Katie Griffiths

T00175748

CouchDB Database

Advanced Database Programming

Table of Contents

[Introduction 3](#_Toc496218823)

[Question 1 3](#_Toc496218824)

[JSON Document Example 4](#_Toc496218825)

[CRUD with REST Interface 5](#_Toc496218827)

[Create 5](#_Toc496218828)

[Read 5](#_Toc496218829)

[Update 6](#_Toc496218830)

[Delete 7](#_Toc496218831)

[Question 2 8](#_Toc496218832)

[Evaluation of API 8](#_Toc496218833)

[CRUD through the API 9](#_Toc496218834)

[Connecting to CouchDB through nano 9](#_Toc496218835)

[Create 9](#_Toc496218836)

[Read 9](#_Toc496218837)

[Update 10](#_Toc496218838)

[Delete 11](#_Toc496218839)

[GUI Interface 12](#_Toc496218840)

[Creation with the GUI 13](#_Toc496218841)

[Adding an Item to your shopping list with the GUI 14](#_Toc496218842)

[Updating with the GUI 14](#_Toc496218843)

[Delete with the GUI 15](#_Toc496218844)

[MapReduce with the GUI 16](#_Toc496218845)

[Advantages and Disadvantages of the API 16](#_Toc496218846)

[Conclusions 16](#_Toc496218847)

# Introduction

Apache CouchDB is open source database software that focuses on ease of use and having a scalable architecture. It has a document-oriented NoSQL database architecture and is implemented in the concurrency-oriented language Erlang. It uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API.

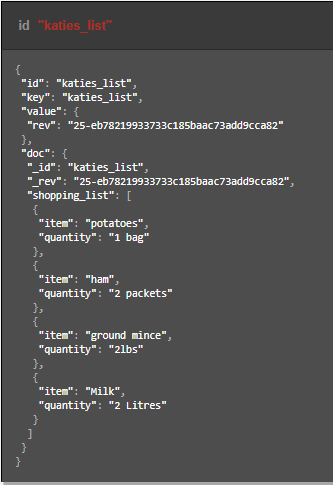
Unlike a relational database, a CouchDB database does not store data and relationships in tables. Instead, each database is a collection of independent documents. Each document maintains its own data and self-contained schema. An application may access multiple databases, such as one stored on a user's mobile phone and another on a server. All calls to CouchDB happen over its REST interface and replication can be one-way or bidirectional and ad hoc or continuous, it gives you a lot of flexibility to decide how to structure, protect and distribute your data.

# Question 1

For this project I chose to create a web application to manage shopping lists across a household. As the application would need to be accessible from both desktop and mobile, I felt that implementing it as a web app would be the best approach. CouchDB is well suited for a project like this, as each shopping list can be implemented as a distinct document containing both the actual list (As an array of items with quantities) and various meta information (Such as a list name). I felt that JavaScript was the ideal choice for this project due to the fact that JSON is native to it, making it trivial, in many cases to work with the CouchDB documents in a variety of ways without having to worry about issues that can occur in other languages when switching back and forth between JSON and their native language implementations of arrays etc. While it would have been possible to create this project as an entirely client side application, directly performing ajax requests to the CouchDB server through either native JavaScript, or a front end library such as jQuery or React, I found that it was quicker to get up and running using node with express as my web framework. This allowed me to easily implement a web server that could handle requests, templating and other useful helpers for creating a website.

# JSON Document Example

## 



# CRUD with REST Interface

## Create

Using the POST command lets you create a new document. For this I created a document representing a shopping list. The values given back in response to the command contains a JSON object with information of the id and rev. These are useful for when you need to retrieve a document that has been created by using the id value.

Command:

curl -i -X POST "http://localhost:5984/shoppinglist/" ^

-H "Content-Type: application/json" ^

-d "{ \"name\": \"bought\_items\" }"

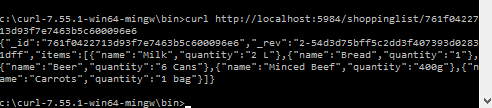


## Read

Using the GET method allows you to retrieve information about an existing document. Using the specific, unique id of the document in the cURL command returns the information contained in the document. These GET requests are always safe as they are read-only, so CouchDB does not make any changes to the documents.

Command:

curl http://localhost:5984/shoppinglist/761f0422713d93f7e7463b5c600096e6



## Update

The PUT method is used to update an existing document. In the command both the id and rev fields must exactly match the documents id and rev value for the document to update or this will fail, this is to ensure that the wrong document is not accidentally changed. The command for PUT consists of the database URL with the documents id value.

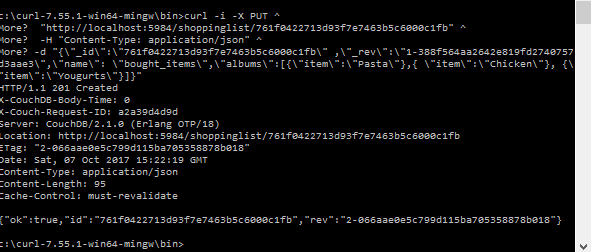
Command:

curl -i -X PUT ^

"http://localhost:5984/shoppinglist/761f0422713d93f7e7463b5c6000c1fb" ^

-H "Content-Type: application/json" ^

-d "{\"\_id\":\"761f0422713d93f7e7463b5c6000c1fb\" ,\"\_rev\":\"1-388f564aa2642e819fd2740757d3aae3\",\"name\": \"bought\_items\",\"albums\":[{\"item\":\"Pasta\"},{ \"item\":\"Chicken\"}, {\"item\":\"Yougurts\"}]}"



## Delete

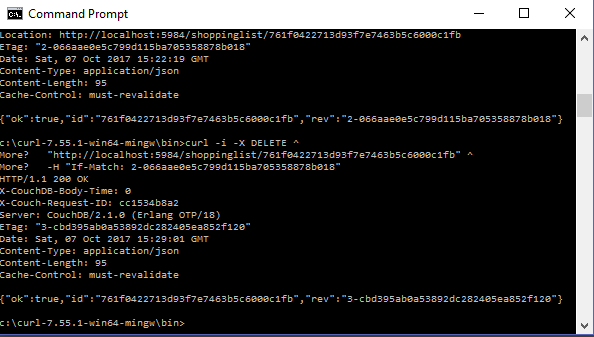
To delete a CouchDB document the DELETE method is used. This command needs to reference both the id and revision values of the document that is going to be deleted, much like the PUT method. Even though you have deleted the document, cURL will return a new rev value. This is because the documents are not actually removed from the disk it is overwritten with a new, empty.

Command:

curl -i -X DELETE ^

"http://localhost:5984/shoppinglist/761f0422713d93f7e7463b5c6000c1fb" ^

-H "If-Match: 2-066aae0e5c799d115ba705358878b018"



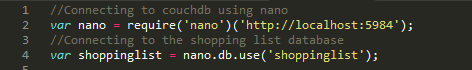
# Question 2

# Evaluation of API

For this project I chose to develop the application using JavaScript (through node). CouchDB is very intuitive to use from JavaScript, as they both deal with JSON exclusively. This made going down this route the obvious approach for me. As CouchDB is accessed via a standard REST API, it is possible to query it directly using either the http library that comes with node or a popular library such as Requests, for this project I chose to use the popular library “nano” which allows you to abstract this for ease of use. This seems to be, by far the most popular CouchDB library available in NPM, and for good reason: The library has now become part of the Apache CouchDB project. Nano is, by design, a minimalistic CouchDB driver. It features proxy requests from CouchDB directly to the end user and errors are proxied directly from CouchDB, this means it is easy for someone that is familiar with CouchDB to use nano. Nano allows you to create a very clean code base which is based entirely on your requests. Once nano is installed via npm, you can connect to CouchDB by using var nano = require(‘nano’)(‘http://localhost :5984’); once you are connect you can then start using the database by saying var shoppinglist = nano.db.use('shoppinglist'); . If a call back function is not specified but in nano it always receives three arguments err meaning an error, body meaning the http response body from CouchDB and header which is the http response header from CouchDB. The main disadvantage to using a library like nano is that the additional layer of abstraction will of course add some overhead that could be avoided by making the requests directly through CouchDBs REST API, but in my opinion this is easily offset by the easier implementation of the project using nano.

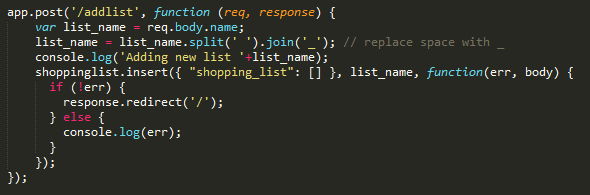
# CRUD through the API

## Connecting to CouchDB through nano



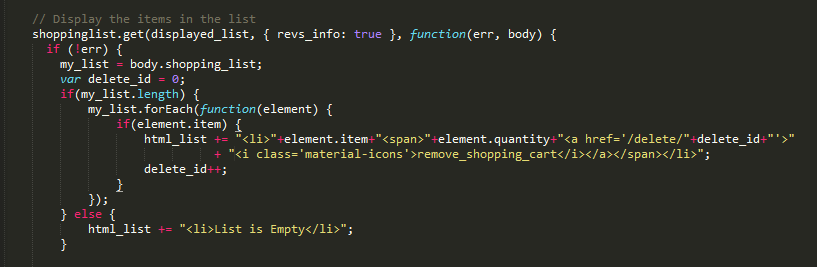
## Create

Creating a document through nano. This satisfies the CREATE requirement of CRUD, in the case of my project, each individual list is stored in its own document. So when a POST request is received by my application on the /addlist route, a new document is created. I do some minor string replacements at this stage to ensure that spaces are replaced with underscores.



## Read

Reading a document through nano. For my project, the READ functionality was what I implemented first. As each list is its own document, to display the list on screen, we first need to fetch the document from CouchDB. As the actual list is stored as an array contained within the document, I can then iterate through it using Array.forEach and convert the information contained into html for display. I do some minor validation checking here to enable me to handle empty lists.



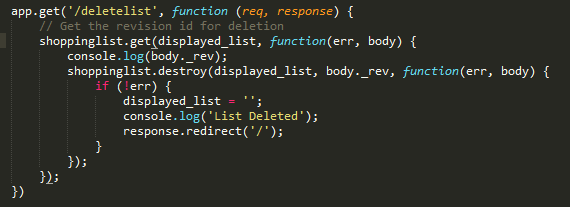
## Update

UPDATE is implemented in a couple of different ways within my app. As most of the data displayed on screen is contained within arrays in the individual documents, to add or remove an item from any given list requires first fetching the contents of this list, taking the array and modifying it (with Array.push() to add an item or Array.splice() to remove one) and then updating the CouchDB document with my new, altered document. We can then redirect the user back to the main, list overview page.

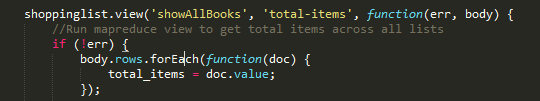


## Delete

My application allows users to delete existing lists. This requires performing a document deletion within CouchDB, once this is done I can redirect the user back to the overview page. The overview page also contains some functionality to ensure that the application does not accidentally try and display a document that does not exist any longer due to deletion like this.



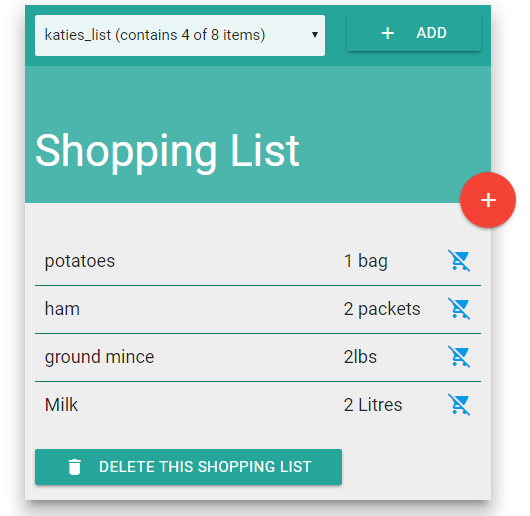
While I was initially unsure how best to use MapReduce in my application, I felt that perhaps displaying the total number of items across all lists might be a useful thing to have. To implement this the reducer simply counts the array elements in the “shopping\_list” attribute, by implementing this directly on CouchDB, it saves me having to manually iterate through them in JavaScript.



# GUI Interface

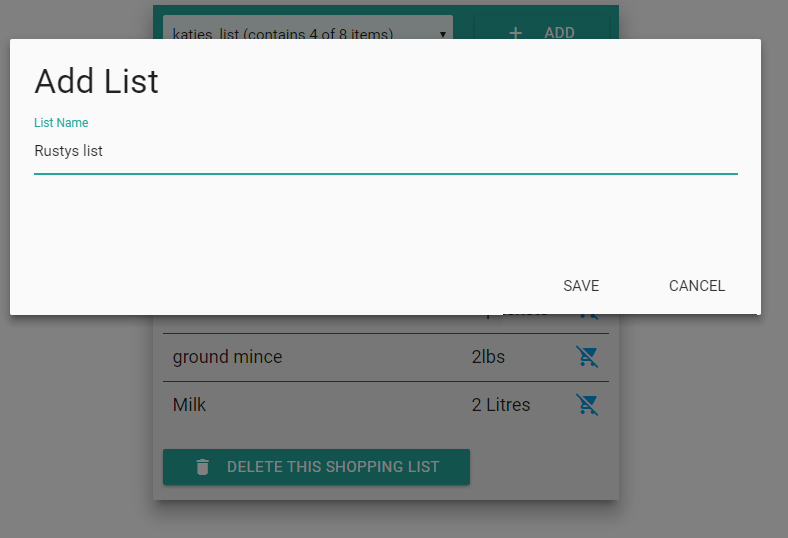
This is the initial screen when application is loaded. The top bar of the application is used to both see the individual documents (shopping lists) contained in the database and to change which list is currently being displayed. There is a minor bit of jQuery required for this to submit the form whenever a new value is chosen in the select. This uses both a Map (To get a list of the documents with the number of items contained within each) and a separate MapReduce to calculate the total number of items across all lists). The other features include:

* Add button, this opens a modal allowing the user to create a new document in the database. (CREATE)
* Plus button, allowing the user to add an item to the currently displayed document. (UPDATE)
* Shopping cart icon, which allows the user to remove a single item from the list. (UPDATE)
* Delete this shopping list button, this button deletes the entire document from the database and application. (DELETE)

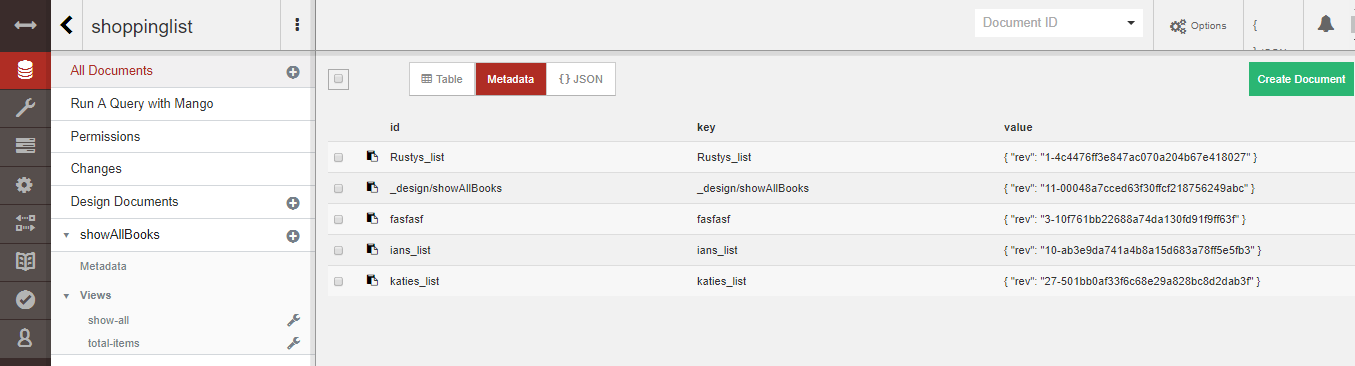


## Creation with the GUI

When you click on the add button this brings a modal containing a text field that allows you to enter the name of the list. This is then used for the id of the new document it creates. You can also cancel this if you change your mind.

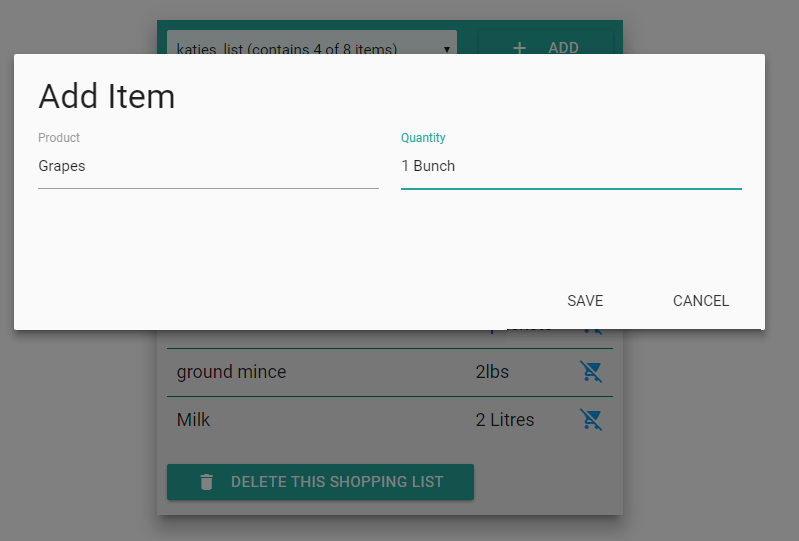


After this is created through the UI, the application updates the database by creating the new document.



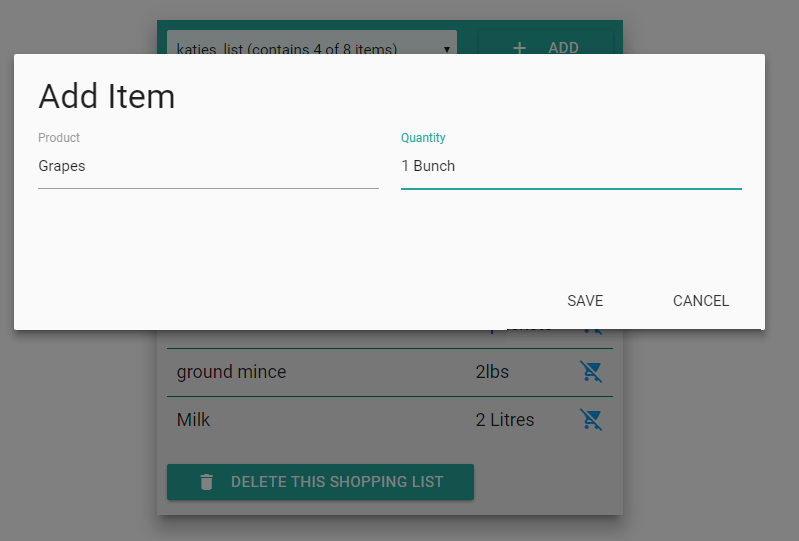
## Adding an Item to your shopping list with the GUI

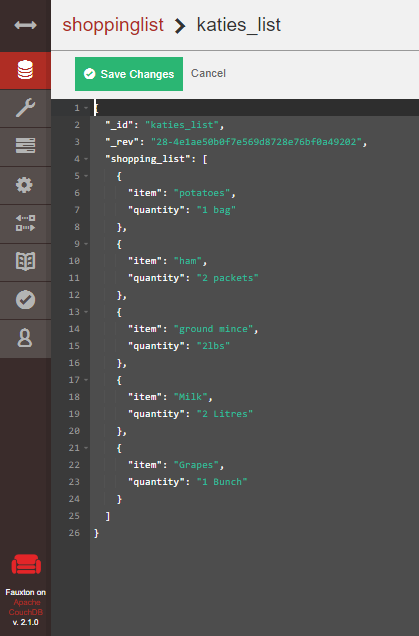
Adding an item to your shopping list can be done by clicking on the plus icon, which brings up the add item modal, you can then enter in the product name and the quantity for that product. While it would have been easily possible to limit a visitor to just using numbers for the quantity, I felt that it would be better to allow this to be saved as a free text field, allowing for quantities like “500g” or “5 bags”.



## Updating with the GUI

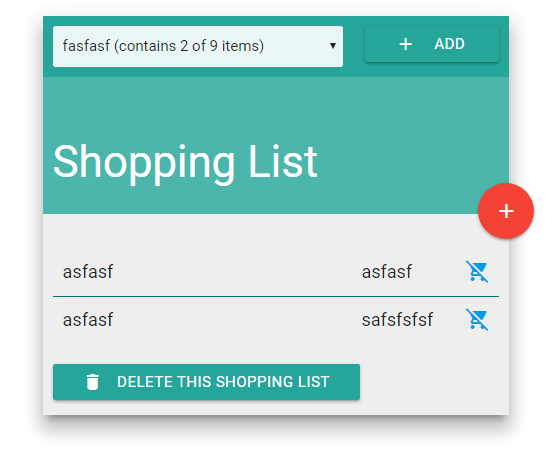
Once the visitor has updated their list via the GUI, the application will update the array contained within the selected document and then update the document in CouchDB to match the new array. The two images below show an item being added and CouchDB database being updated with the new item.

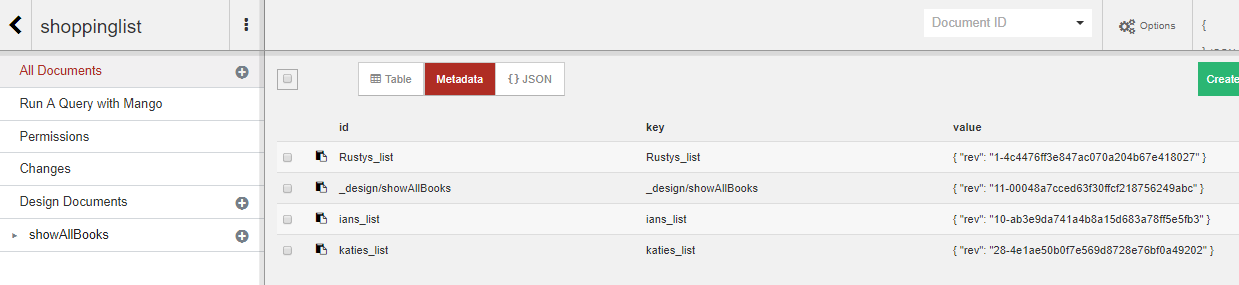




## Delete with the GUI

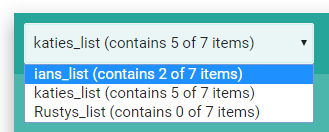
For a document to be deleted, you must select the document from the drop down menu and then click on the delete this shopping list button, this will then delete the document directly in CouchDB and redirect the user back to the most recently added list. This can be seen in the screen shot below.





## MapReduce with the GUI

This image shows the implantation of the mapreduce function. In my project, I use MapReduce to see how many items both in any are given shopping list as well as the total amount of items across all the stored shopping lists. In the real world, a user might have a separate shopping list for each store they plan on visiting (e.g. one Butchers list, and one TESCO list) so this enables them to see their total progress.



# Advantages and Disadvantages of the API

The main advantage to using nano is that the level of abstraction is fairly light. There is not much of a learning curve as in most cases it will return errors and data directly from CouchDB. Nano uses simplistic functions and huge web applications can be written while keeping a testable and maintainable structure. The only relevant disadvantage to using nano over the native http functionality or any other direct AJAX request is the additional overhead that it will bring to the project. As the library is so lightweight, this is minor.

# Conclusions

I found CouchDb very easy to learn and get to grips with. I found that the JSON documents that it stored were particularly well suited for an application like this. While researching I found it quite easy to find a reliable API and driver as the Apache foundation now maintains nano as part of its greater CouchDB project. JavaScript in my opinion was definitely the right choice to implement an application like this as it, of course, deals with JSON natively which gave an advantage in quickly setting up the documents and performing the CRUD operations. I found it difficult to implement the MapReduce functionality, as I could not find out how to run a Map without the reduce in the nano documentation. Ideally my project would not need the second map, as they are basically the same other than the fact that one contains a reducer it is likely that I would have eventually figured this out though. Overall I am quite pleased to the way the application turned out, with it also having the advantage of being displayed on an android phone with a little more time spent on the responsiveness of the interface, any visual bugs that currently exist on the smaller screen could have been completely resolved.