10 we had an initial implementation for browsing the changes that were made recently. Lastly, we selected the last 10 changes and show those to the user.

The next part of the changelog was to enable users to specify from which documents they want to see the changes. For this, we first had to add a filter to the database, this filter is added along with the views in the reset script. The code for the filter can be seen [here](#) or below:

```
function(doc, req) {
  var values=JSON.parse(req.query.ids);
  if(values.indexOf(doc._id) > -1) {
    return true;
  }else {
    return false;
  }
}
```

As you can see, the filter we created works by comparing ids. We took the user search parameters and reused our 15 views to get a list of ids that match. This way we could show changes in only those matching ids. Our final GUI for the changelog is shown below.

Filter Changelog

Title:	Journal or Publisher:
<input type="text" value="algorithms"/>	<input type="text"/>
Author:	Year:
<input type="text"/>	<input type="text" value="2006"/>
<input type="button" value="Submit"/>	

Changelog

Title	Author(s)	Year	Source	View PDF	Edit	Delete	Add PDF
Kernel Based Algorithms for Mining Huge Data Sets: Supervised, Semi-supervised, and Unsupervised Learning.	Te Ming Huang, Vojislav Kecman, Ivica Kopriva	2006	2006-17				
Resource Allocation in Wireless Networks: Theory and Algorithms.	Slawomir Stanczak, Marcin Wiczanowski, Holger Boche	2006	stanczakwb06				