**CouchDB**

Tuesday 12 August 2014

09:35

Able to scale down as well as up, CouchDB fits problem spaces of varying size and complexity with ease.

CouchDB is the quintessential JSON- and REST-based document-oriented database.

First released in 2005, CouchDB was designed with the Web in mind and all the innumerable flaws, faults, failures, and glitches

Consequently, CouchDB offers a robustness unmatched by most other databases.

Whereas other systems tolerate occasional network drops, CouchDB thrives even when connectivity is only rarely available.

Somewhat like MongoDB, CouchDB stores *documents*—JSON objects consisting of key-value pairs where values may be any of several types, including other objects nested to any depth.

There is no ad hoc querying, though; indexed views produced by incremental mapreduce are the principal way you find documents.

Instead of focusing only on big-iron cluster installations, CouchDB aims to support a variety of deployment scenarios from the datacenter down to the smartphone.

You can run CouchDB on your Android phone, on your MacBook, and in your datacenter.

Written in Erlang, CouchDB is heartily built—the only way to shut it down is to kill the process!

With its append-only storage model, your data is virtually incorruptible and easy to replicate, back up, and restore.

CouchDB is document-oriented, using JSON as its storage and communication language.

All calls to CouchDB happen over its REST interface.

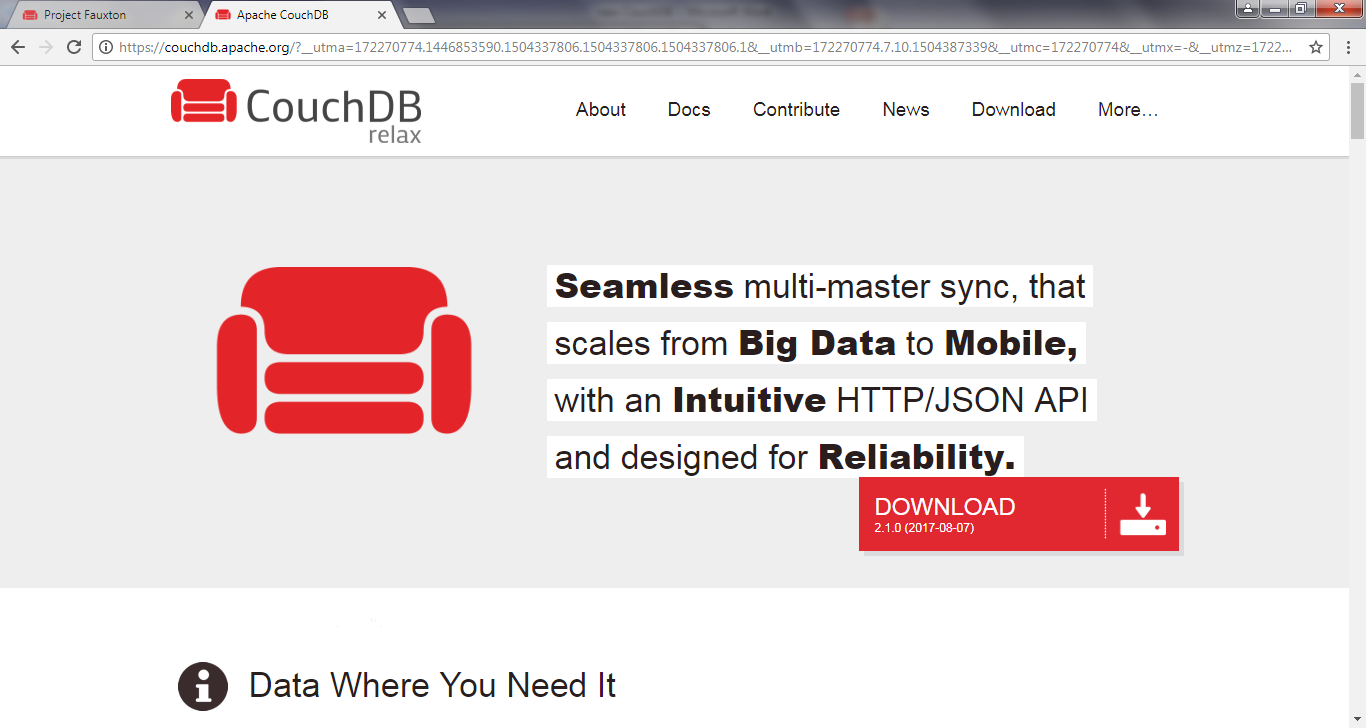
Replication can be one-way or bidirectional and ad hoc or continuous.

CouchDB gives you a lot of flexibility to decide how to structure, protect, and distribute your data.

Read the introduction at <http://docs.couchdb.org/en/2.1.0/intro/index.html>

**CRUD, Fauxton, and cURL**

**Download for Windows:** <http://couchdb.apache.org/>



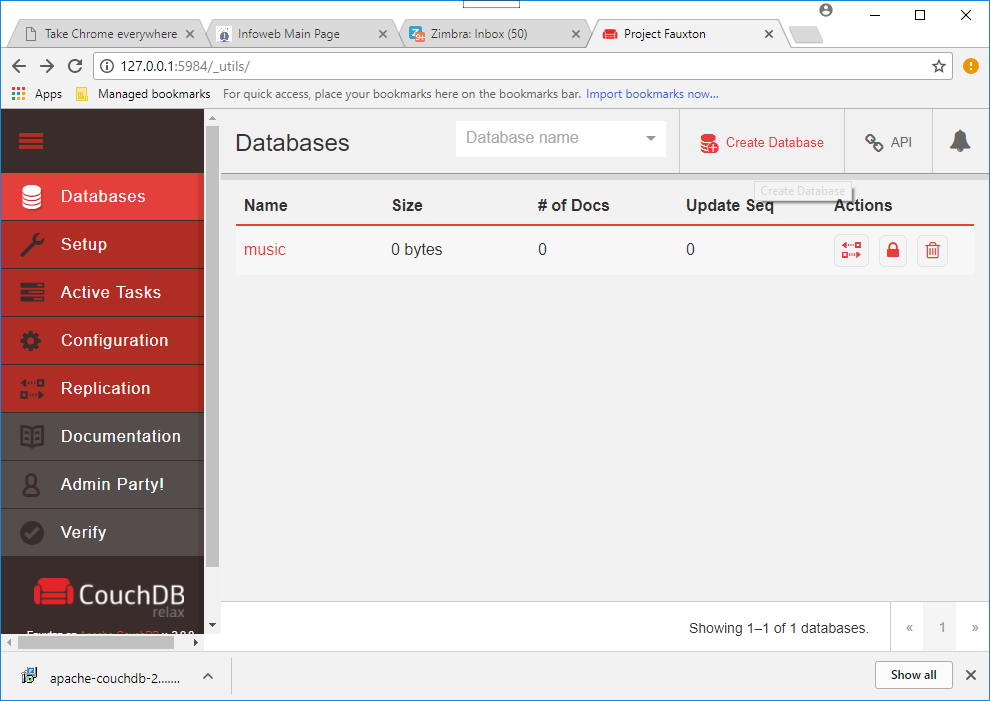
We’re going to kick-start our CouchDB exploration by using CouchDB’s friendly Fauxton web interface to perform basic CRUD operations. (http://couchdb.apache.org/fauxton-visual-guide/)

After that, we’ll use cURL to make REST calls.

All libraries and drivers for CouchDB end up sending REST requests under the hood, so it makes sense to start by understanding how they work.

**Getting Comfortable with** Fauxton

CouchDB comes with a useful web interface called Faxuton. Once you have CouchDB installed and running (start it from services if you can’t find an icon) , open a web browser to http://127.0.0.1:5984/\_utils/



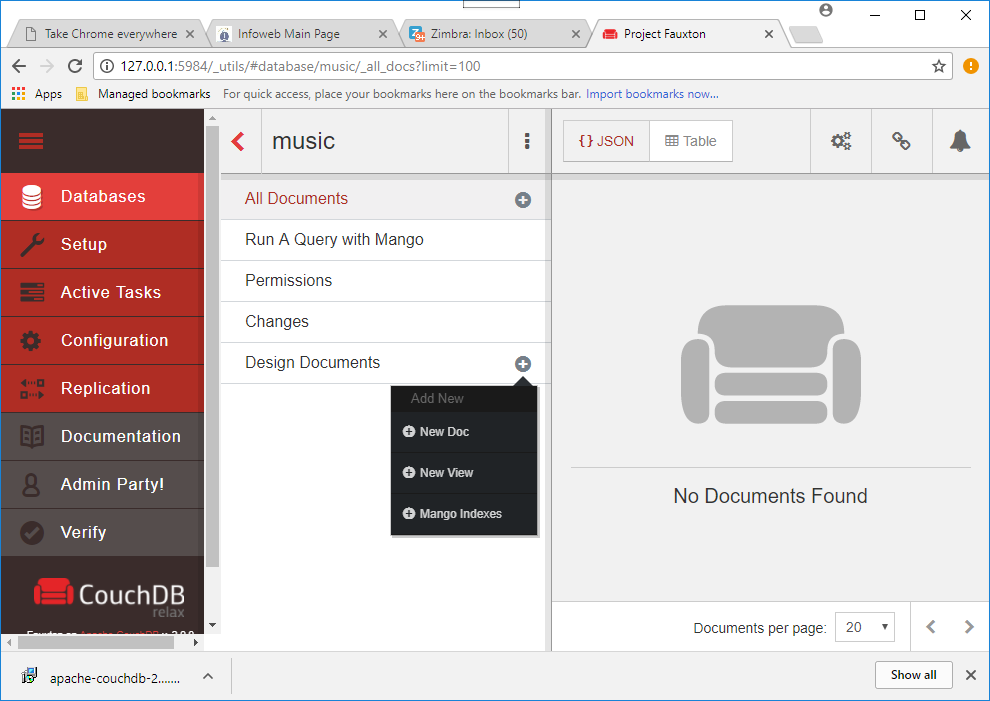
This will open the Overview page

Before we can start working with documents, we need to create a database to house them.

We’re going to create a database to store musicians along with their album and track data.

Click the Create Database... button.

In the pop-up, enter *music* and click Create.



This will redirect you automatically to the database’s page.

From here, we can create new documents or open existing ones.

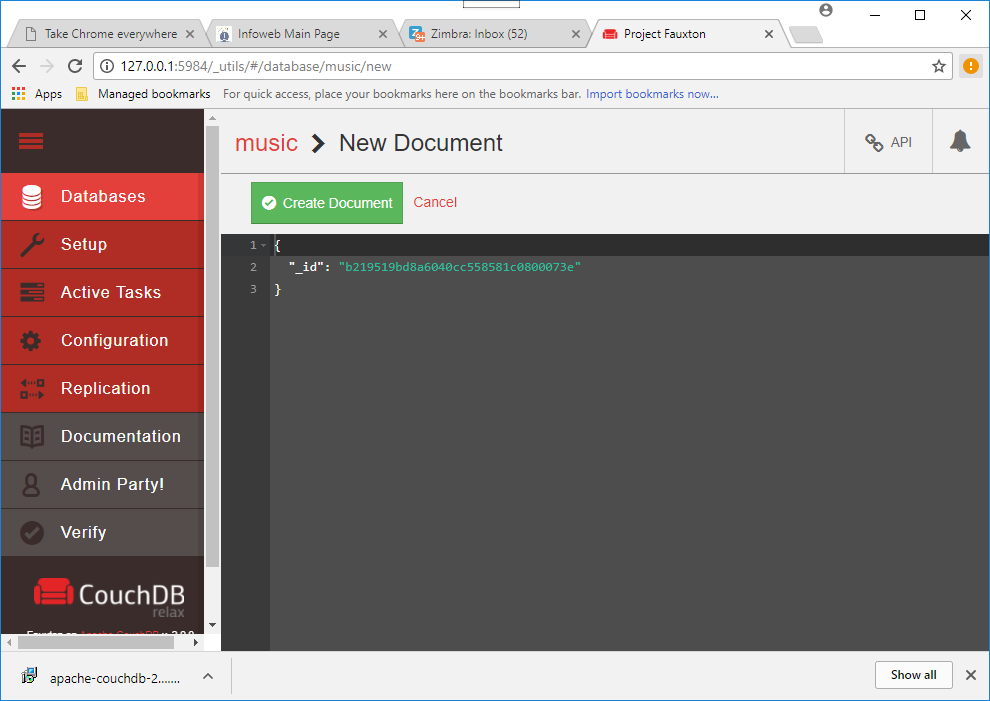
On the music database’s page, click the Design Document plus button and New Doc

This will take you to a new page

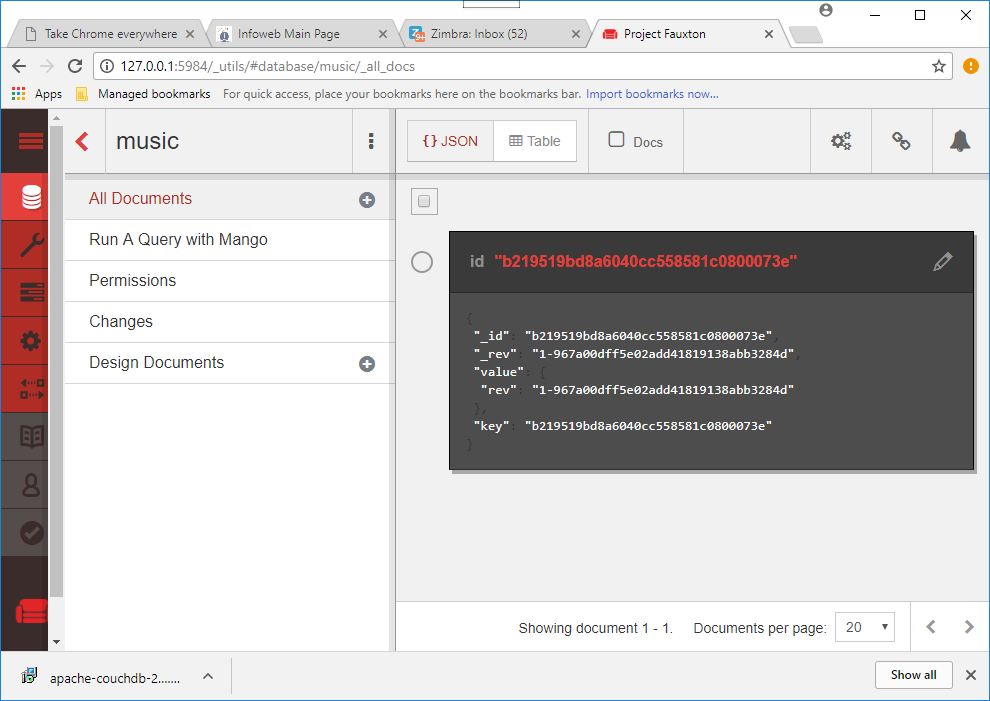
Just as in MongoDB (next), a document consists of a JSON object containing key- value pairs called *fields*.

All documents in CouchDB have an \_id field, which must be unique and can never be changed.

You can specify an \_id explicitly, but if you don’t, CouchDB will generate one for you.



In our case, the default is fine, so click Create Document to finish.



Immediately after saving the document, CouchDB will assign it an additional field called \_rev.

The \_rev field will get a new value every time the document changes.

The format for the revision string consists of an integer followed by a dash and then a pseudorandom unique string.

The integer at the beginning denotes the numerical revision—in this case 1.

Field names that begin with an underscore have special meaning to CouchDB, and \_id and \_rev are particularly important.

To update or delete an existing document, you must provide *both* an \_id and the matching \_rev.

If either of these do not match, CouchDB will reject the operation.

This is how it prevents conflicts—by ensuring only the most recent document revisions are modified.

There are no transactions or locking in CouchDB.

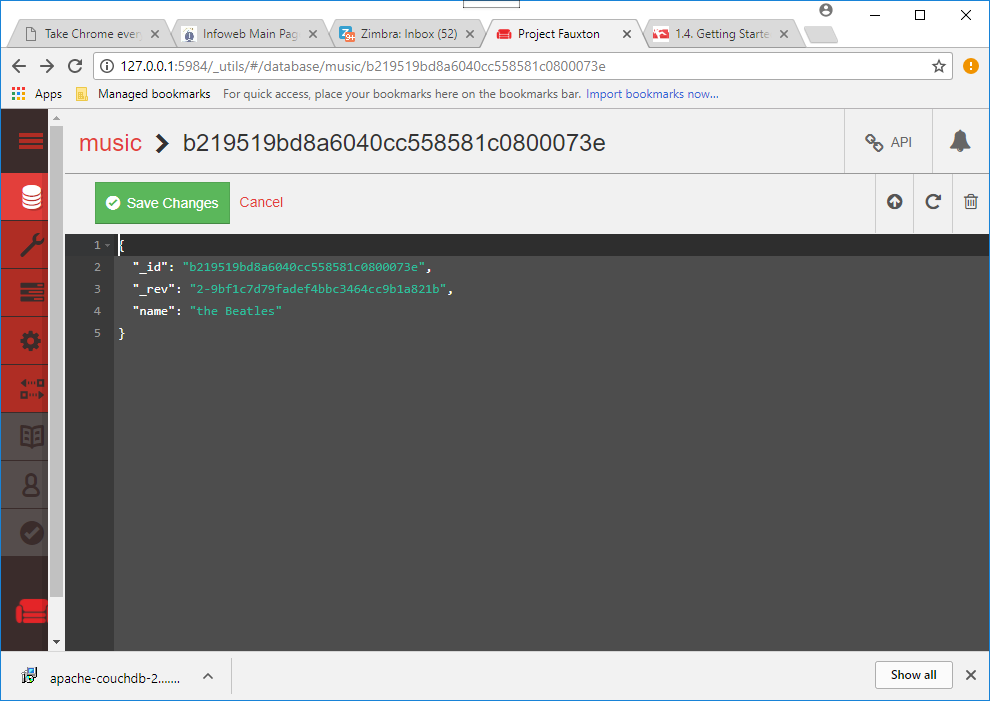
To modify an existing record, you first read it out, taking note of the \_id and \_rev.

Then, providing the you request an update by full document, including the \_id and \_rev.

All operations are first come, first served.

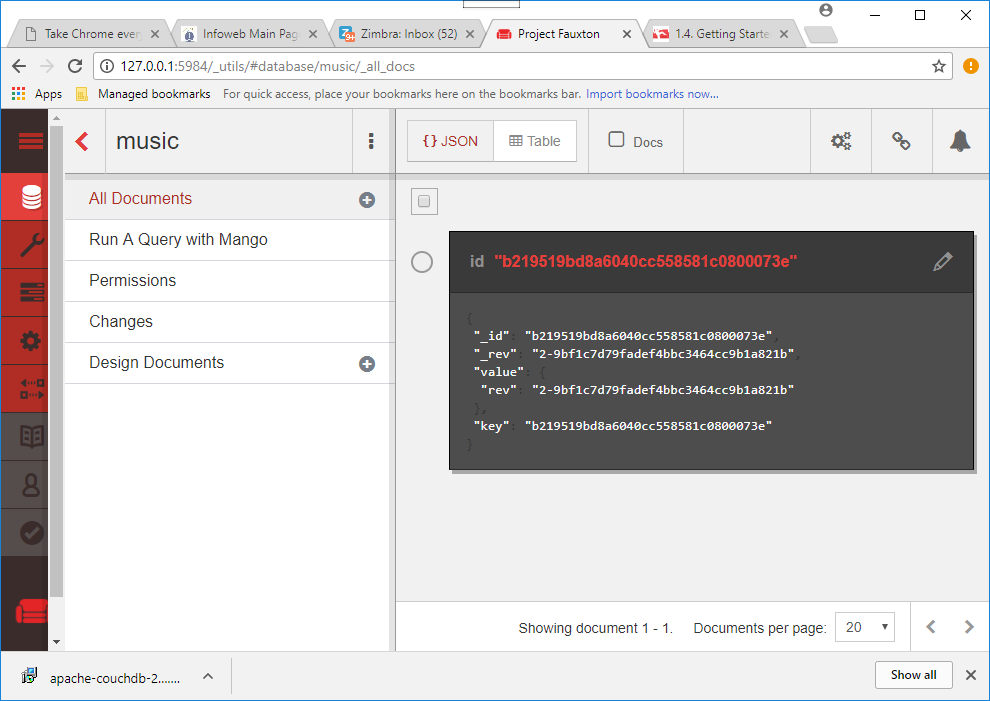
By requiring a matching \_rev, CouchDB ensures that the document you think you’re modifying hasn’t been altered behind your back while you weren’t looking.

To create a new field, simply use the editor to write valid JSON. Add a new field by appending a comma to the \_id value, then adding the text:



then click the Save Document button.

Notice how the \_rev field now begins with 2.



CouchDB is not limited to storing string values.

It can handle any JSON structure nested to any depth.

Click the Document again.

This time, set a field *albums*, and for Value enter the following (this is not an exhaustive list):

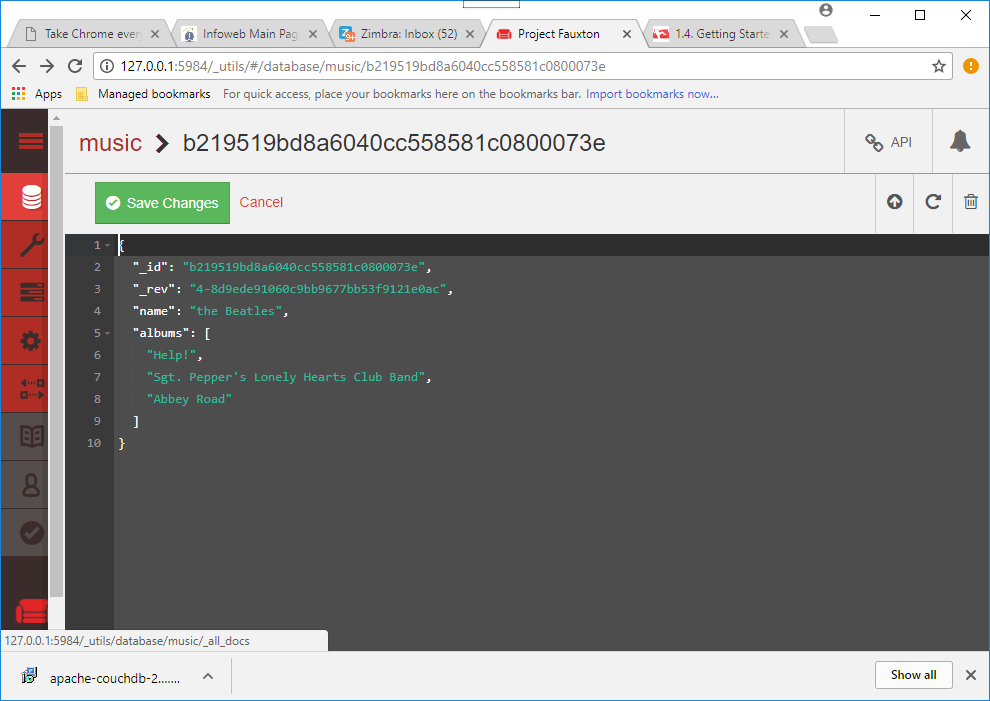
[

*"Help!"*,

*"Sgt. Pepper's Lonely Hearts Club Band"*,

*"Abbey Road"*

]



After this click save

There’s more relevant information about an album than just its name, so let’s add some.

Modify the albums field and replace the value you just set with this:

[{

"title": *"Help!"*, "year": 1965

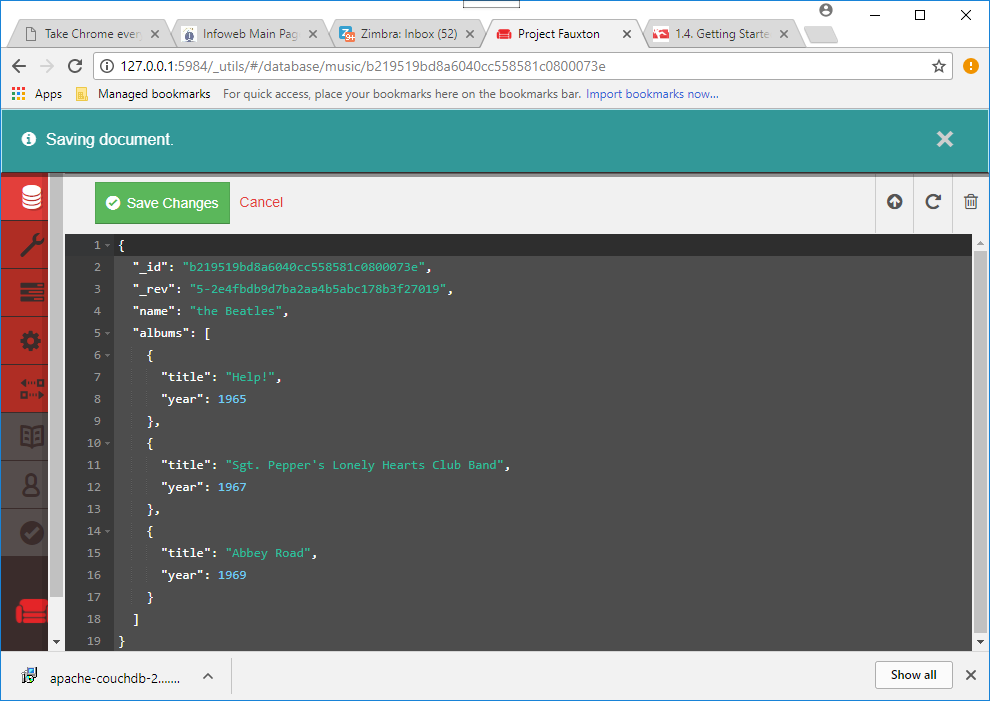
},{

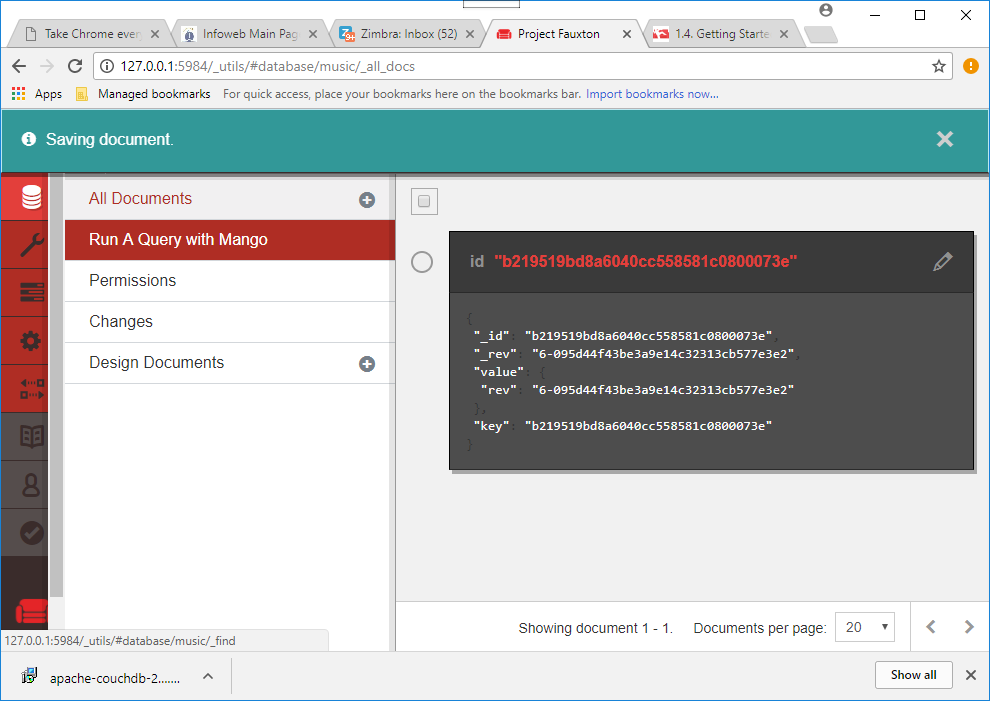
"title": *"Sgt. Pepper's Lonely Hearts Club Band"*, "year": 1967

},{

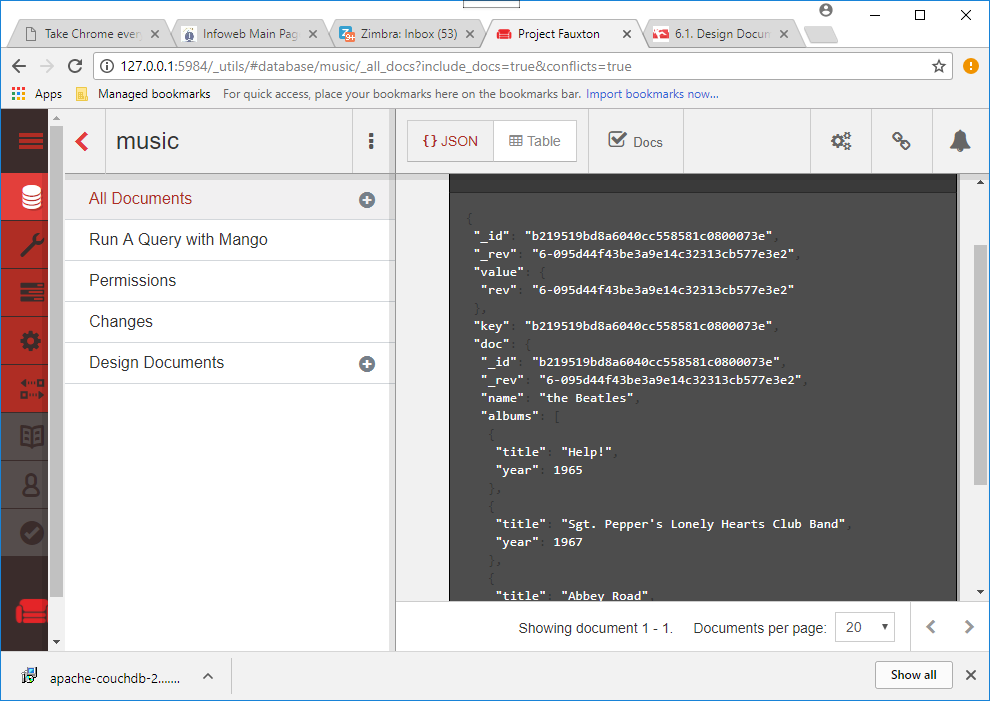
"title": *"Abbey Road"*, "year": 1969

}]

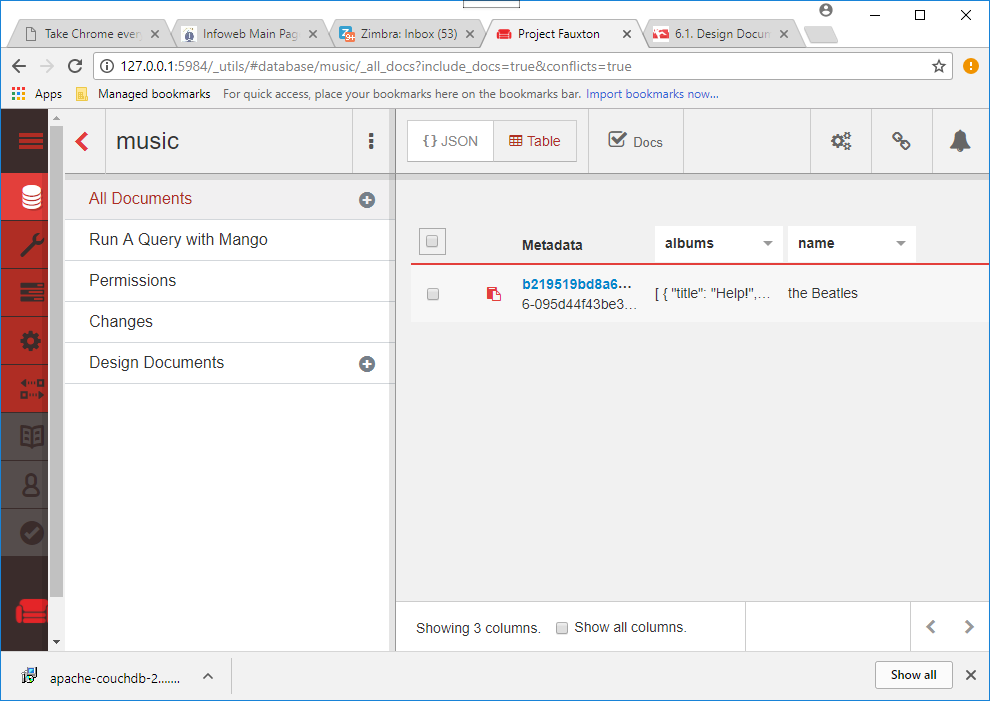




After you save the document, this time you should be able to expand the albums value to expose the nested documents underneath. If you select the Docs tab you will see the albums and the name



Or in table view



Clicking the Delete Document button would do what you might expect; it would remove the document from the music database.

But don’t do it just yet.

Instead, let’s drop down to the command line and take a look at how to communicate with CouchDB over REST.

**Performing RESTful CRUD Operations with cURL**

All communication with CouchDB is REST-based, and this means issuing commands over HTTP.

we can communicate with CouchDB using the command-line tool cURL.

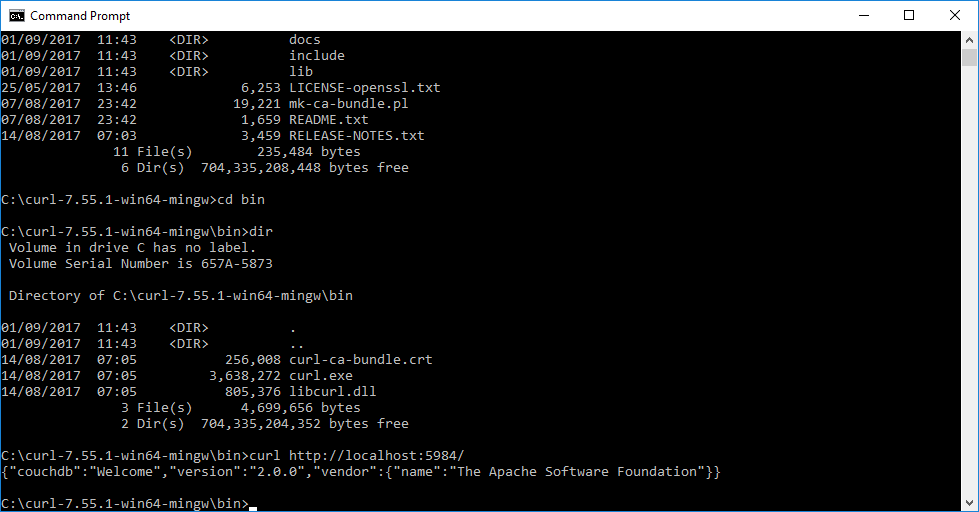
Here we’ll perform some basic CRUD operations before moving on to the topic of views.

To start, open a command prompt and run the following:

(you can get curl here <https://curl.haxx.se/dlwiz/?type=bin>

**curl** [**http://localhost:5984/**](http://localhost:5984/)

{"couchdb":"Welcome","version":"1.6.1"}

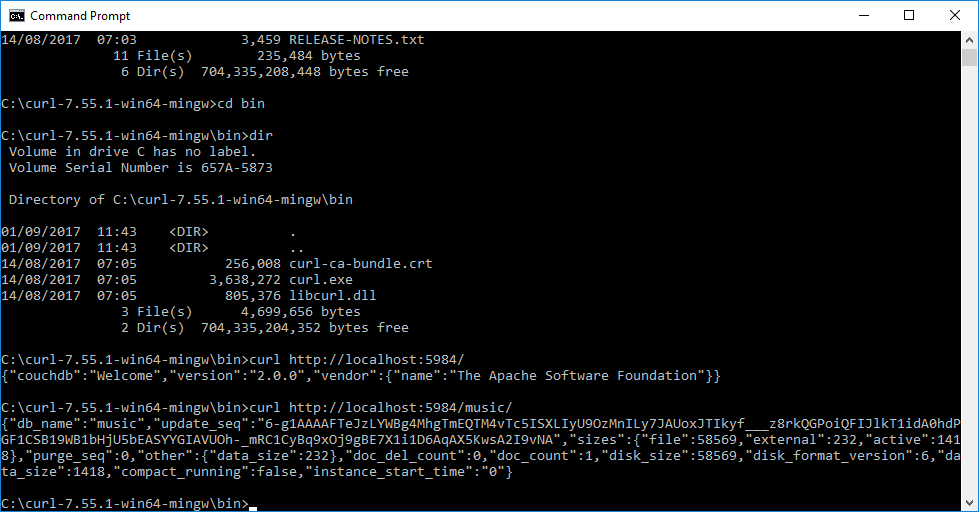


Issuing GET requests (cURL’s default) retrieves information about the thing indicated in the URL.

Accessing the root as you just did merely informs you that CouchDB is up and running and what version is installed.

Next let’s get some information about the music database we created earlier (output formatted here for readability):

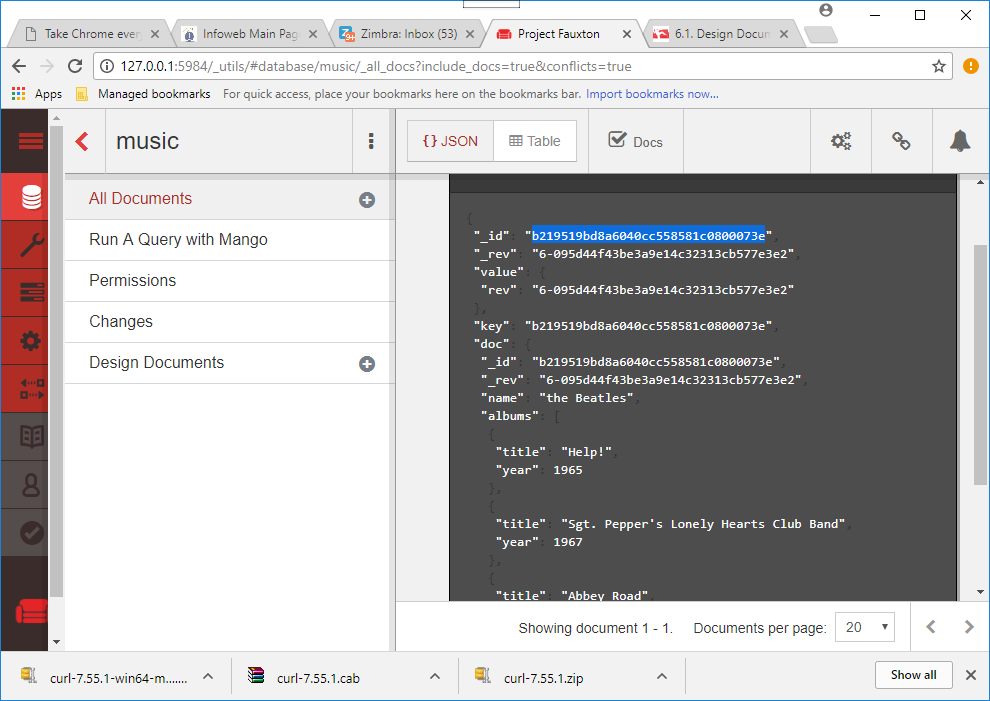
**curl** [**http://localhost:5984/music/**](http://localhost:5984/music/)



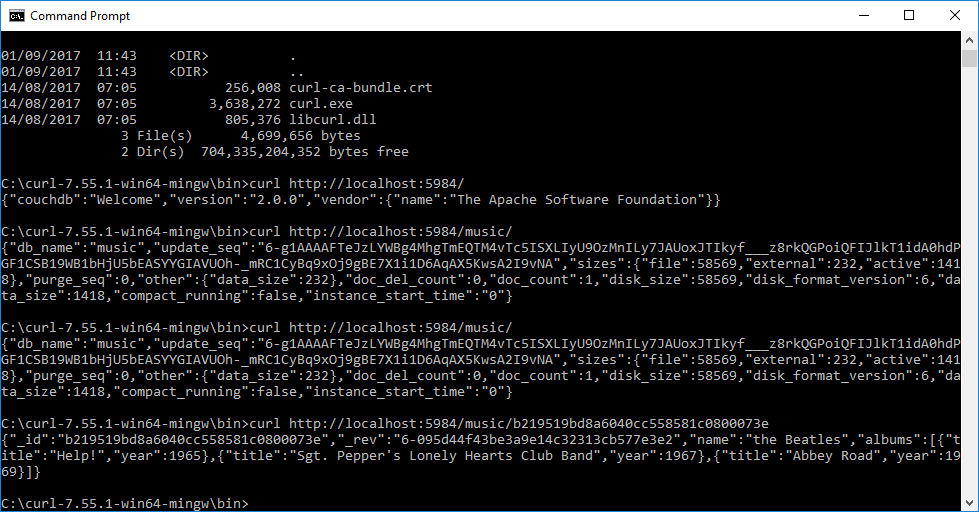
This returns some information about how many documents are in the database, how long the server has been up, and how many operations have been performed.

**Reading a Document with GET**

To retrieve a specific document, append its \_id to the database URL like so:



**curl http://localhost:5984/music/b219519bd8a6040cc558581c0800073e**



In CouchDB, issuing GET requests is always safe.

CouchDB won’t make any changes to documents as the result of a GET.

To make changes, you have to use other HTTP commands like PUT, POST, and DELETE.

Curl commands: <http://curl.haxx.se/docs/manpage.html>

The DOS environment doesn't seem to like single quotes. Any example that uses single quotes to surround values is invalid on windows. The single quotes must be replaced with double quotes.

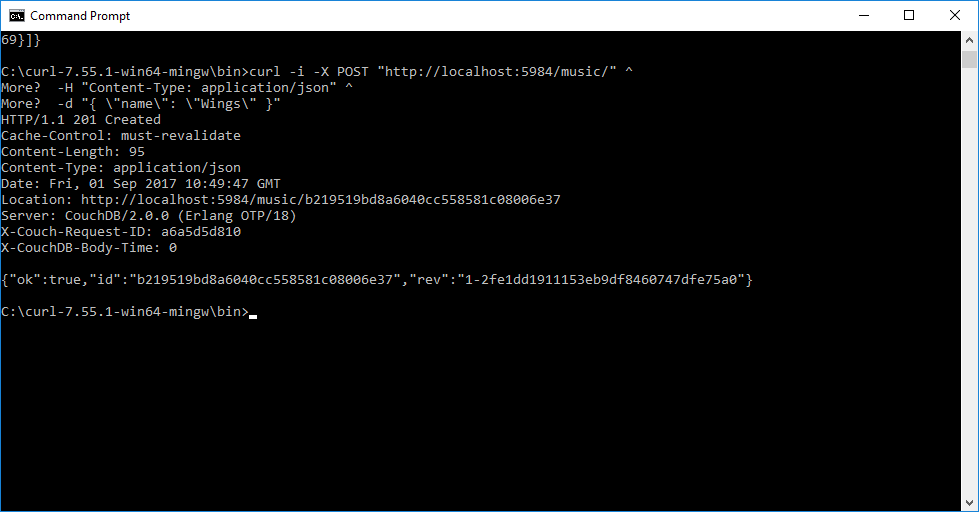
|  |  |
| --- | --- |
| 1down vote | My curl.exe eats only -d "{\"param\":\"value\"}"  i.e. doublequotes around data, and doublequotes masked with backslash inside |

**Creating a Document with POST**

To create a new document, use POST.

Make sure to specify a Content-Type header with the value *application/json*; otherwise, CouchDB will refuse the request. To separate lines use ^ in Dos (\ in Linux)

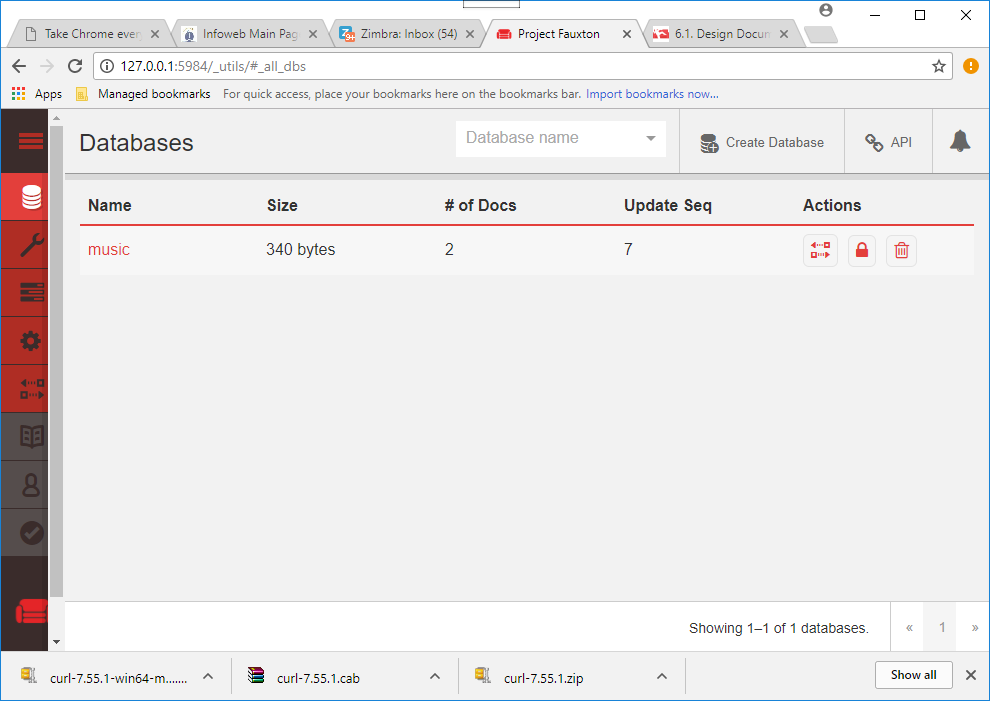
**curl -i -X POST "**[**http://localhost:5984/music/**](http://localhost:5984/music/)**" ^  
 -H "Content-Type: application/json" ^  
 -d "{ \"name\": \"Wings\" }"**

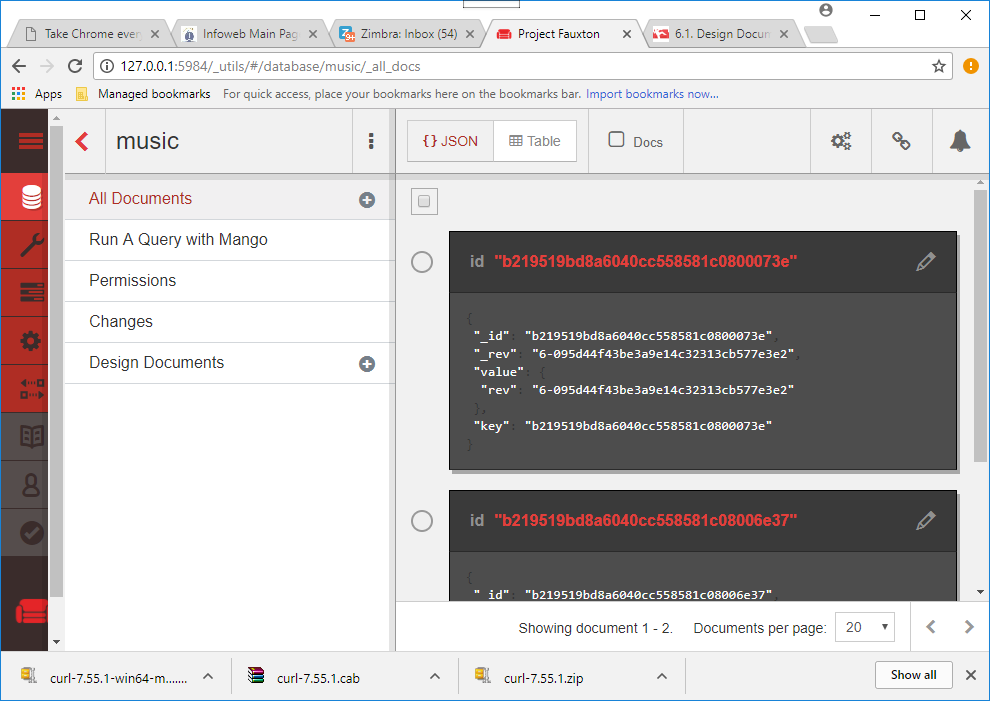


 The HTTP response code 201 Created tells us that our creation request was successful.

The body of the response contains a JSON object with useful information such as the \_id and \_rev values.

You can check Fauxton for two documents now:





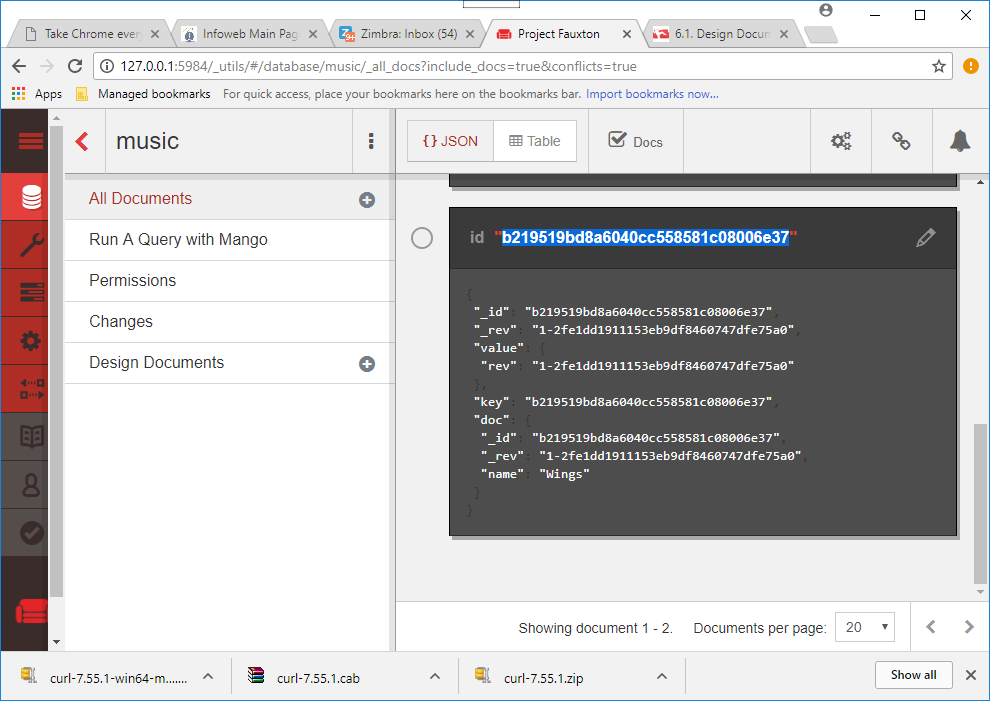
**Updating a Document with PUT**

The PUT command is used to update an existing document or create a new one with a specific \_id.

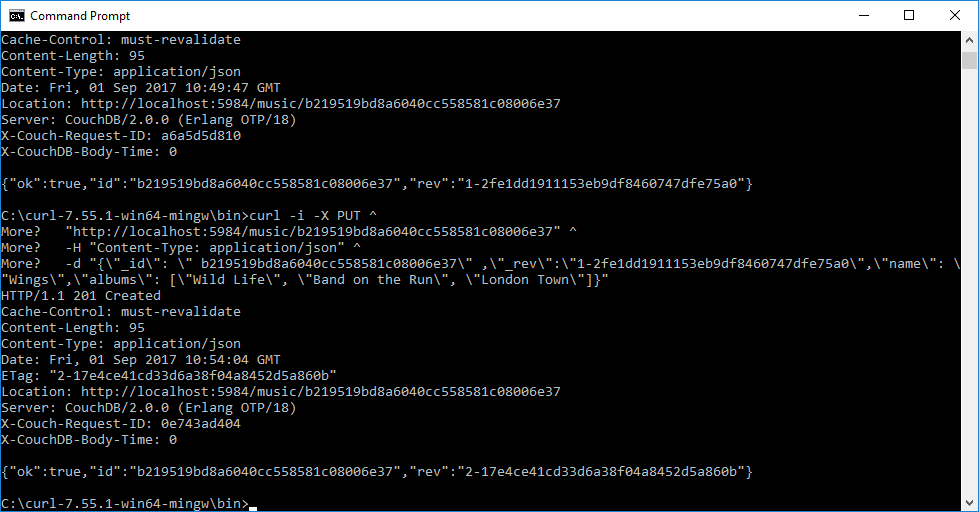
Both the \_id and \_rev fields must exactly match the document being updated, or the operation will fail.

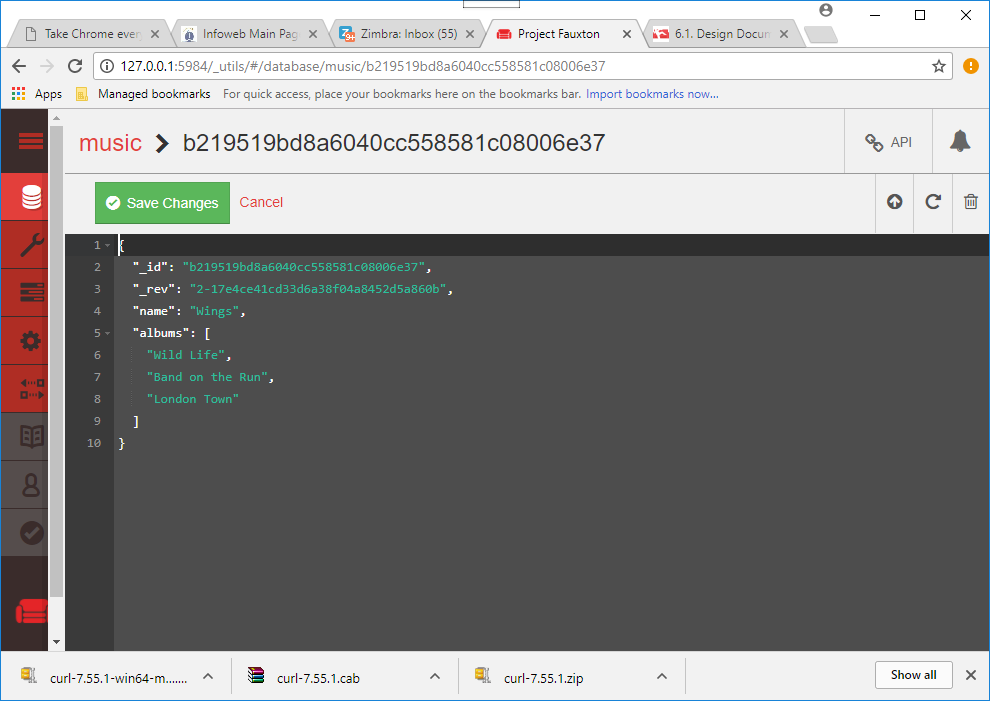
You will need to use the \_id and \_rev of the document you just created, you can copy it from Fauxton.

Just like GET, the URL for a PUT URL consists of the database URL followed by the document’s \_id.



**curl -i -X PUT ^  
 "http://localhost:5984/music/3492796e7586fd13daee913d10002a30" ^  
 -H "Content-Type: application/json" ^  
 -d "{**\"\_id\": \"3492796e7586fd13daee913d10002a30\" ,\"\_rev\":\"1-2fe1dd1911153eb9df8460747dfe75a0\",\"name\": \"Wings\",\"albums\": [\"Wild Life\", \"Band on the Run\", \"London Town\"]}**"**





**(I have created this database for you called music.couch, paste it into the C:\Program Files (x86)\Apache Software Foundation\CouchDB\var\lib\couchdb folder)**

**And again updating the document (with added titles) – already done!**

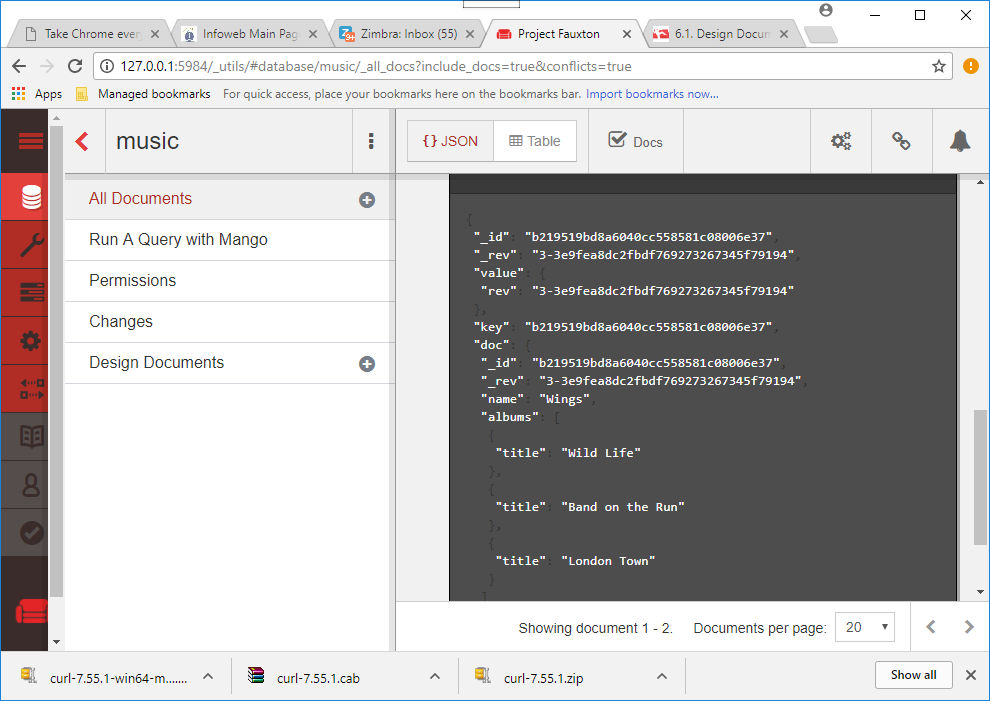
**curl -i -X PUT ^**

**"http://localhost:5984/music/ff964a202868bee9ce26351abf006461" ^**

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

**-d "{**\"\_id\":\"ff964a202868bee9ce26351abf006461\" ,\"\_rev\":\"2-17e4ce41cd33d6a38f04a8452d5a860b\",\"name\": \"Wings\",\"albums\":[{\"title\":\"Wild Life\"},{ \"title\":\"Band on the Run\"}, {\"title\":\"London Town\"}]}**"**





Unlike MongoDB, in which you modify documents *in place*, with CouchDB you always overwrite the entire document to make any change.

The Fuxton web interface we saw earlier may have made it look like you could modify a single field in isolation, but behind the scenes it was rerecording the whole document when you hit Save.

As we mentioned earlier, both the \_id and \_rev fields must exactly match the document being updated, or the operation will fail.

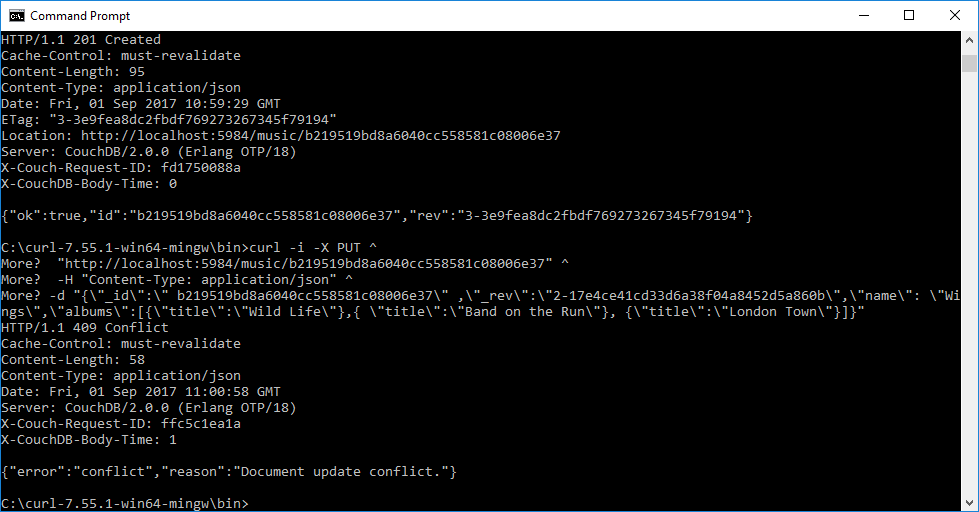
To see how, try executing the same PUT operation again.

**curl -i -X PUT ^**

**"http://localhost:5984/music/**b219519bd8a6040cc558581c08006e37**" ^**

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

**-d "{**\"\_id\":\" b219519bd8a6040cc558581c08006e37\" ,\"\_rev\":\"2-17e4ce41cd33d6a38f04a8452d5a860b\",\"name\": \"Wings\",\"albums\":[{\"title\":\"Wild Life\"},{ \"title\":\"Band on the Run\"}, {\"title\":\"London Town\"}]}**"**



You’ll get an HTTP 409 Conflict response with a JSON object describing the problem.

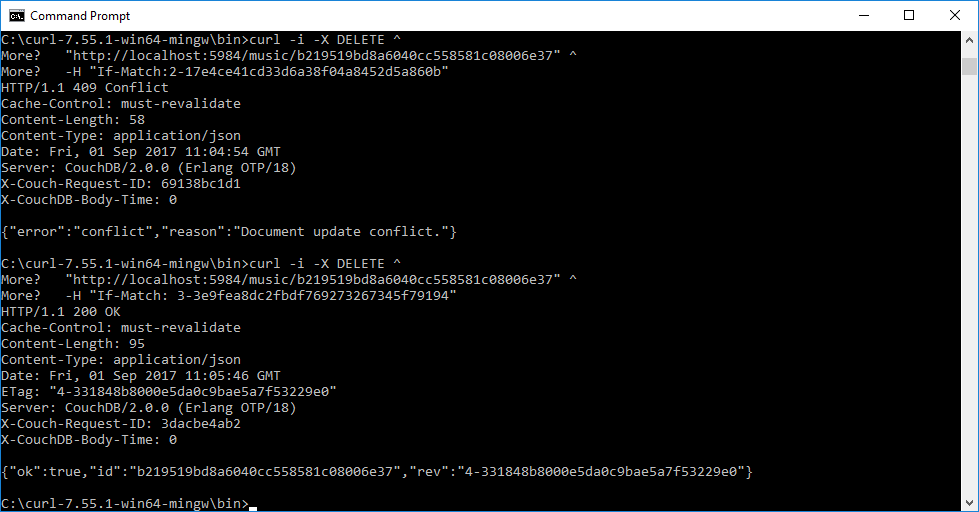
This is how CouchDB enforces consistency.

Removing a Document with DELETE

Create a new empty document for this exercise

Finally, we can use the DELETE operation to remove a document from the database.

**curl -i -X DELETE ^  
 "**[**http://localhost:5984/music/**](http://localhost:5984/music/b5dbb285f0872881c76ea01707001a04)**60b872436acb5301b1b770aa67009b49" ^  
 -H "If-Match:** 1-967a00dff5e02add41819138abb3284d**"**



The DELETE operation will supply a new revision number, even though the document is gone.

It’s worth noting that the document wasn’t really removed from disk, but rather a new empty document was appended, flagging the document as deleted.

Just like with an update, CouchDB does not modify documents in place.

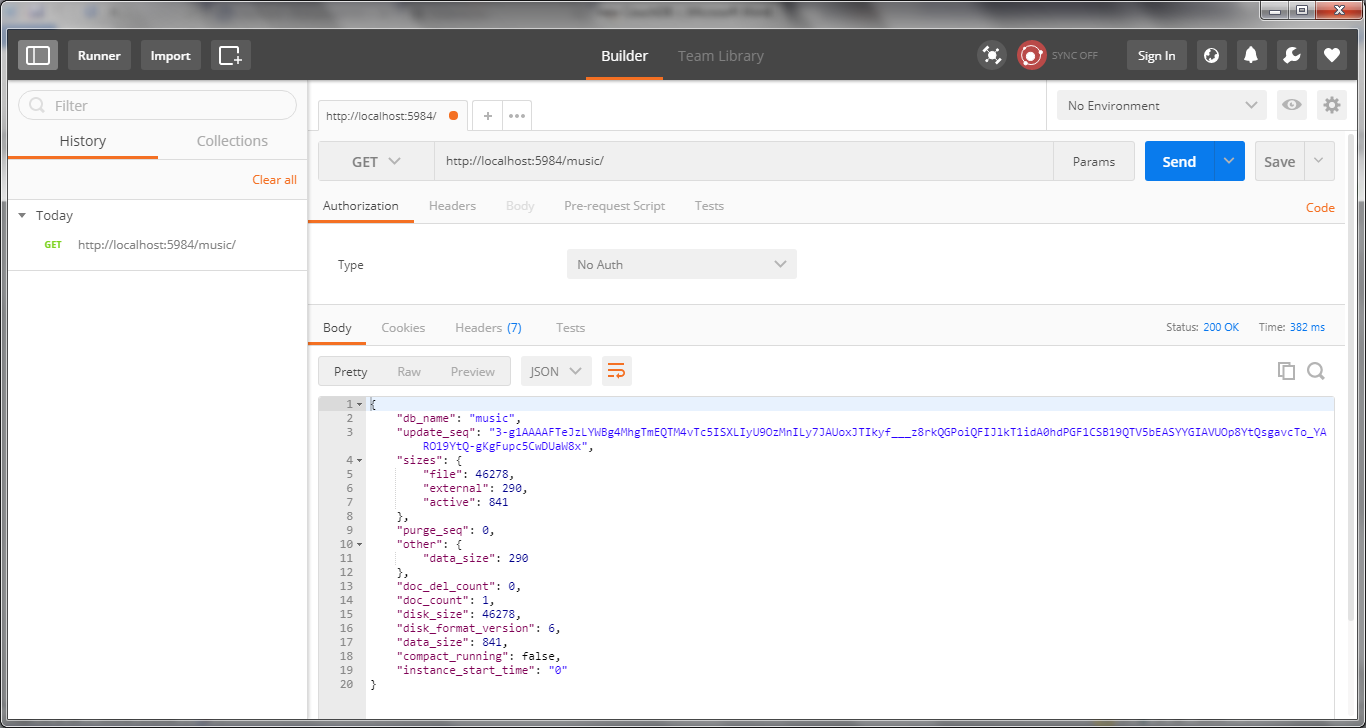
But for all intents and purposes, it’s deleted.

 Next we’ll dig into creating indexed *views*, which will provide other avenues for retrieving documents than just specifying them by their \_id values.

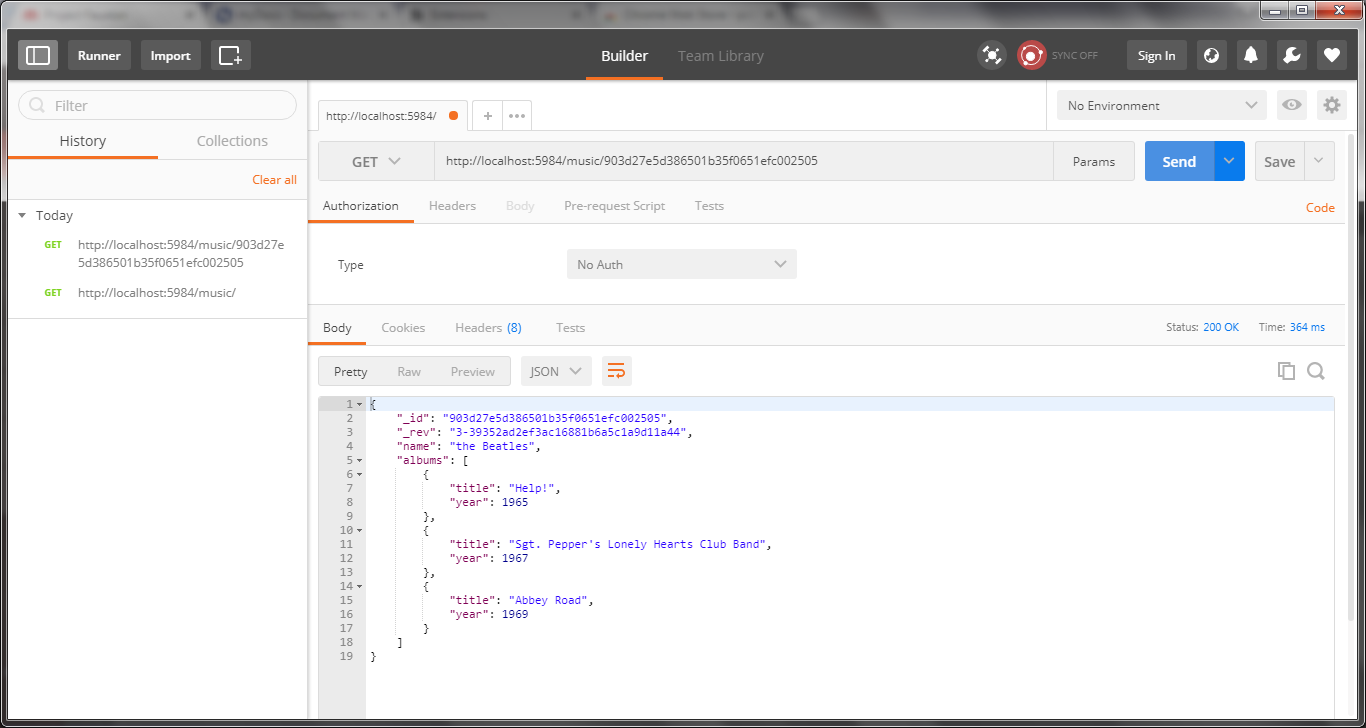
Exercise

Try installing Postman on chrome to generate http messages – nicer than curl

(Also generated curl code, java code, c# code etc)



And



Try doing POST via Postman

**Creating and Querying Views**

In CouchDB, a *view* is a window into the documents contained in a database.

Views are the principal way that documents are accessed in all but trivial cases—like those individual CRUD operations we saw earlier

Views are useful for many purposes:

* Filtering the documents in your database to find those relevant to a particular process.
* Extracting data from your documents and presenting it in a specific order.
* Building efficient indexes to find documents by any value or structure that resides in them.
* Use these indexes to represent relationships among documents.
* Finally, with views you can make all sorts of calculations on the data in your documents. For example, if documents represent your company’s financial transactions, a view can answer the question of what the spending was in the last week, month, or year.

**Accessing Documents Through Views**

A view consists of mapper and reducer functions that are used to generate an ordered list of key-value pairs.

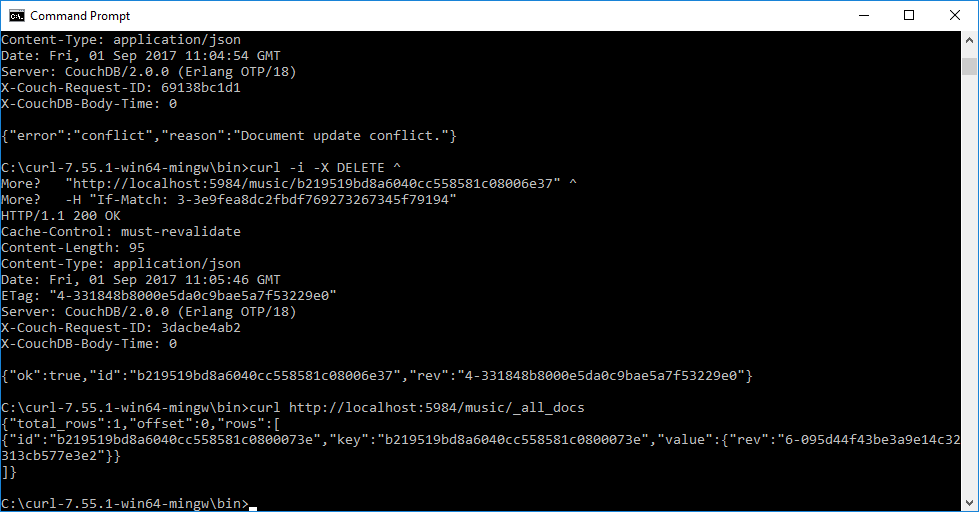
Both keys and values can be any valid JSON.

The simplest view is called \_all\_docs.

It is provided out of the box for all databases and contains an entry for each document in the database, keyed by its string \_id.

To retrieve all the things in the database, issue a GET request for the \_all\_docs view.

**curl** [**http://localhost:5984/music/\_all\_docs**](http://localhost:5984/music/_all_docs)



You can see in the previous output the two documents we’ve created so far.

The response is a JSON object that contains an array of rows.

Each row is an object with three fields:

• id is the document’s \_id.

• key is the JSON key produced by the mapreduce functions.

• value is the associated JSON value, also produced through mapreduce.

In the case of \_all\_docs, the id and key fields match, but for custom views this will almost never be the case.

By default, views won’t include all of each document’s content in the value returned.

To retrieve all of the document’s fields, add the include\_docs=true URL parameter.

**curl** [**http://localhost:5984/music/\_all\_docs?include\_docs=true**](http://localhost:5984/music/_all_docs?include_docs=true)



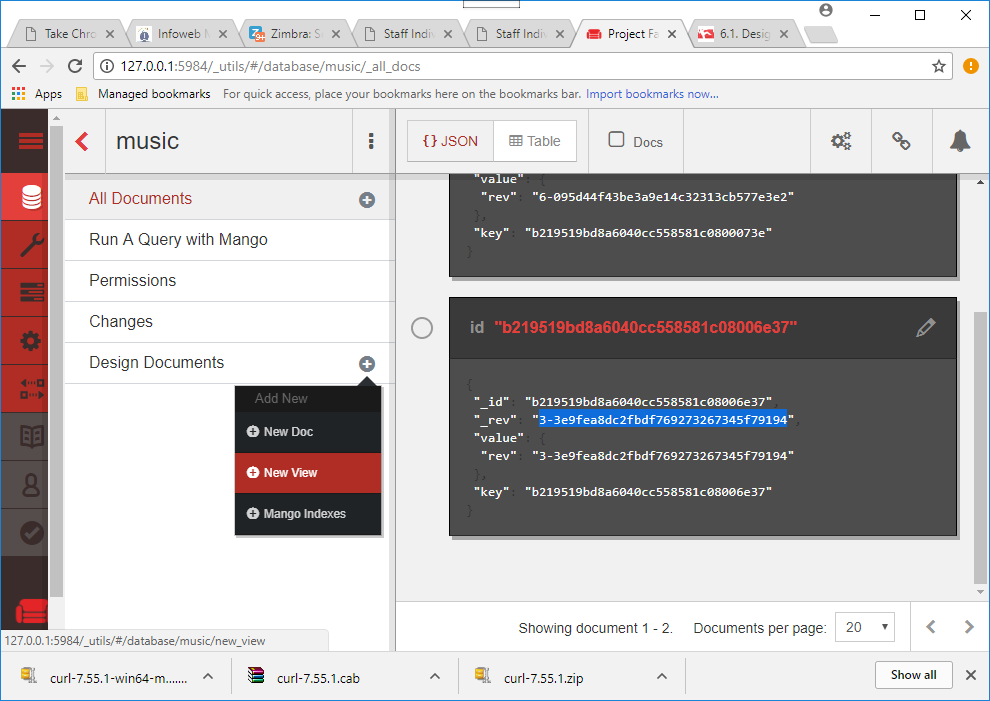
Here you can see that the other properties name and albums have been added to the value object in the output.

**Writing Your First View**

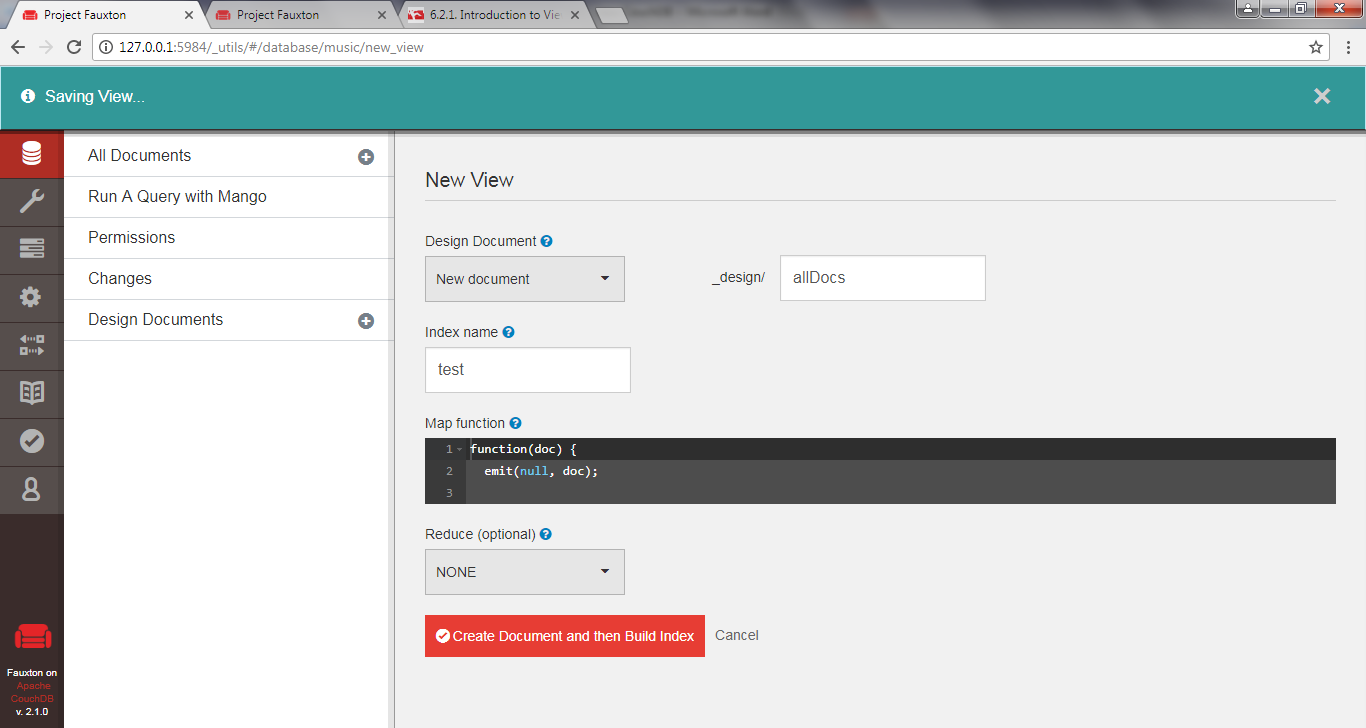
 To start, we’ll reproduce the behavior of the \_all\_docs view, and after that, we’ll make increasingly complex views to extract deeper information from our documents for indexing.

To create a view, open a browser to Fauxton

Next open the music database by clicking the link.



In the design document select new View



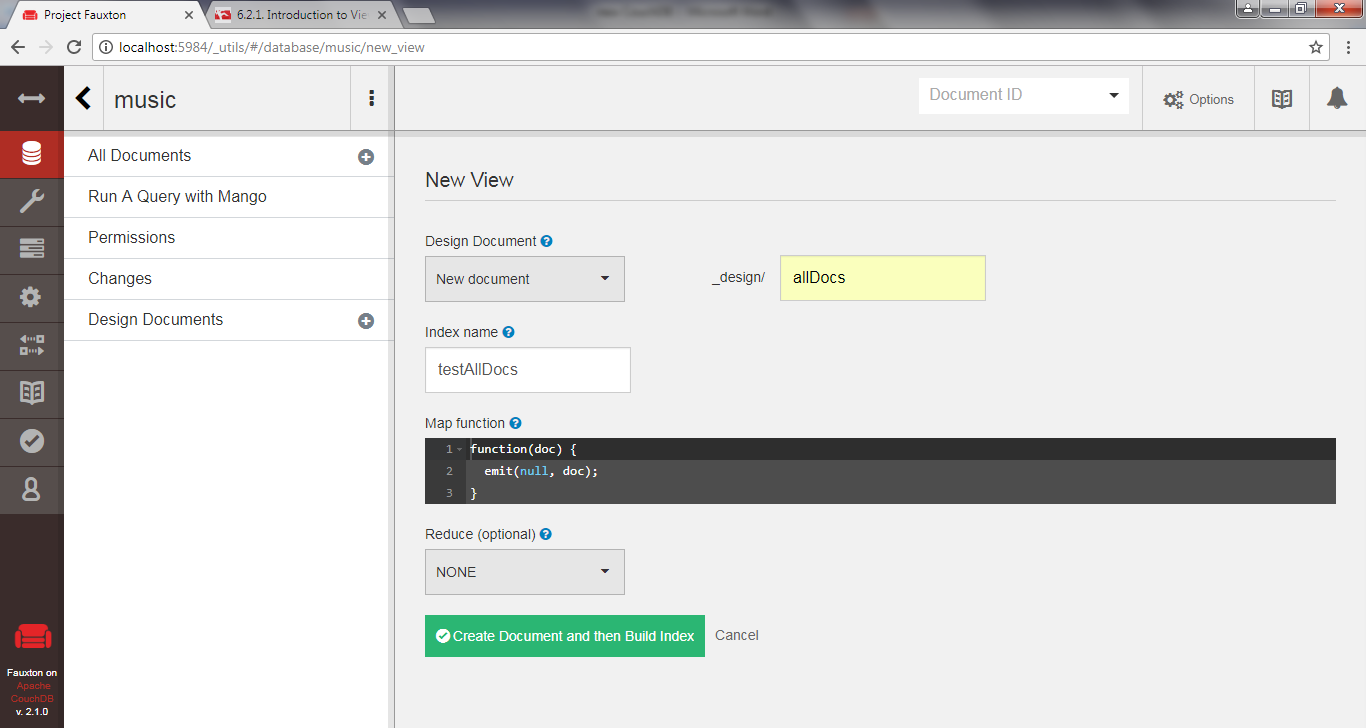
Call The view allDocs and create an index named test

Edit the code in the left Map Function box should look like this:

function(doc) {  
 emit(null, doc);

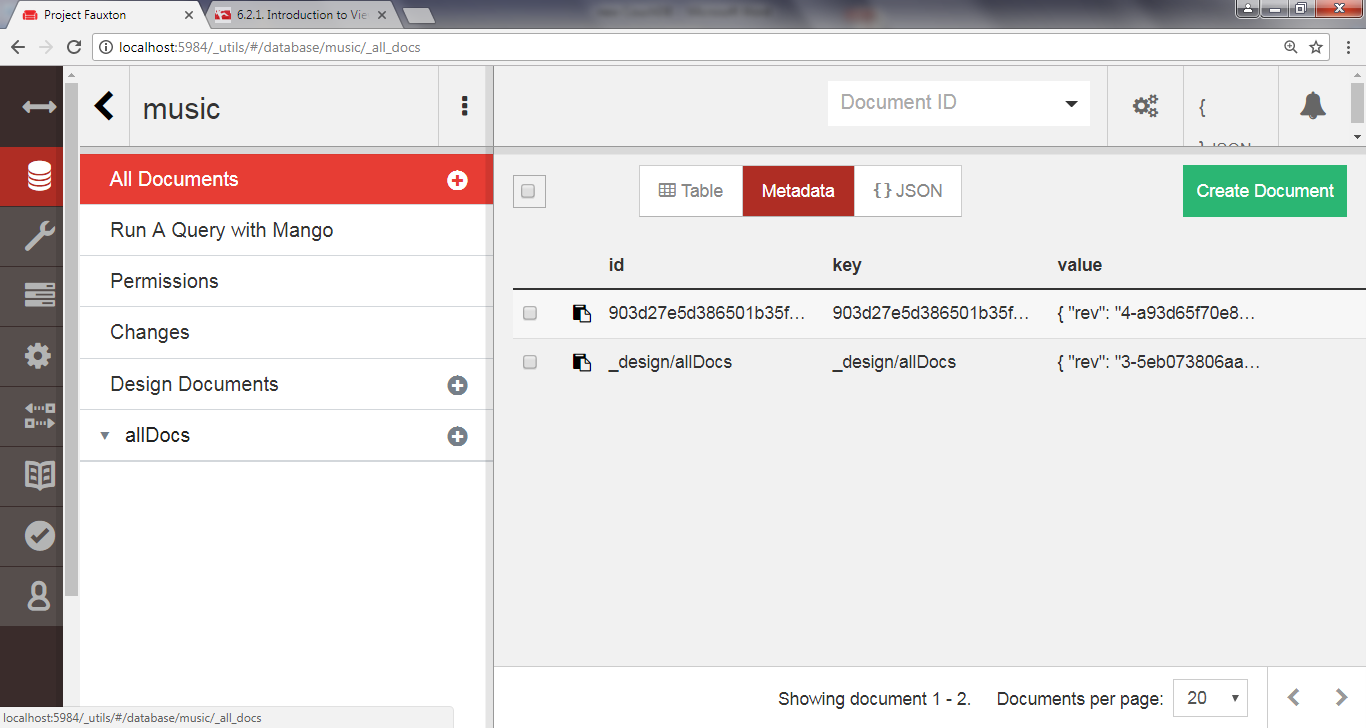
}

Change the index name to allDocs as well (an index can be used to quickly find documents later.



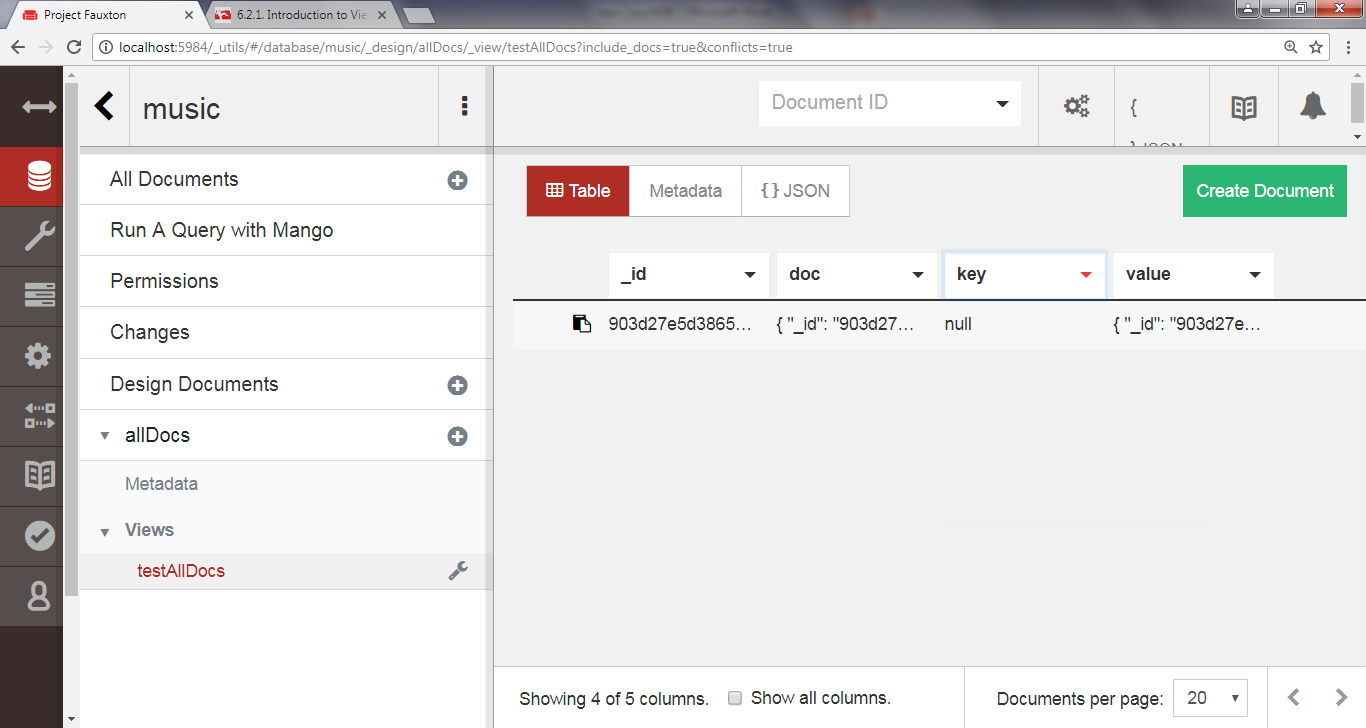
HERE \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_10/9

If you look at All Documents you will see the view and the document



When you select the view using its index name (testAllDocs) CouchDB will execute this function once for each document in the database, passing in that document as the doc parameter each time.

This will generate a table with a single row of results resembling the following:



The secret to this output, and all views, is the emit() function.

emit takes two arguments: the key and the value.

A given map function may call emit one time, many times, or no times for a given document.

In the previous case, the map function emits the key-value pair null/doc.

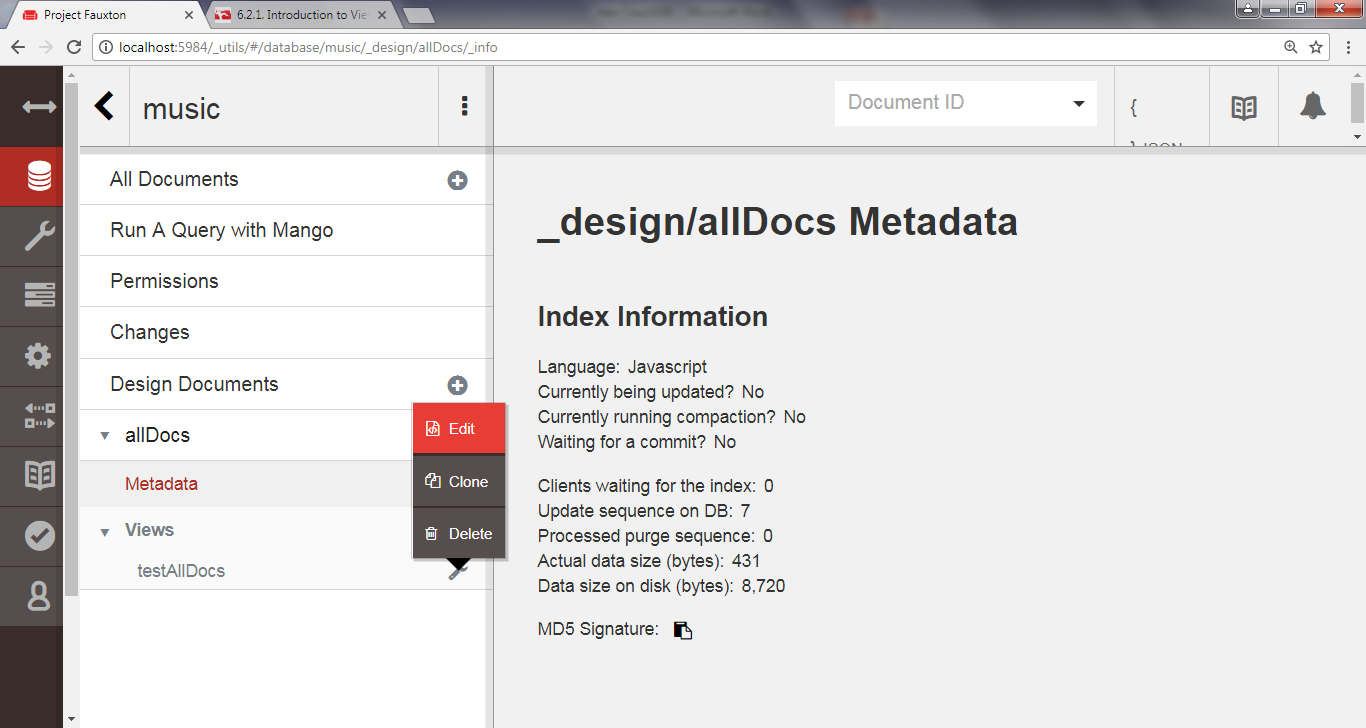
As we see in the output table, the key is indeed null, and the value is the same object we saw when we requested it directly from cURL.

To make a mapper that achieves the exact same thing as \_all\_docs, we need to emit something a little different.

Recall that \_all\_docs emits the document’s \_id field for the key and a simple object containing only the \_rev field for the value.

With that in mind, change the Map Function code to the following, and then save and view again.

To do this right click on AllDocs and select edit

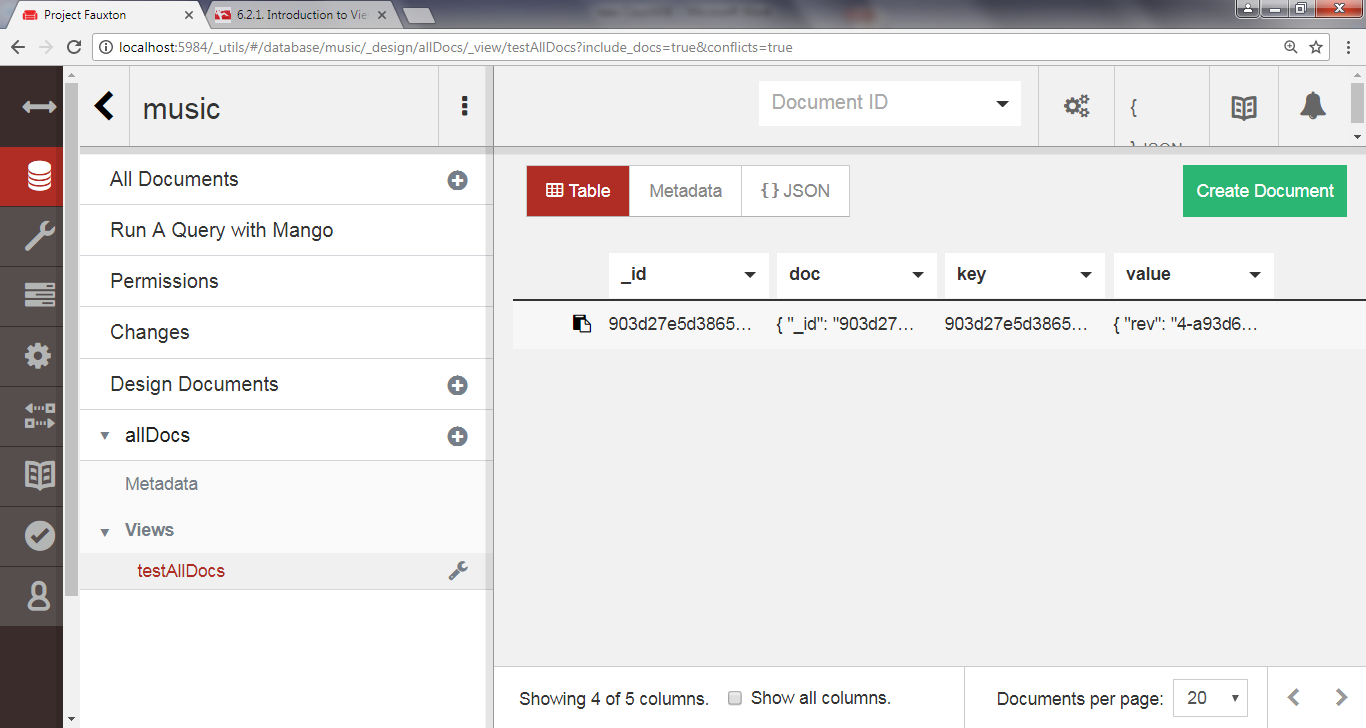


function(doc) {  
 emit(doc.\_id, { rev: doc.\_rev });

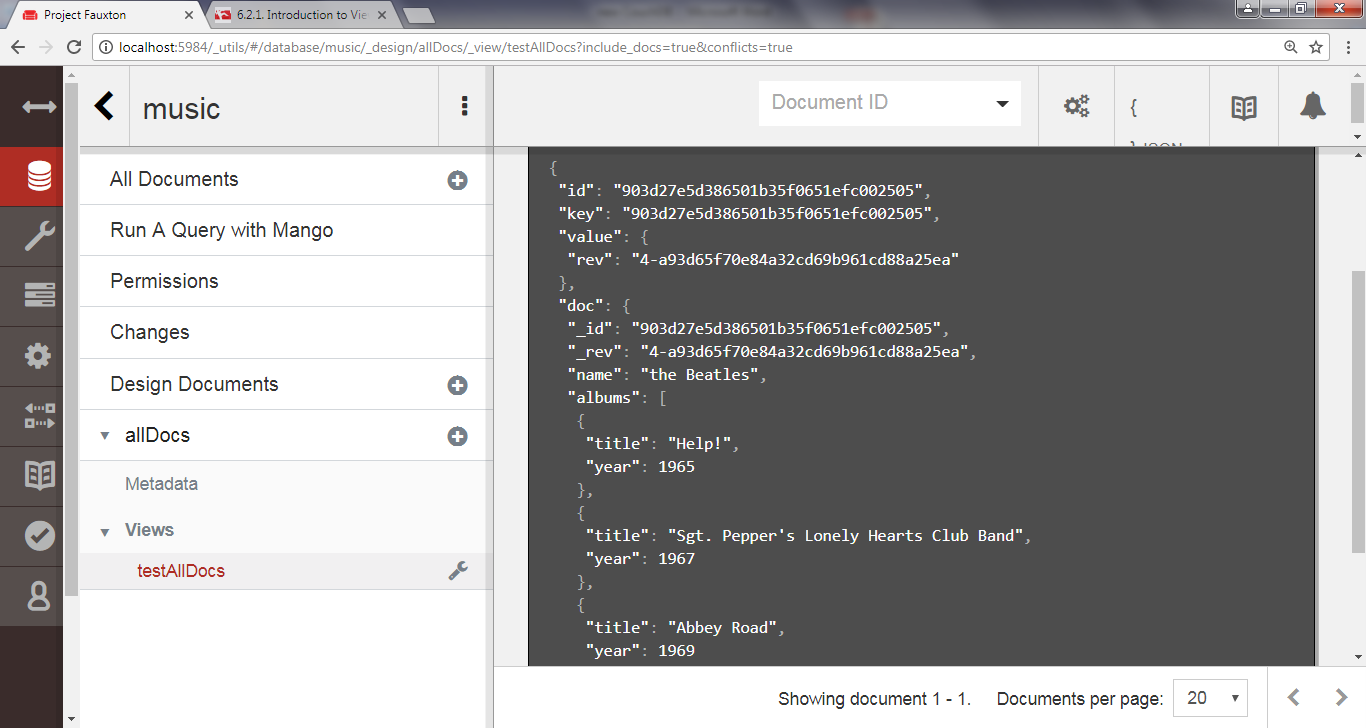
}

The output table should now resemble the following table, echoing the same key-value pair we saw earlier when enumerating records via \_all\_docs:

(Note it also appends the complete document (so its equivalent to the second curl example we ran with the include\_docs=true parameter)



And



Note that you don’t have to use Fauxton to execute temporary views.

**Saving a View as a Design Document**

When CouchDB executes a view, it must execute the provided map function for *each and every document* in the database.

This is extremely resource-intensive, chewing up a lot of processing power, and it’s slow.

A design document is a real document in the database, just like the Beatles document we created earlier.

As such, it can show up in views and be replicated to other CouchDB servers in the usual fashion.

Design documents always have IDs that start with \_design/ and contain one or more views.

The view name distinguishes this view from others housed in the same design document.

Deciding which views belong in which design document is largely application-specific and subject to taste.

**Finding Artists by Name**

Now that we’ve covered the basics of view creation, let’s develop an application- specific view.

Recall that our music database stores artist information, including a name field that contains the band’s name.

Using the normal GET access pattern or the \_all\_docs view, we can access documents by their \_id values, but we’re more interested in looking up bands by name.

In other words, today we can look up the document with \_id equal to b5dbb285f0872881c76ea01707000e82, but how do we find the document with name equal to *The Beatles*?

For this, we need a view.

In Fauxton, head back to the View page, click on design documents + and select new view, enter the following Map Function code with designDocument name Artists and index byName, save and view the result

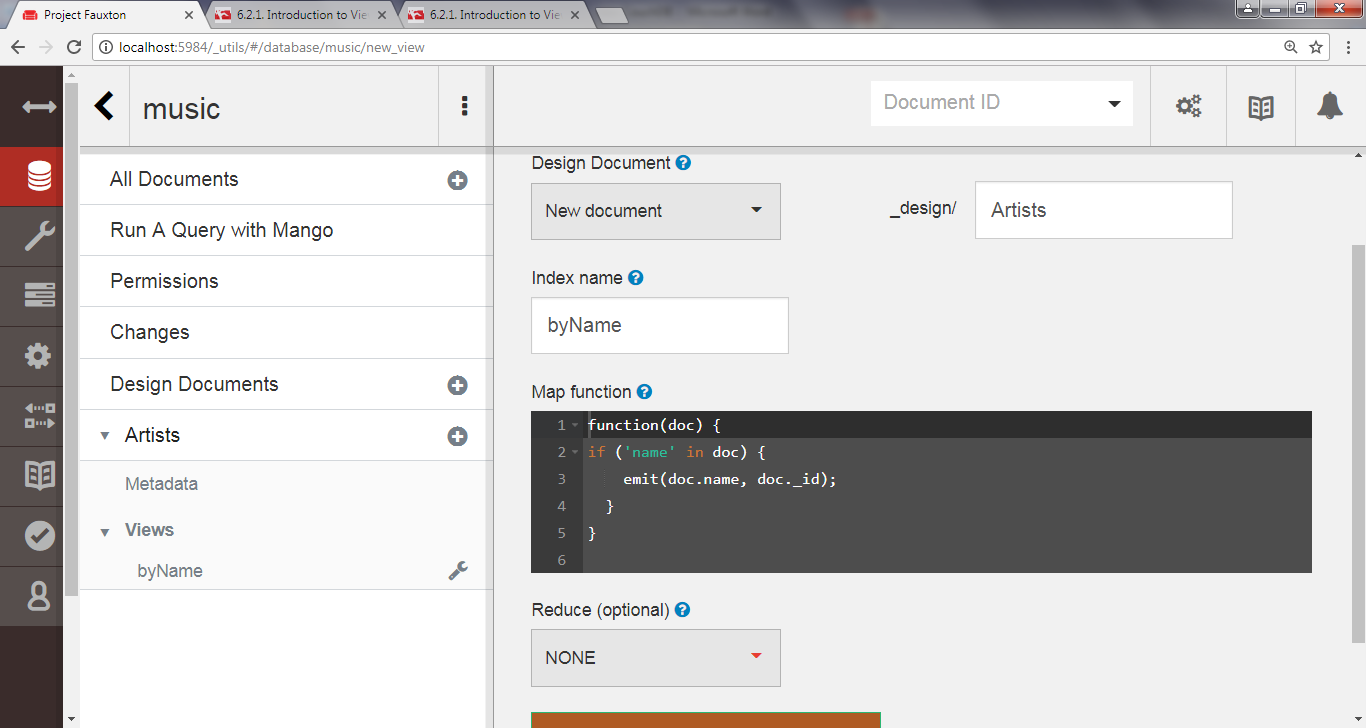
couchdb/artists\_by\_name\_mapper.js

**function**(doc) {

**if** (*'name'* **in** doc) {

emit(doc.name, doc.\_id);  
 }

}

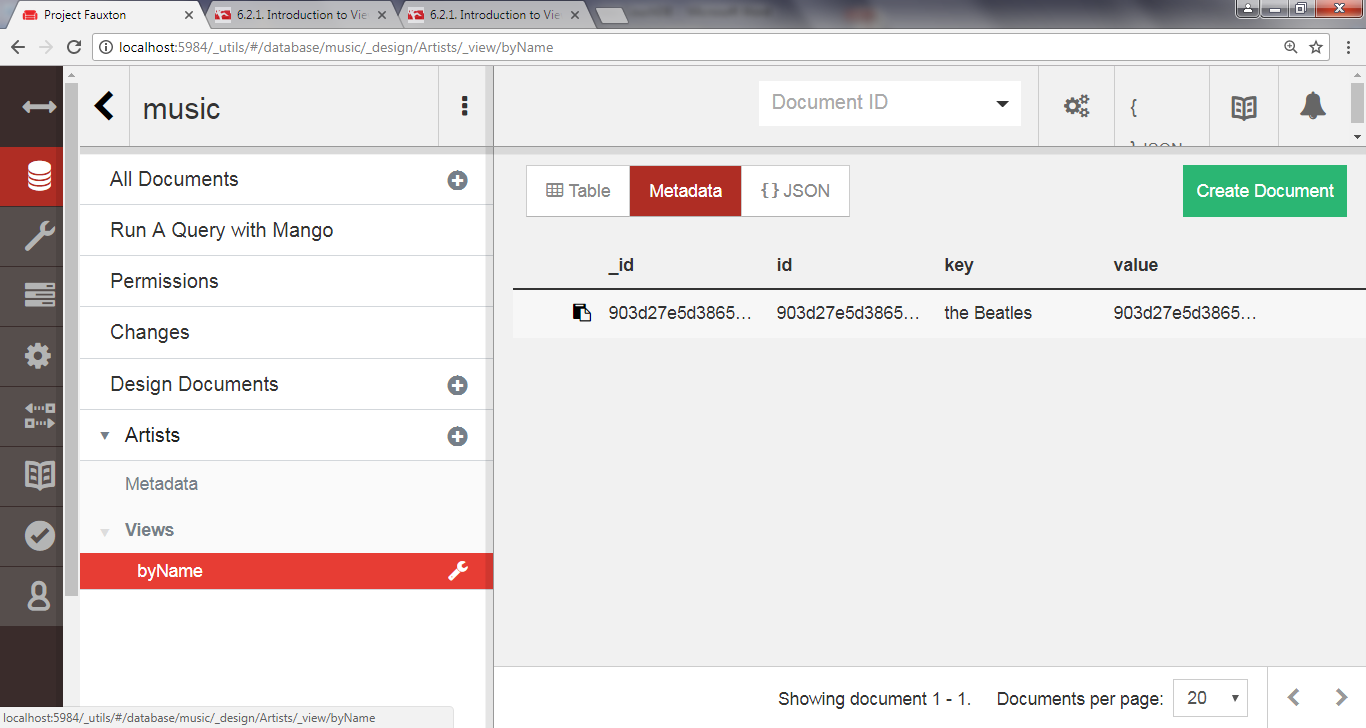


This function checks whether the current document has a name field and, if so, emits the name and document \_id as the relevant key-value pair.

This should produce a table like this:

Key. Value

The Beatles" "74c7a8d2a8548c8b97da748f43000ac4"



**Finding Albums by Name**

This time, let’s make a view that lets us find albums.

This will be the first example where the map function will emit more than one result per document.

Again return to the View page; click on the new view under Artists then enter the following mapper with name byAlbumName

couchdb/albums\_by\_name\_mapper.js

**function**(doc) {

**if** (*'name'* **in** doc && *'albums'* **in** doc) {

doc.albums.forEach(**function**(album){

**var** key = album.title || album.name || album, value = { by: doc.name, album: album };  
 emit(key, value);

}); }

}

 This function checks whether the current document has a name field and an albums field.

If so, it emits a key-value pair for each album where the key is the album’s title or name and the value is a compound object containing the artist’s name and the original album object.

It produces a table like this:

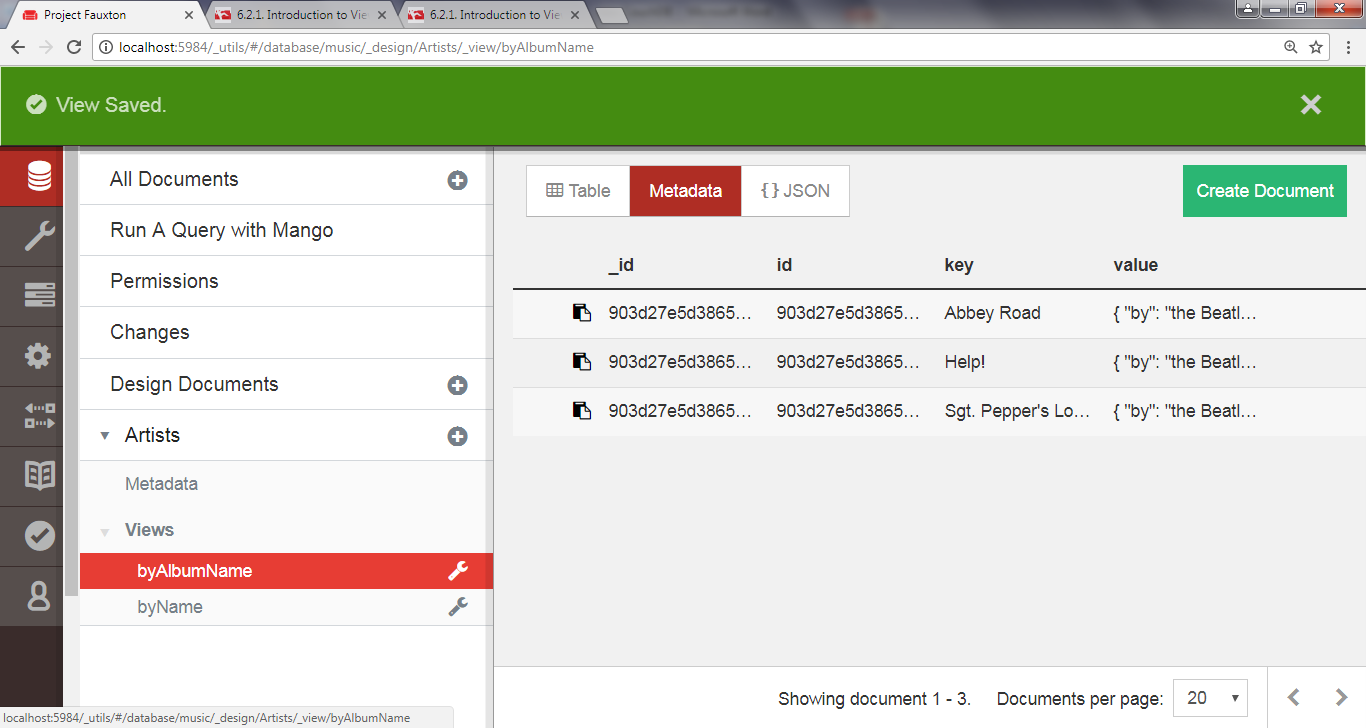
Key. Value

"Abbey Road" {by: "The Beatles", album: {title: "Abbey Road", year: 1969}}

Help!" {by: "The Beatles", album: {title: "Help!", year: 1965}}

"Sgt. Pepper's Lonely Hearts Club Band"

{by: "The Beatles", album: {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}}



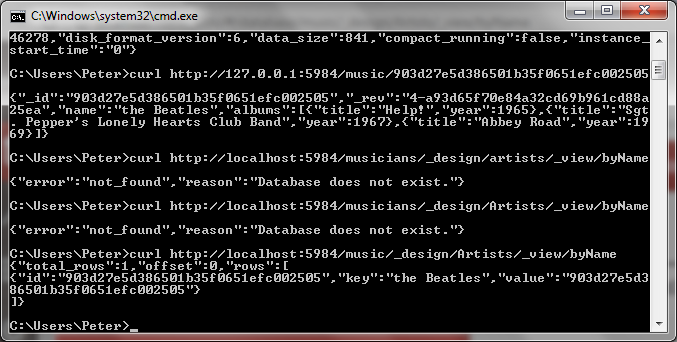
**Querying Our Custom Artist and Album Views**

Now that we have a couple of custom design documents saved, let’s jump back to the command line and query them with the curl command.

We’ll start with the Artists By Name view.

On the command line, execute the following:

**curl** [**http://localhost:5984/music/\_design/Artists/\_view/byName**](http://localhost:5984/music/_design/Artists/_view/byName)



To query a view, construct the path /<database\_name>/\_design/<design\_doc>/\_view/ <view\_name>, replacing the parts as appropriate.

In our case, we’re querying the byName view in the Artists design document of the music database.

No surprise here that the output includes our two document, keyed by the band name.

Next, let’s try to find Albums By Name:

**curl** [**http://localhost:5984/music/\_design/Artists/\_view/byAlbumName**](http://localhost:5984/music/_design/Artists/_view/byAlbumName)

{

"total\_rows":3,  
 "offset":0,  
 "rows":[{

"id":"74c7a8d2a8548c8b97da748f43000ac4",  
 "key":"Abbey Road",  
 "value":{

"by":"The Beatles",  
 "album":{

"title":"Abbey Road",

"year":1969  
 }

} },{

"id":"74c7a8d2a8548c8b97da748f43000ac4",  
 "key":"Help!",  
 "value":{

"by":"The Beatles",  
 "album":{

"title":"Help!",

"year":1965  
 }

} },{

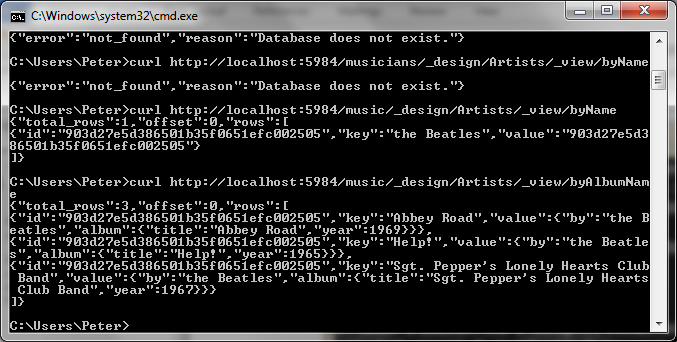
"id":"74c7a8d2a8548c8b97da748f43000ac4",  
 "key":"Sgt. Pepper's Lonely Hearts Club Band",  
 "value":{

"by":"The Beatles",  
 "album":{

"title":"Sgt. Pepper's Lonely Hearts Club Band",  
 "year":1967

} }

}] }



CouchDB will ensure that the records are presented in alphanumerical order by the emitted keys.

In effect, this is the indexing that CouchDB offers.

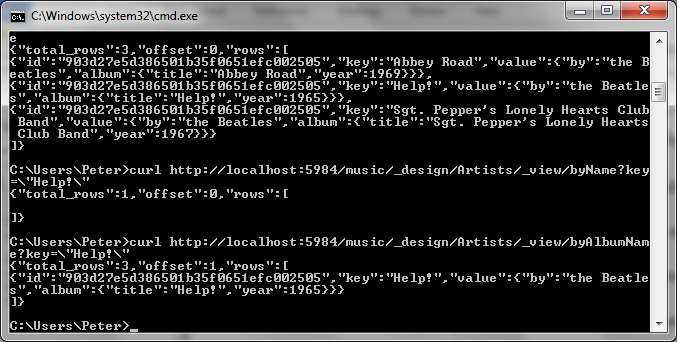
When designing your views, it’s important to pick emitted keys that will make sense when ordered.

Requesting a view in this fashion returns the whole set, but what if we want just a subset?

One way to do that is to use the key URL parameter.

When you specify a key, only rows with that exact key are returned.

**curl http://localhost:5984/music/\_design/Artists/\_view/byAlbumName?key=\"Help!\"**



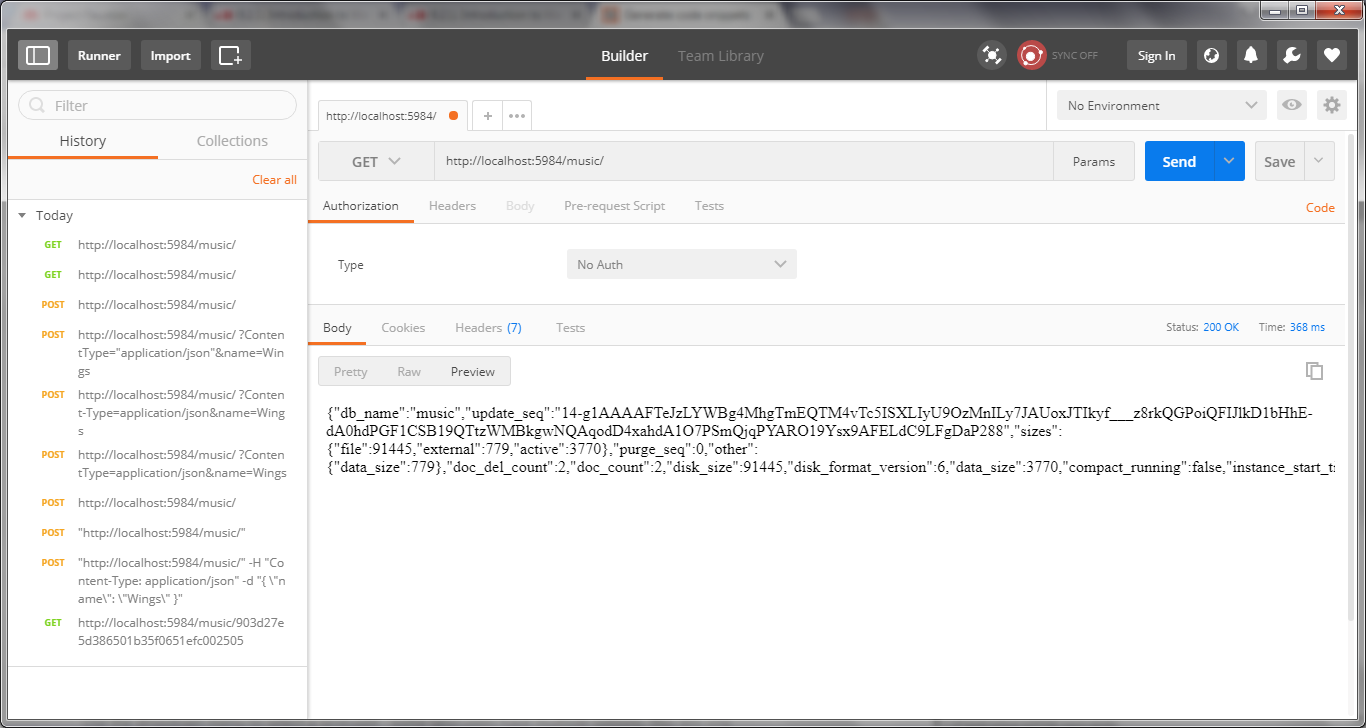
Notice the total\_rows and offset fields in the response.

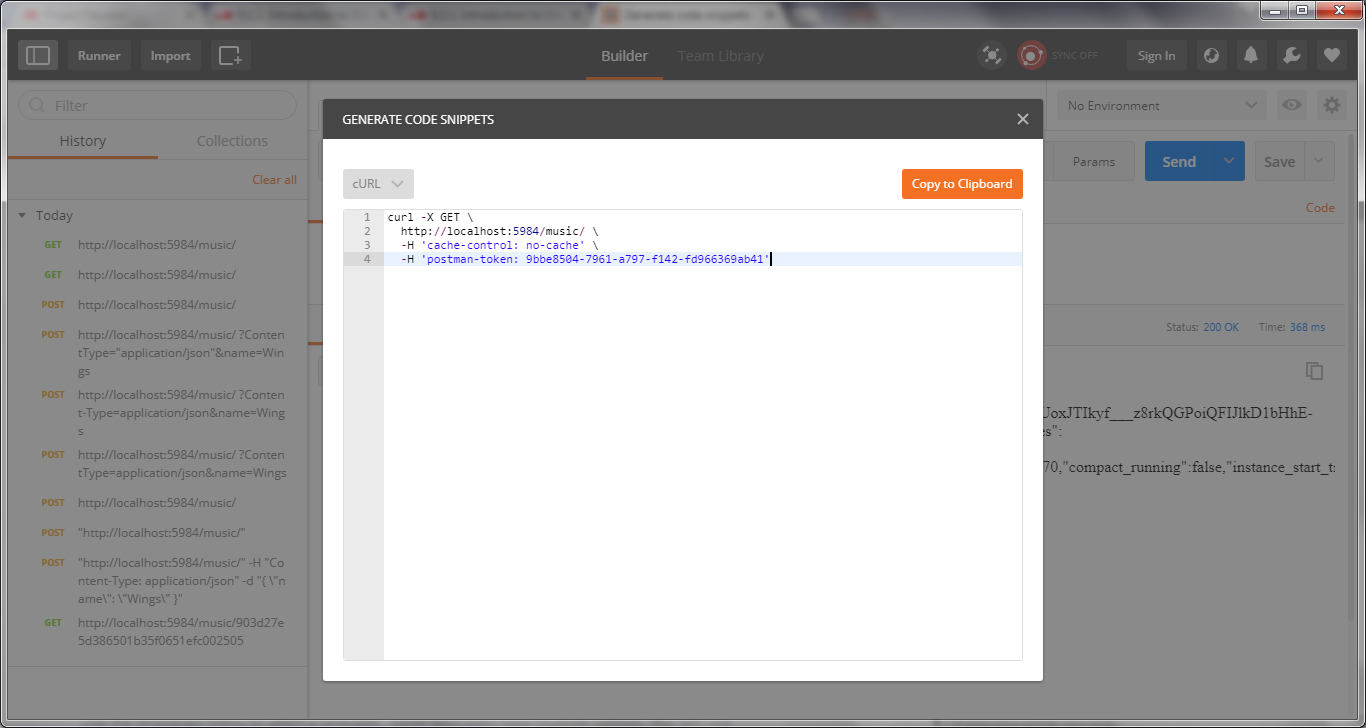
The total\_rows field counts the total number of records in the view, not just the subset returned for this request.

The offset field tells us how far into that full set the first record presented appears.

 Note you can use Postman (chrome) to generate curl code

Put the GET call etc in postman and select code and select the language (here curl)





You can then copyt and post the code to the environment you want (java, curl etc)

**Creating Advanced Views with Reducers**

Mapreduce-based views provide the means by which we can harness CouchDB’s indexing and aggregation facilities.

Previously all our views consisted of only mappers.

Now we’re going to add reducers to the mix, developing new capabilities against the database

First, return to the View page (for Artists), set the index name to noAlbums and then enter the following map function:

function(doc) {

(doc.albums || []).forEach(function(album){

emit(doc.name, 1);

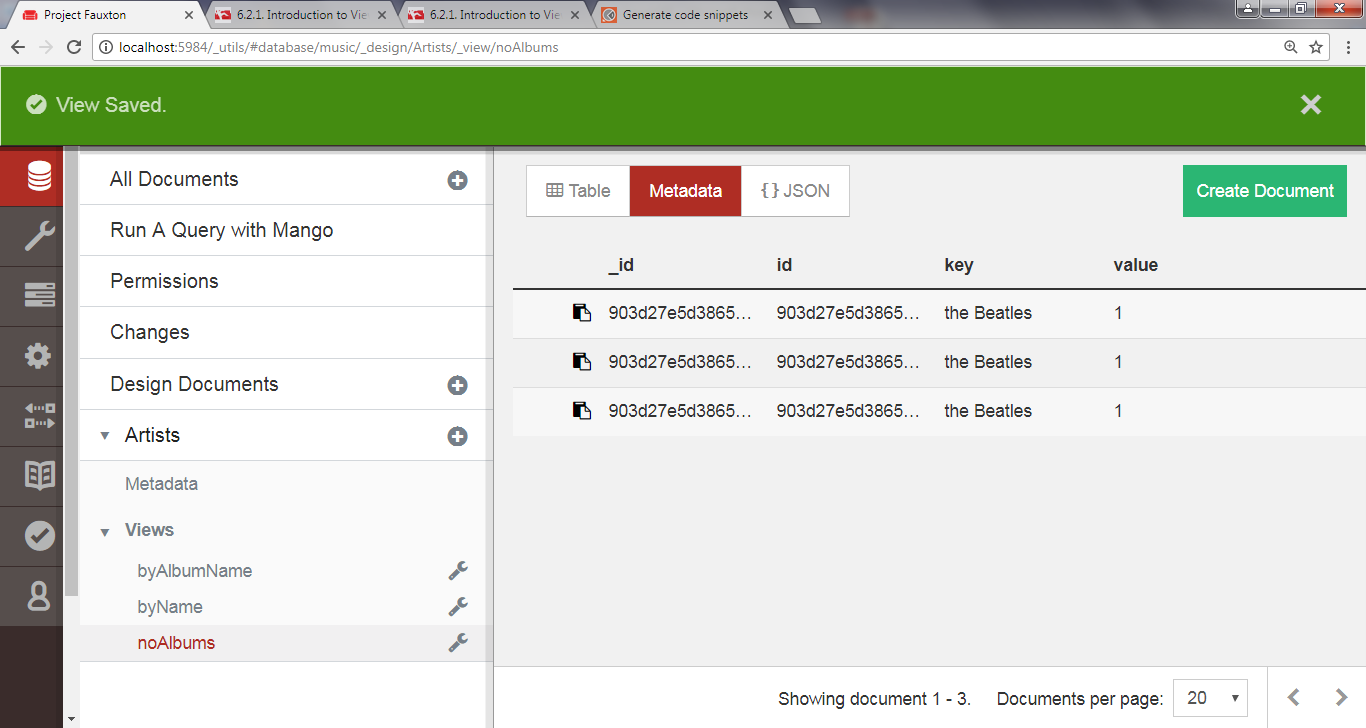
});

}

This function digs into the artist document and then down into each album

For each album, it emits a key-value pair consisting of the band name and an ID (the key) and the number 1.

Save it and view the result



With the map function in place, go back and edit the view and enter the following under Reduce Function ( select CUSTOM)

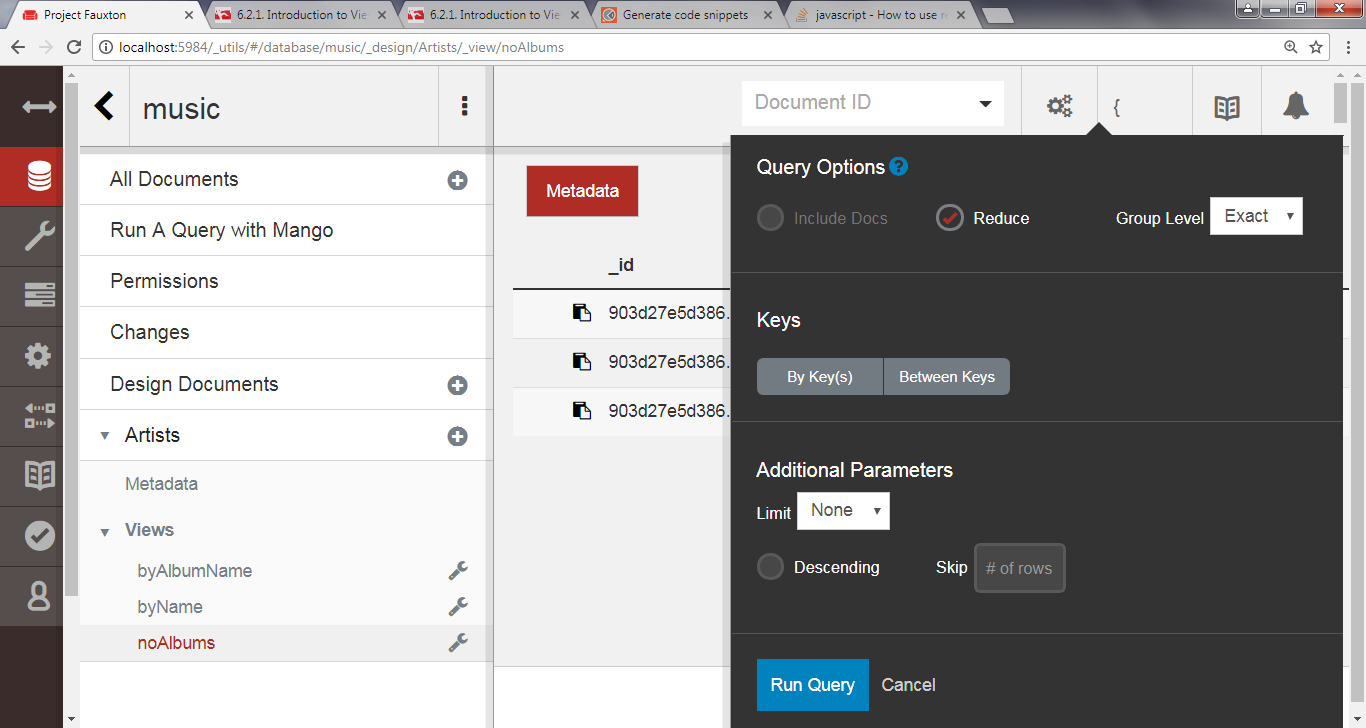
**function**(key, values, rereduce) { **return** sum(values);

}

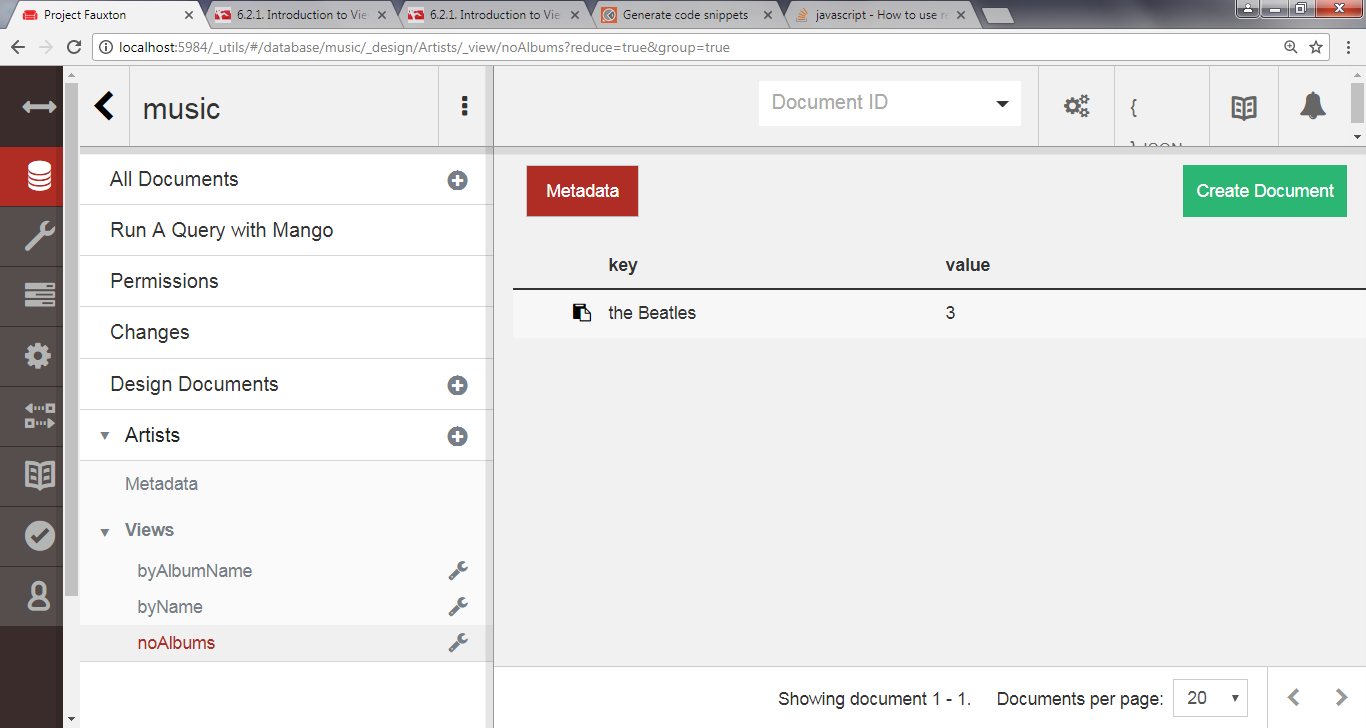
This code merely sums the numbers in the values list—which we’ll talk about momentarily once we’ve run the view.

Finally, view the results.

You will need to select settings and click reduce to on (by the way if you don’t want to see the documents in the results click this to off)



The output should resemble the following table:



Key Value

|  |  |
| --- | --- |
| "The Beatless" | 3 |

What happened? In short, the reducer *reduced* the output by combining like mapper rows in accordance with our Reducer Function.

The CouchDB mapreduce engine works conceptually like the other mapreducers (Riak’s and MongoDB)

Specifically, here’s a high-level outline of the steps CouchDB takes to build a view:

1. Send documents off to the mapper function.
2. Collect all the emitted values.
3. Sort emitted rows by their keys.
4. Send chunks of rows with the same keys to the reduce function.
5. If there was too much data to handle all reductions in a single call, call the reduce function again but with previously reduced values.
6. Repeat recursive calls to the reduce function as necessary until no duplicate keys remain.

Reduce functions in CouchDB take three arguments: key, values, and rereduce.

The first argument, key, is an array of tuples—two element arrays containing the key emitted by the mapper and the \_id of the document that produced it.

The second argument, values, is an array of values corresponding to the keys.

The third argument, rereduce, is a boolean value that will be true if this invocation is a *rereduction*.

That is, rather than being sent keys and values that were emitted from mapper calls, this call is sent the products of previous reducer calls.

In this case, the key parameter will be a tuple of the mapper key and the document's name attribute

Stepping Through Reducer Calls

Let’s work through an example based on the output we just saw.

Consider documents (artists) with a number of albums

The mappers run on the documents and emit key-value pairs of the form “the Beatles”/1.

At some point, enough of these have been emitted that CouchDB invokes a reducer.

That call might look like this:

reduce(

[[*"the Beatles"*, id1], [*"the Beatles"*, id2], ...], [1, 1, ...],

false

)

*// keys are the same  
// values are all 1  
// rereduce is false*

Recall that in our reducer function we take the sum() of values.

Since they’re all 1, the sum will simply be the length—effectively a count of how many albums the artist has

CouchDB keeps this return value for later processing.

For the sake of this example, let’s call that number 10.

Some time later, after CouchDB has run these kinds of calls several times, it decides to combine the intermediate reducer results by executing a rereduce:

reduce(  
 null, // key array is null

[10, 10, 8],// values are outputs from previous reducer calls

true // rereduce is true

)

Our reducer function again takes the sum() of values.

This time, the values add up to 28.

Rereduce calls may be recursive.

They go on as long as there is reduction to be done, until all the intermediate values have been combined into one.

Most mapreduce systems (like Riak and MongoDB), throw away the output of mappers and reducers after the work is done.

In those systems, mapreduce is seen as a means to an end—something to be executed whenever the need arises, each time starting from scratch.

Not so with CouchDB.

Once a view is codified into a design document, CouchDB will keep the intermediate mapper and reducer values until a change to a document would invalidate the data.

At that time, CouchDB will incrementally run mappers and reducers to correct for the updated data. It won’t start from scratch, recalculating everything each time.

This is the genius of CouchDB views.

CouchDB is able to use mapreduce as its primary indexing mechanism by not tossing away intermediate data values.

**Replicating Data in CouchDB**

CouchDB is all about asynchronous environments and data durability (<http://en.wikipedia.org/wiki/Durability_(database_systems)>)

According to CouchDB, the safest place to store your data is everywhere, and it gives you the tools to do it.

Some other databases maintain a single master node to guarantee consistency.

Still others ensure it with a quorum of agreeing nodes.

CouchDB does neither of these; instead, it supports something called multi-master or master-master replication.

Each CouchDB server is equally able to receive updates, respond to requests, and delete data, regardless of whether it’s able to connect to any other server.

In this model, changes are selectively replicated in one direction, and all data is subject to replication in the same way.

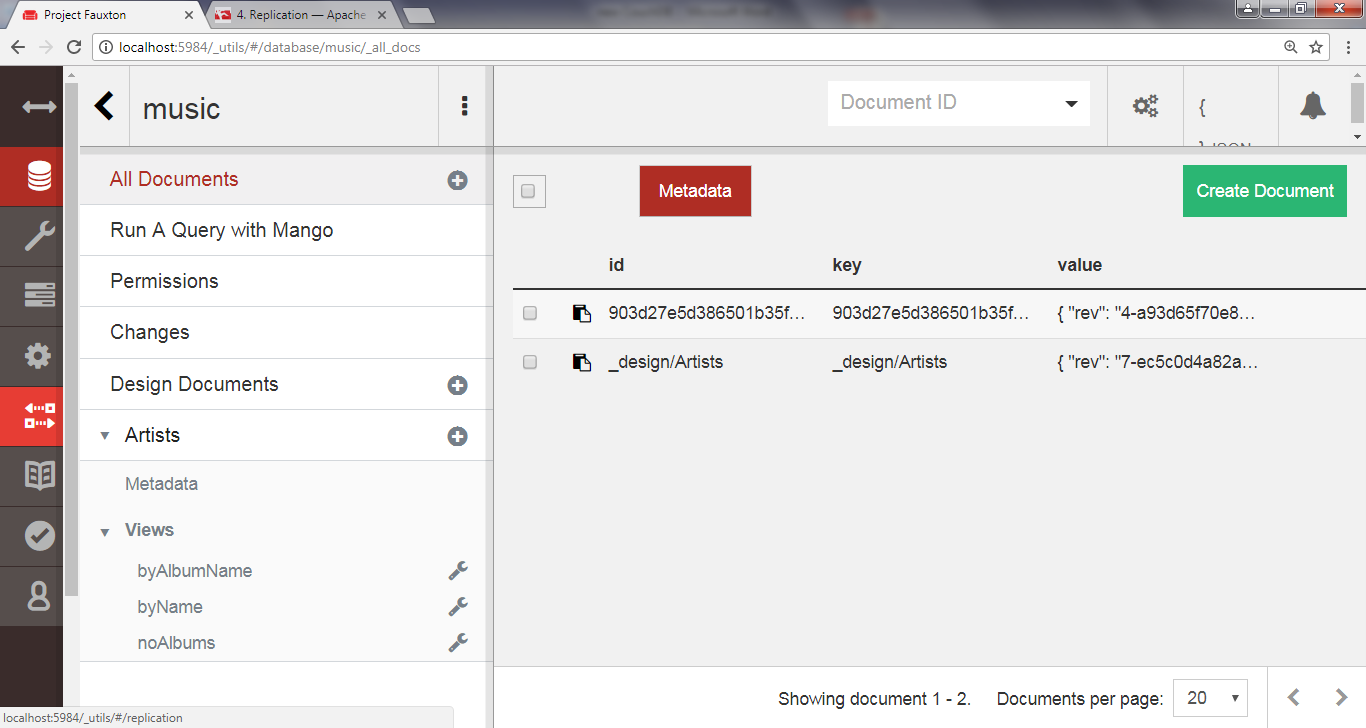
In other words, there is no sharding (<http://en.wikipedia.org/wiki/Shard_(database_architecture)>)

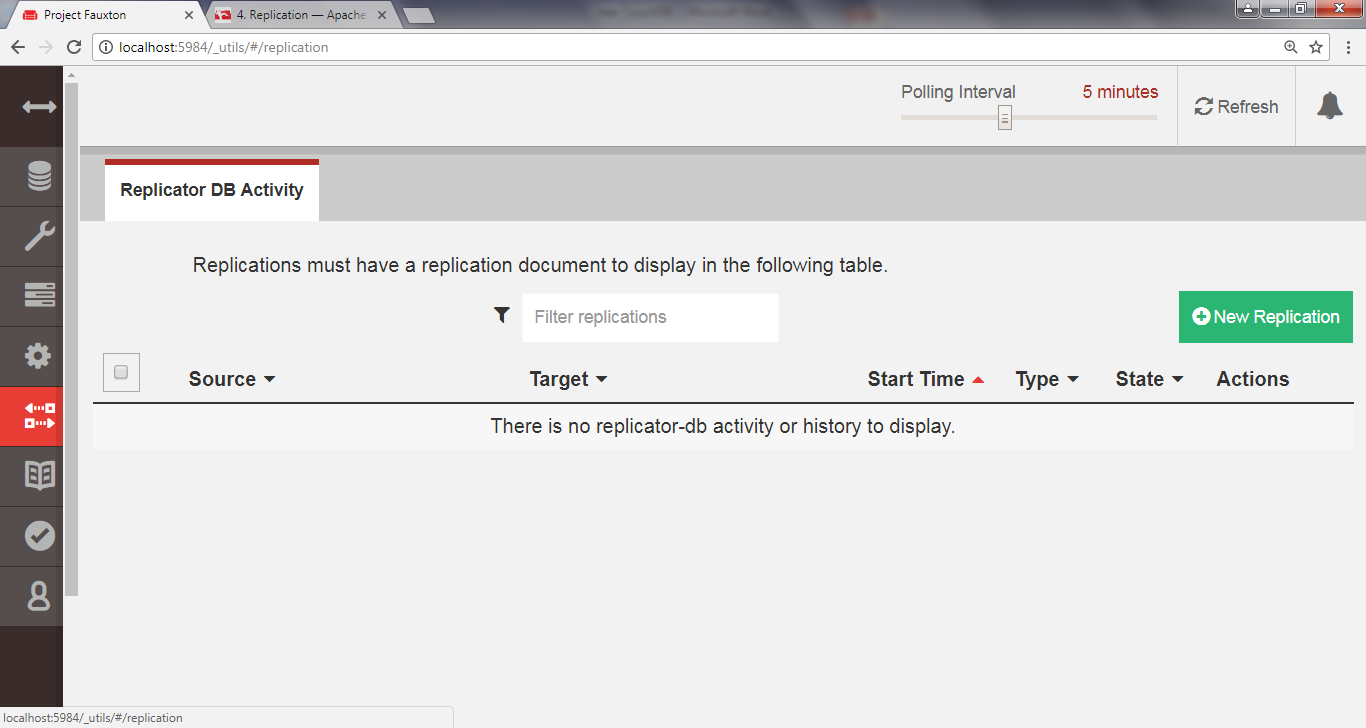
Servers participating in replication will all have all of the data.

First we’ll see how to set up ad hoc and continuous replication between databases.

Then we’ll work through the implications of conflicting data and how to make applications capable of handling these cases gracefully.

To begin, click the Replicator link in the Tools menu on the left side of the page.

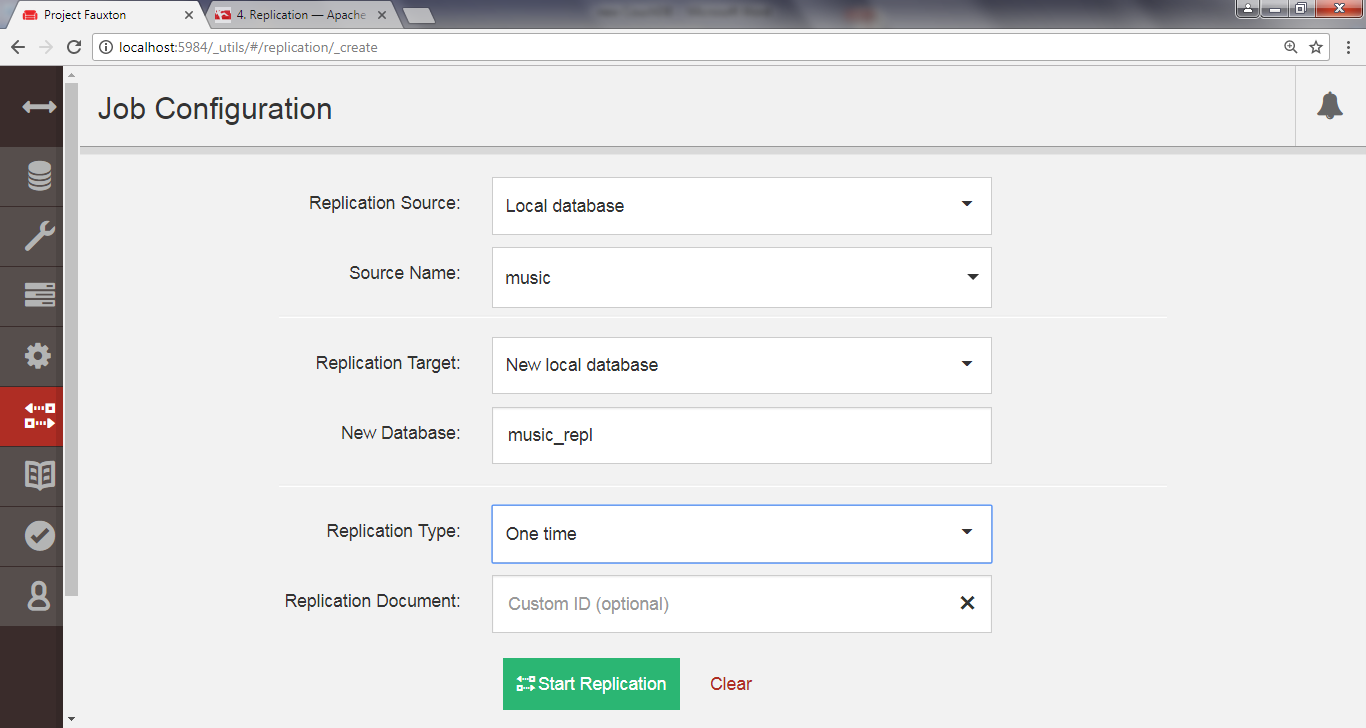




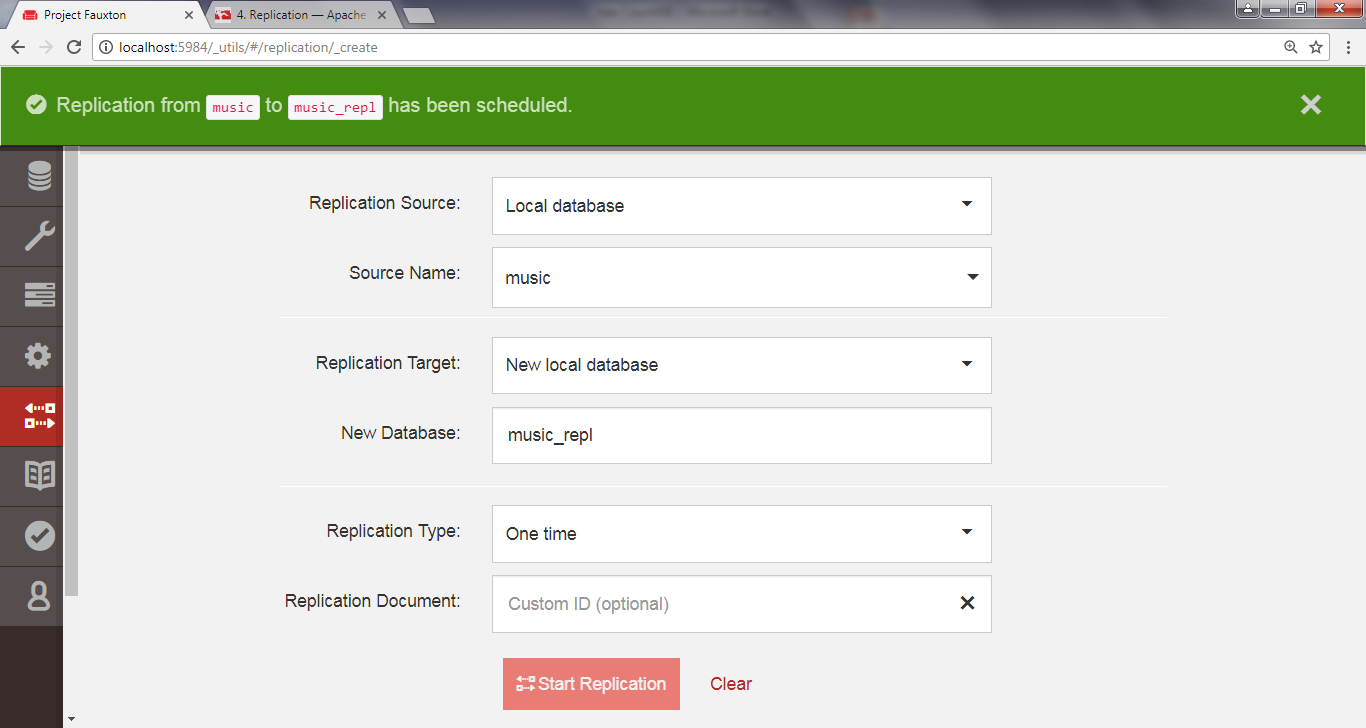
Click new Replication

In the “Source”, choose local and select *music as the name* and enter new local database as the target and *music\_repl* as the name

Select one time as the type, and then click Start Replication

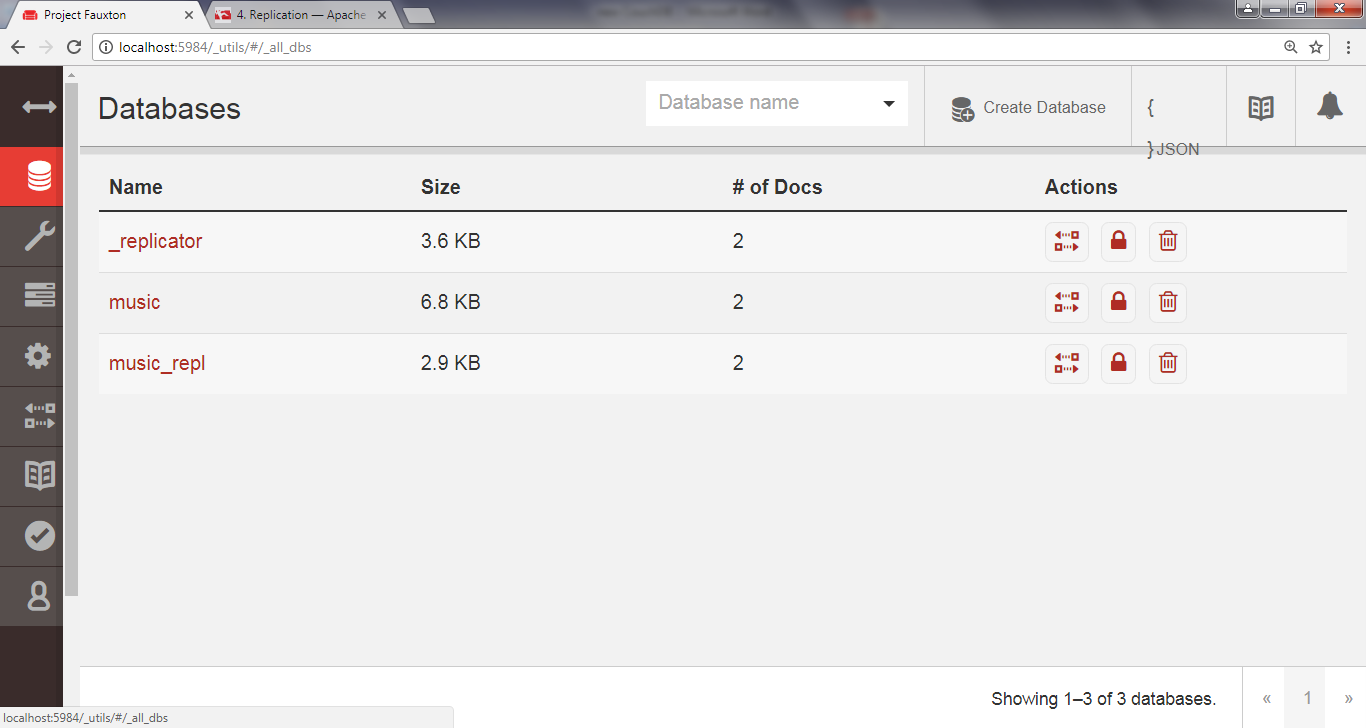


This should produce an event message in the event log below the form.

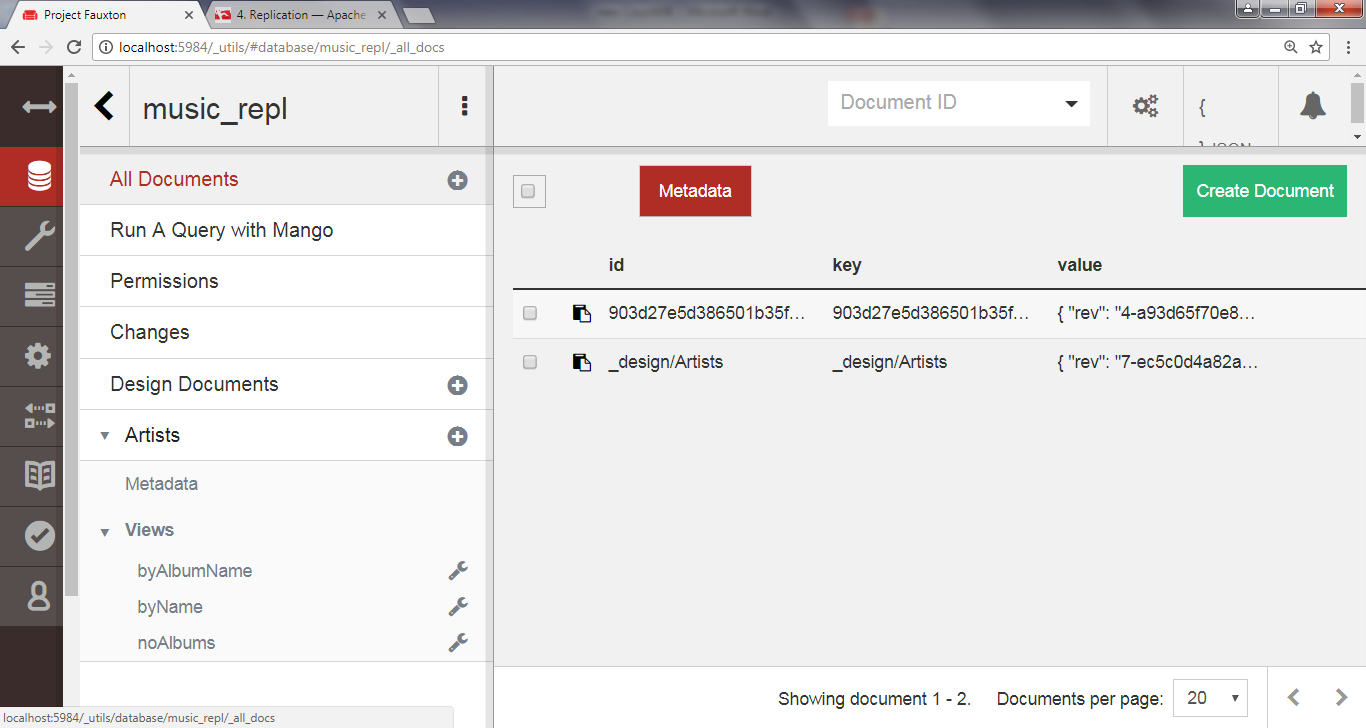


To confirm that the replication request worked, go back to the Fauxton Overview page.

There should now be a new database called music\_repl with the same number of documents as the music database.



If it has fewer, give it some time and refresh the page—CouchDB may be in the process of catching up.



Don’t be concerned if the Update Seq values don’t match.

That’s because the original music database had deletions and updates to documents, whereas the music- repl database had only insertions to bring it up to speed.

Creating Conflicts

Next we’ll create a conflict and then explore how to deal with it.

Keep the Replicator page handy because we’re going to be triggering ad hoc replication between music and music\_repl frequently.

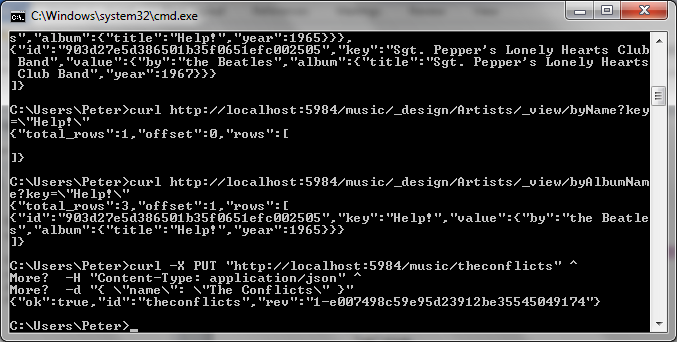
Drop back to the command line, and enter this to create a document in the music database:

(Remember to use ^ and escape " if you are using dos prompt)

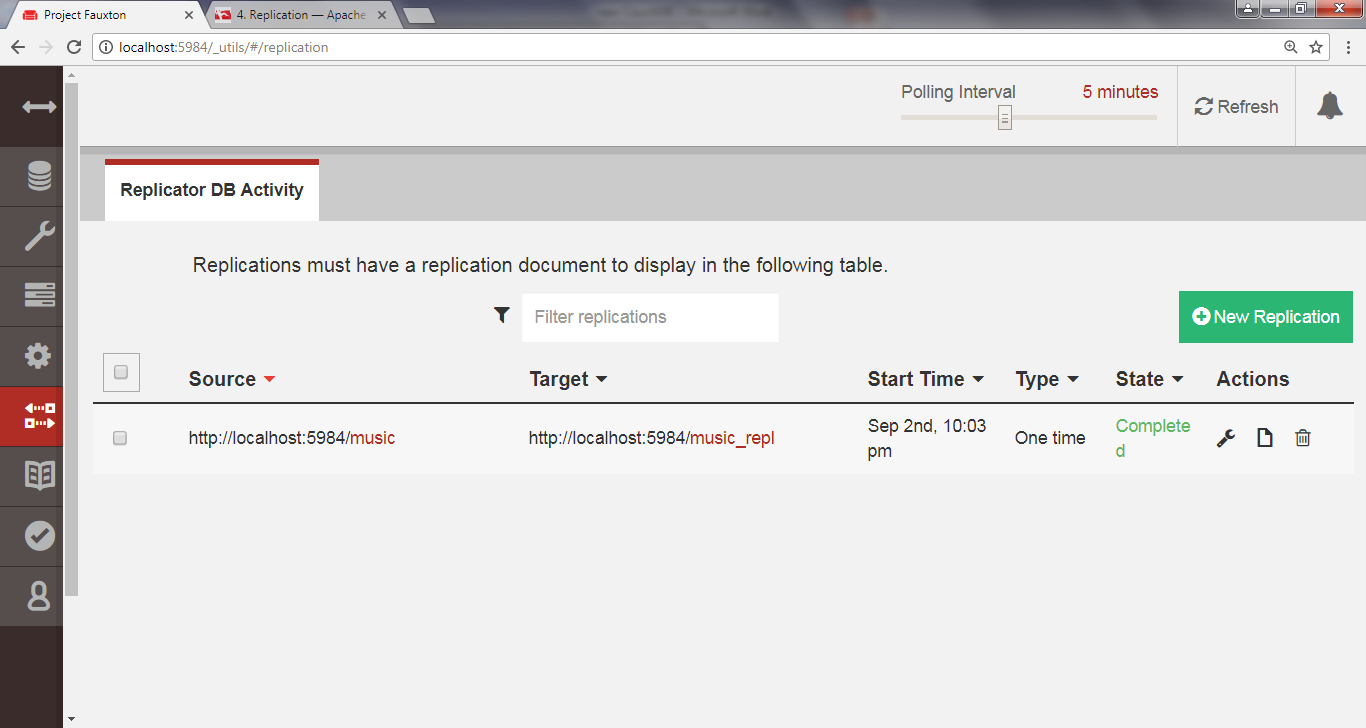
**curl -X PUT "**[**http://localhost:5984/music/theconflicts**](http://localhost:5984/music/theconflicts)**" ^  
 -H "Content-Type: application/json" ^  
 -d "{ \"name\": \"The Conflicts\" }"**

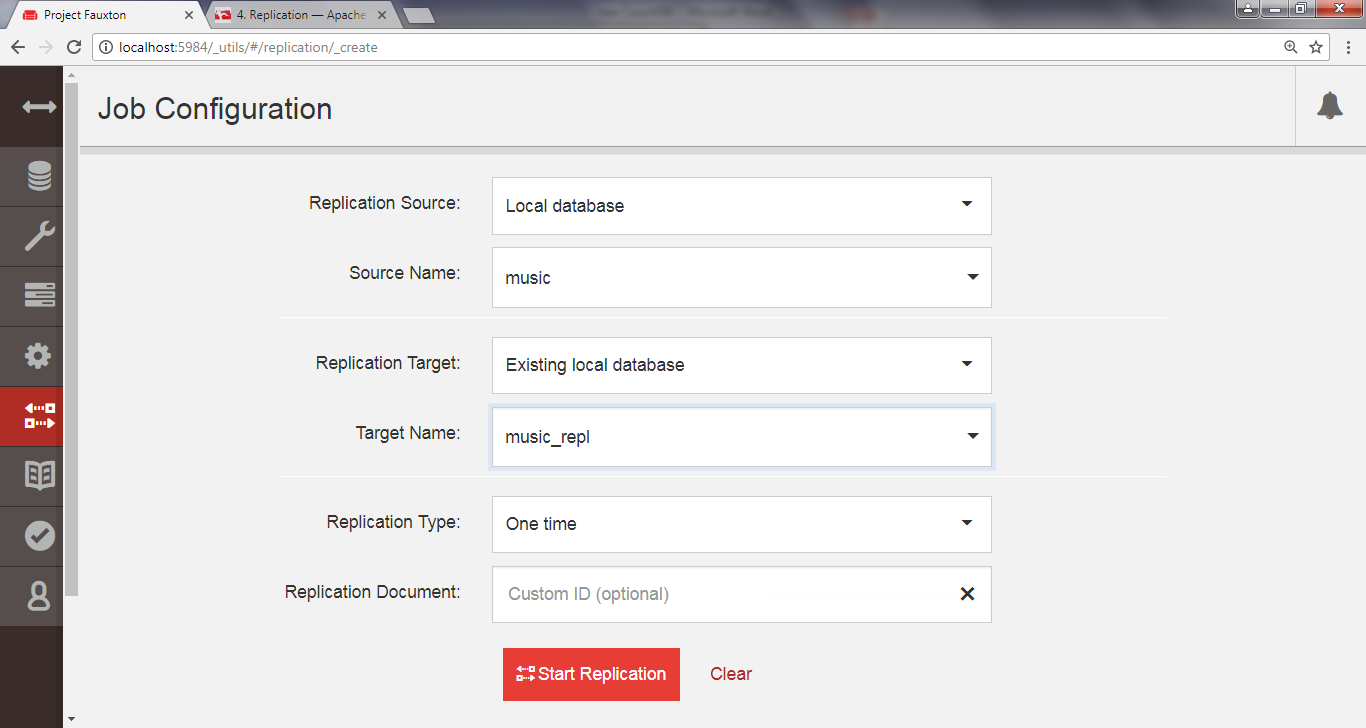
{  
 "ok":true,  
 "id":"theconflicts",  
 "rev":"1-e007498c59e95d23912be35545049174"

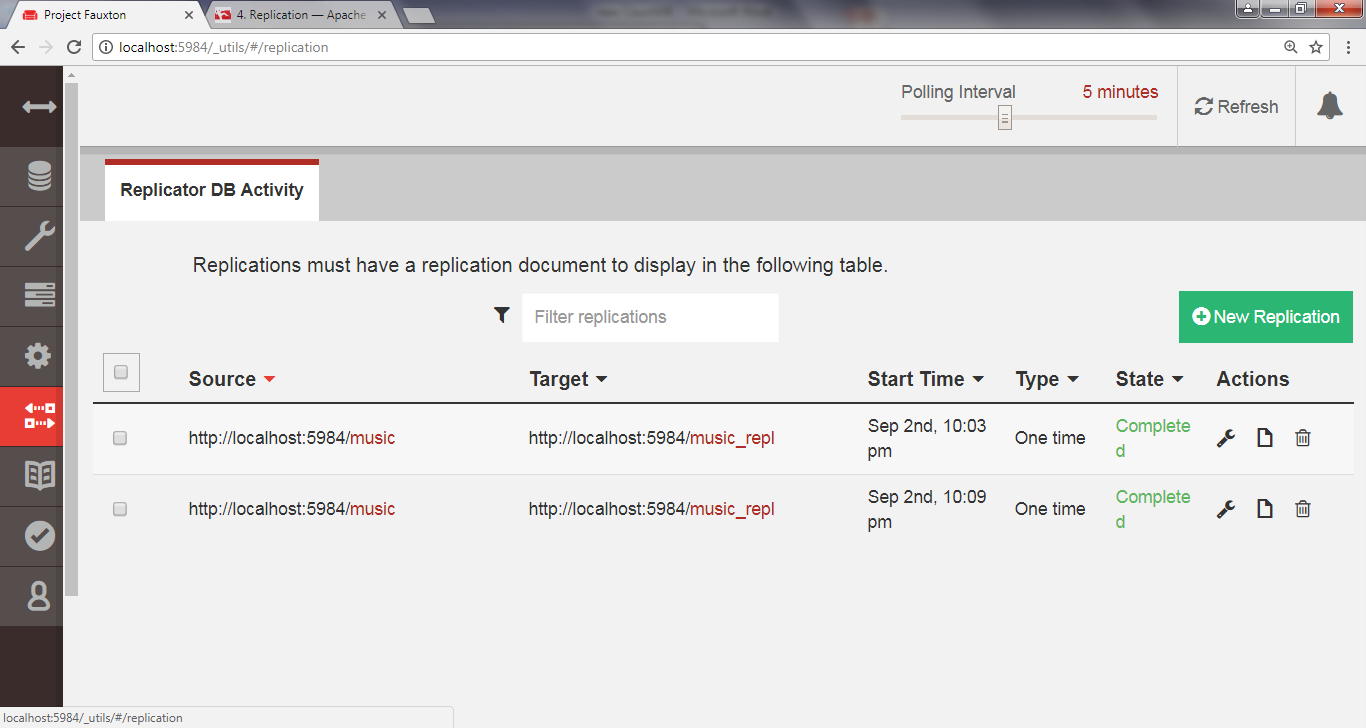
}



On the Replicator page, click Replicate to trigger another synchronization.







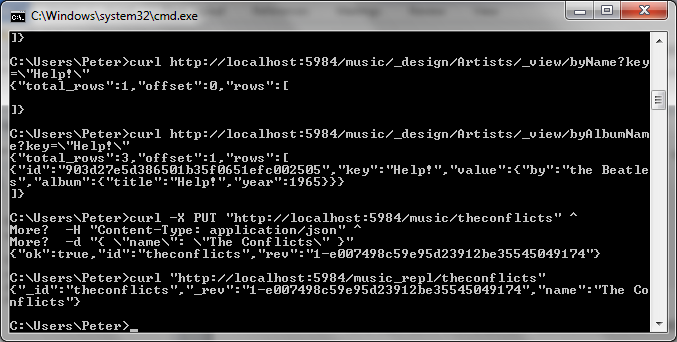
We can confirm that the document was successfully replicated by retrieving it from the music\_repl database.

**curl "**[**http://localhost:5984/music\_repl/theconflicts**](http://localhost:5984/music_repl/theconflicts)**"**

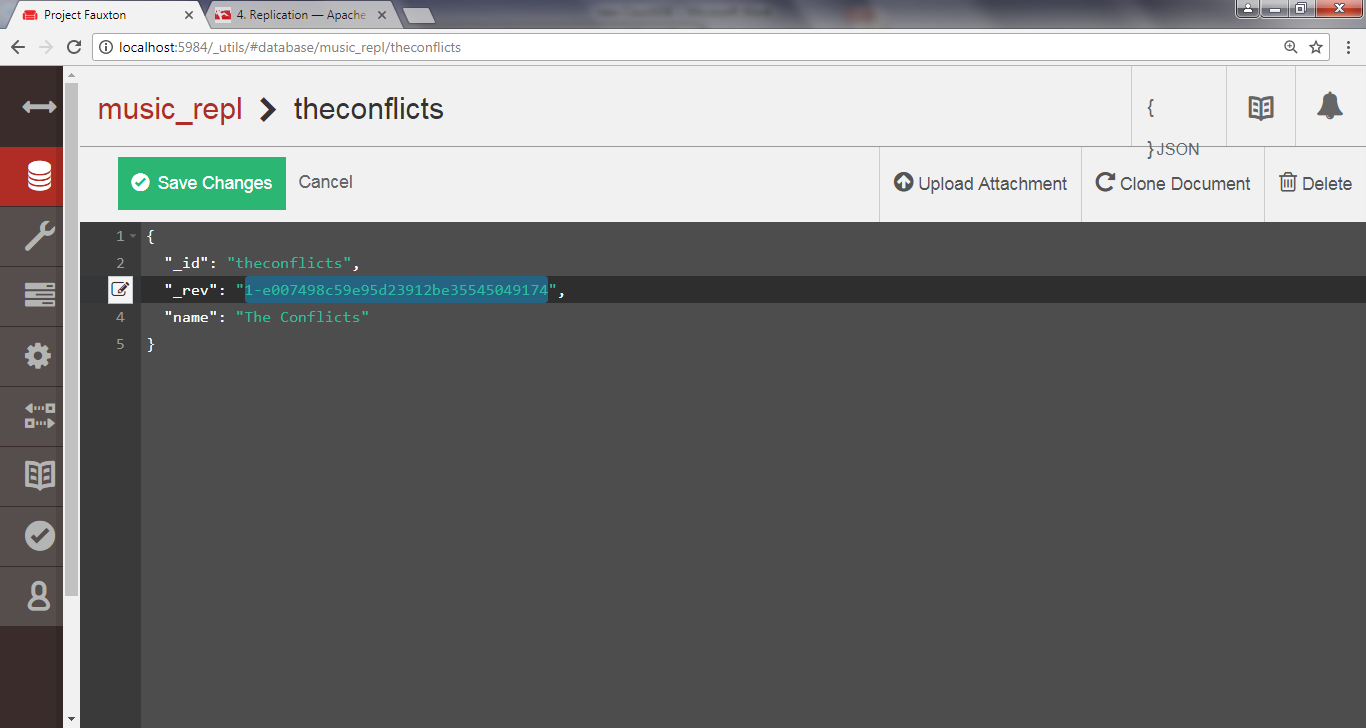
{

"\_id":"theconflicts",  
 "\_rev":"1-e007498c59e95d23912be35545049174",  
 "name":"The Conflicts"

}



Next, let’s update it in music\_repl by adding an album called *Conflicts of Interest*.

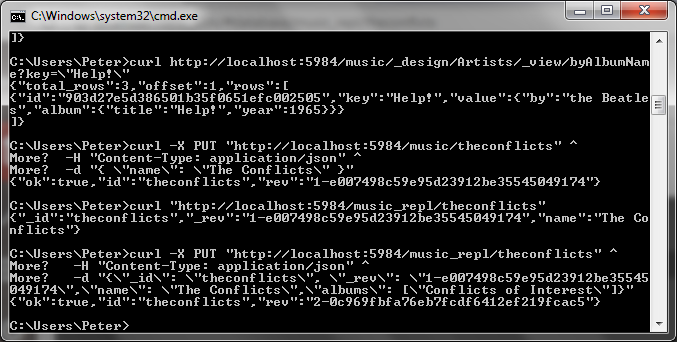


**curl -X PUT "**[**http://localhost:5984/music\_repl/theconflicts**](http://localhost:5984/music_repl/theconflicts)**" ^  
 -H "Content-Type: application/json" ^  
 -d** "**{\**"\_id\": \"theconflicts\", \"\_rev\": \"1-e007498c59e95d23912be35545049174\",\"name\": \"The Conflicts\",\"albums\": [\"Conflicts of Interest\"]}"

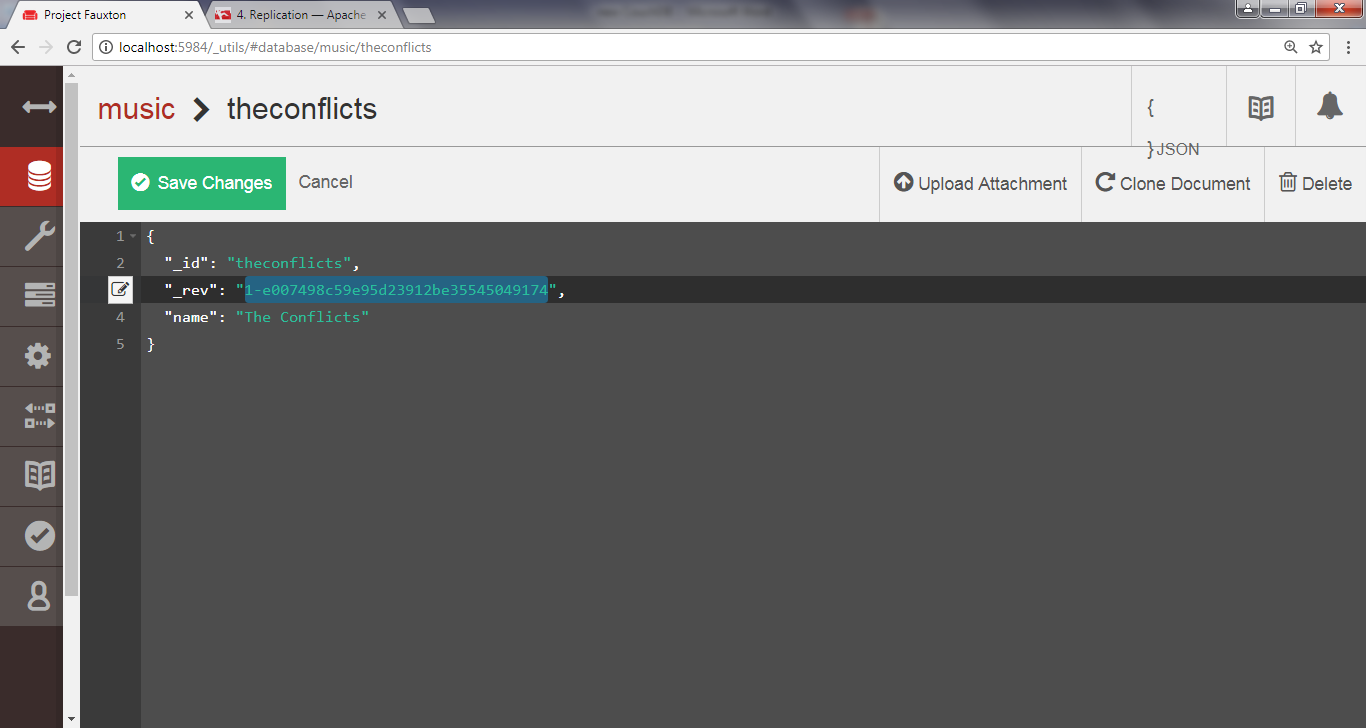
{

"ok":true,  
 "id":"theconflicts",  
 "rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5"

}



And create a conflicting update in music proper by adding a different album: *Conflicting Opinions*.



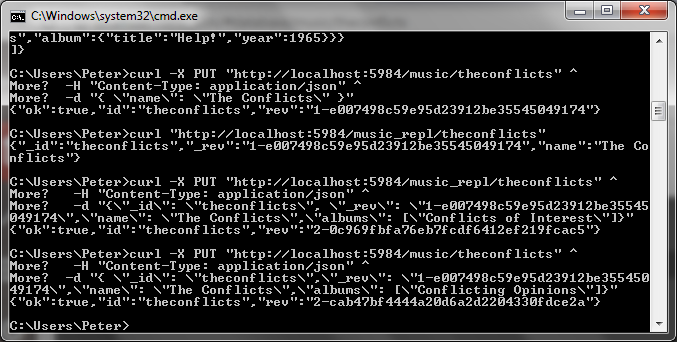
**curl -X PUT "**[**http://localhost:5984/music/theconflicts**](http://localhost:5984/music/theconflicts)**" ^  
 -H "Content-Type: application/json" ^**

**-d** "**{** \"\_id\": \"theconflicts\",\"\_rev\": \"1-e007498c59e95d23912be35545049174\",\"name\": \"The Conflicts\",\"albums\": [\"Conflicting Opinions\"]}"

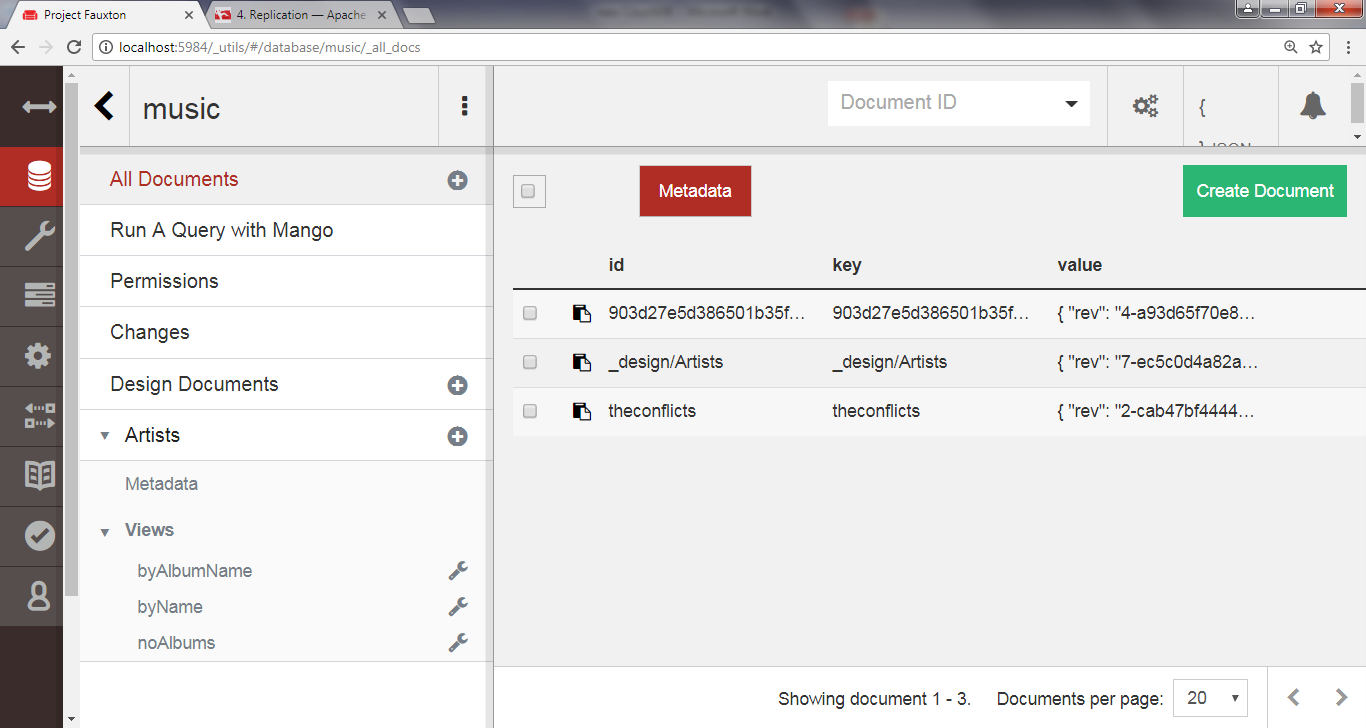
{

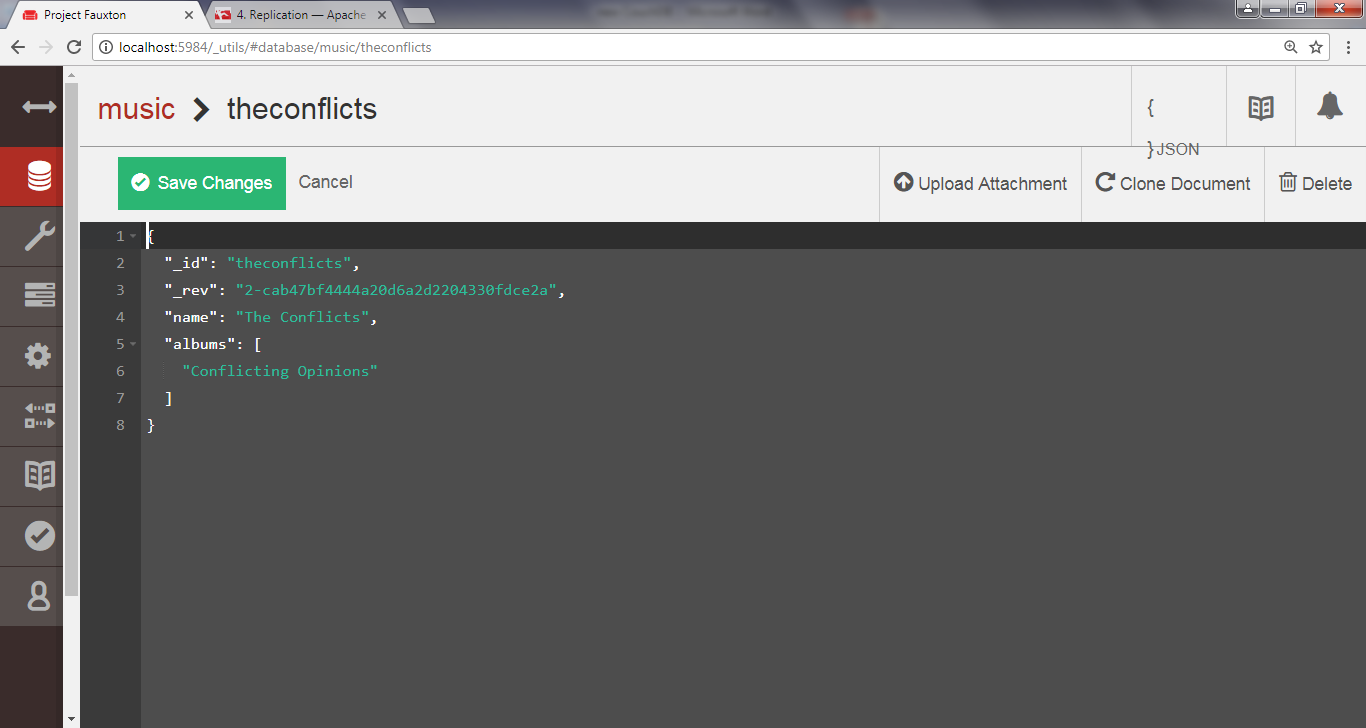
"ok":true,  
 "id":"theconflicts",  
 "rev":"2-cab47bf4444a20d6a2d2204330fdce2a"

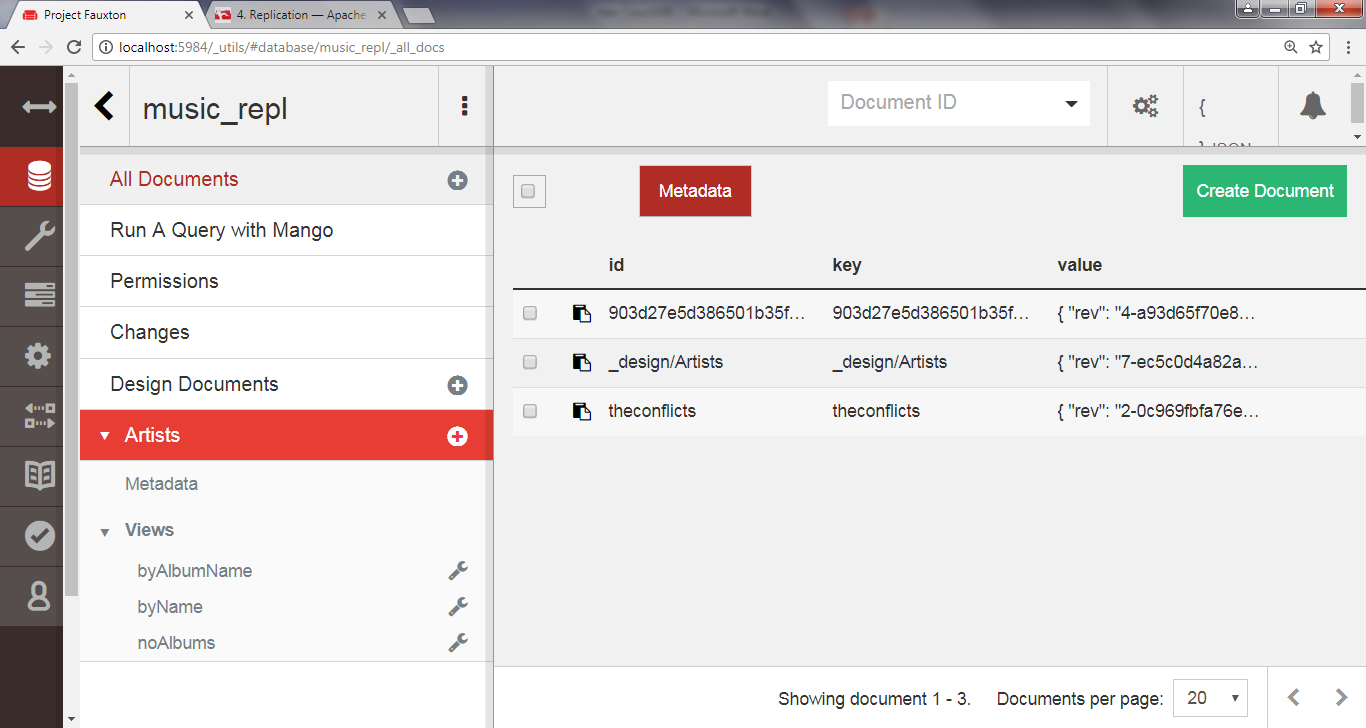
}

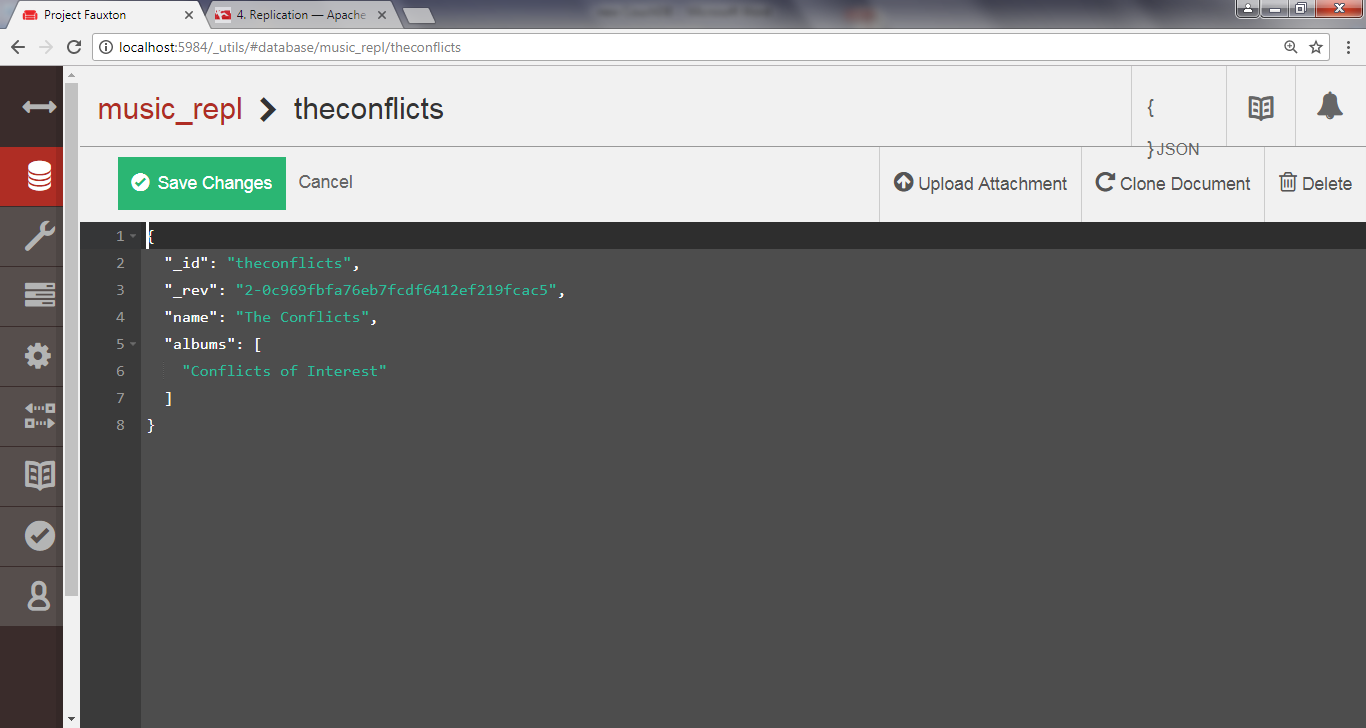


At this point, both the music and music\_repl databases have a document with an \_id value of theconflicts.







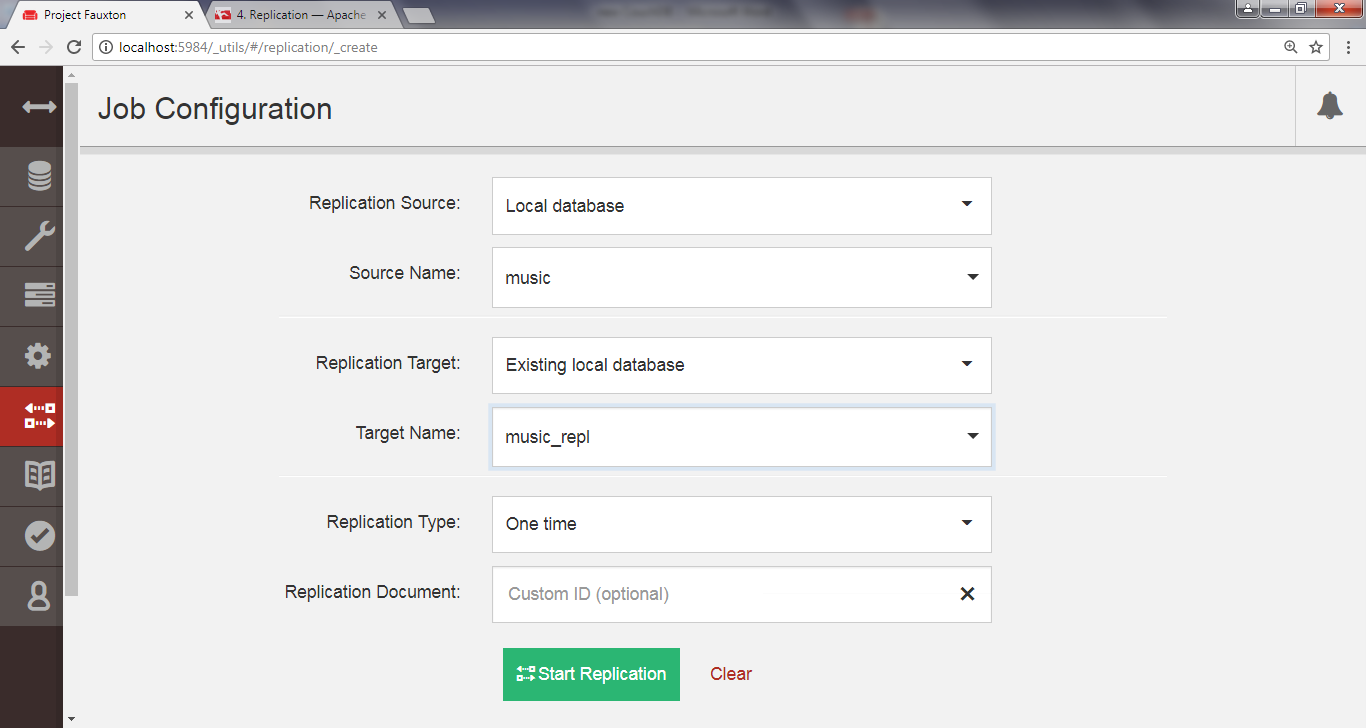


Both documents are at version 2 and derived from the same base revision (1-e007498c59e95d23912be35545049174).

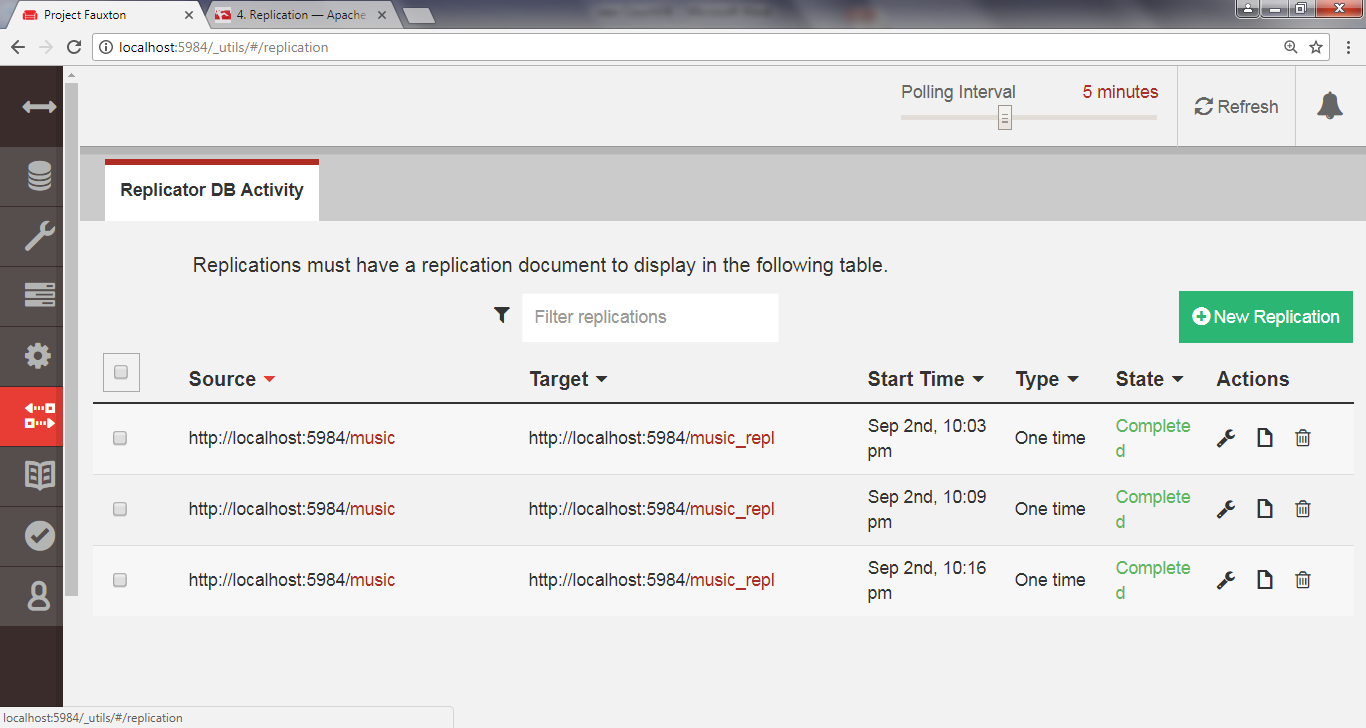
 Now the question is, what happens when we try to replicate between them?

**Resolving Conflicts**

With our document now in a conflicting state between the two databases, head back to the Replicator page and kick off another replication.



If you were expecting this to fail, you may be shocked to learn that the operation succeeds just fine.



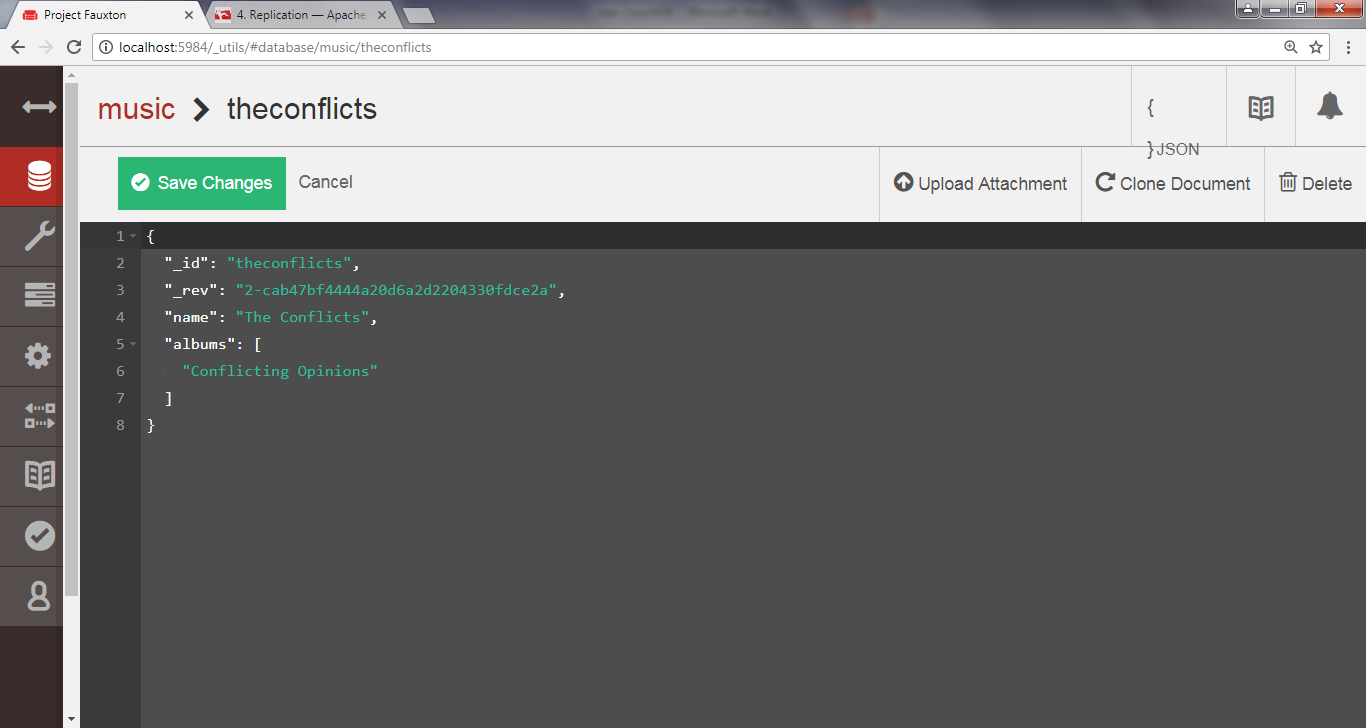
So, how did CouchDB deal with the discrepancy?

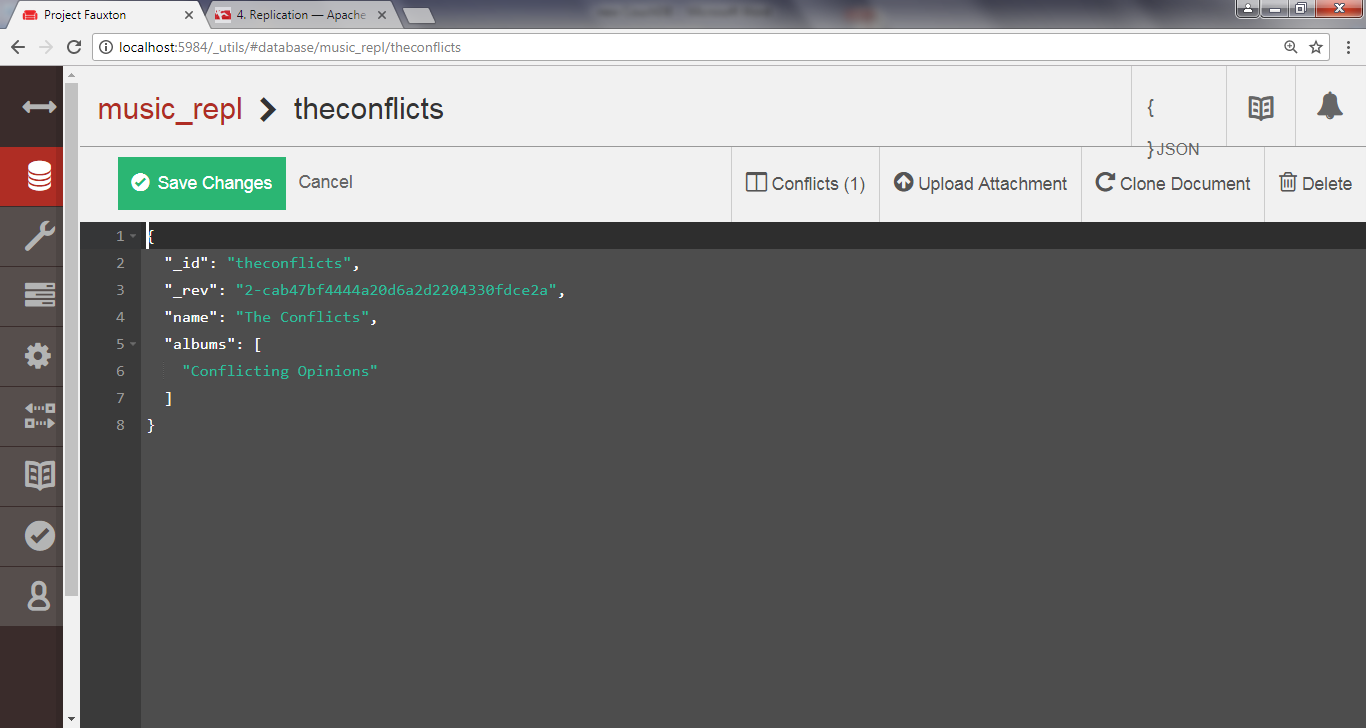
It turns out that CouchDB basically just picks one and calls that one the winner.

Using a deterministic algorithm (

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **deterministic algorithm** is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states From <<http://en.wikipedia.org/wiki/Deterministic_algorithm>>

) all CouchDB nodes will pick the same winner when a conflict is detected.





However, the story doesn’t end there.

CouchDB stores the unselected “loser” documents as well so that a client application can review the situation and resolve it at a later date.

To find out which version of our document won during the last replication, we can request it using the normal GET request channel.

By adding the conflicts=true URL parameter, CouchDB will also include information about the conflicting revisions.

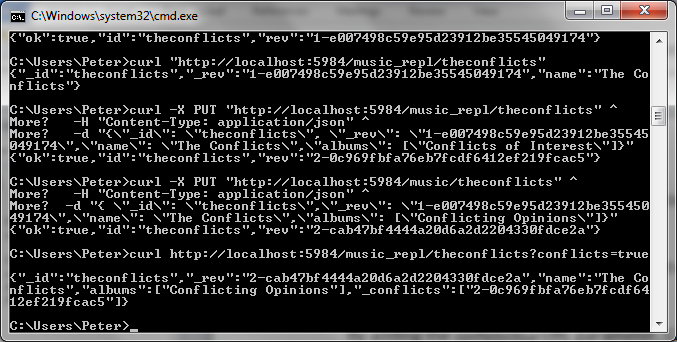
**curl** [**http://localhost:5984/music\_repl/theconflicts?conflicts=true**](http://localhost:5984/music_repl/theconflicts?conflicts=true)

{

"\_id":"theconflicts",  
 "\_rev":"2-cab47bf4444a20d6a2d2204330fdce2a",  
 "name":"The Conflicts",  
 "albums":["Conflicting Opinions"],  
 "\_conflicts":[

"2-0c969fbfa76eb7fcdf6412ef219fcac5"  
 ]

}



So, we see that the second update won.

Notice the \_conflicts field in the response.

It contains a list of other revisions that conflicted with the chosen one.

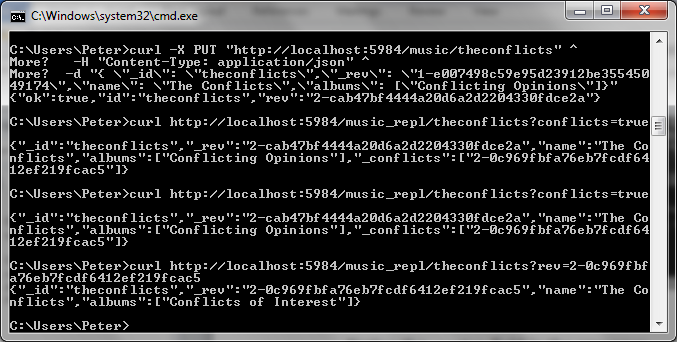
By adding a rev parameter to a GET request, we can pull down those conflicting revisions and decide what to do about them.

**curl** [**http://localhost:5984/music\_repl/theconflicts?rev=2-0c969f**](http://localhost:5984/music_repl/theconflicts?rev=2-0c969f)**...**

{

"\_id":"theconflicts",  
 "\_rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5",  
 "name":"The Conflicts",  
 "albums":["Conflicts of Interest"]

}



CouchDB does not try to intelligently merge conflicting changes.

How to merge two documents is highly application-specific, and a general solution isn’t practical.

In our case, combining the two albums arrays by concatenating them makes sense, but one could easily think of scenarios where the appropriate action is not obvious.

For example, consider you’re maintaining a database of calendar events.

One copy is on your smartphone; another is on your laptop.

You get a text message from a party planner specifying the venue for the party you’re hosting, so you update your phone database accordingly.

Later, back at the office, you receive another email from the planner specifying a *different* venue.

So, you update your laptop database and then replicate between them.

CouchDB has no way of knowing which of the two venues is correct.

The best it can do is make them consistent, keeping the old value around so you can verify which of the conflicting values should be kept.

It would be up to the application to determine the right user interface for presenting this situation and asking for a decision.

**Wrap-Up**

We’ve seen how to do a pretty wide range of tasks with CouchDB, from performing basic CRUD operations to building views out of mapreduce functions.

We learned how to perform ad hoc replication between databases and how to detect and resolve conflicts.

**CouchDB’s Strengths**

CouchDB is a robust and stable member of the NoSQL community.

Built on the philosophy that networks are unreliable and hardware failure is imminent, CouchDB offers a heartily decentralized approach to data storage.

Small enough to live in your smartphone and big enough to support the enterprise, CouchDB affords a variety of deployment situations.

CouchDB is as much an API as a database.

There are an increasing number of alternative implementations and CouchDB service providers built on hybrid back ends.

Because CouchDB is made “of the Web, for the Web,” it’s fairly straightforward to layer in web technologies—such as load balancers and caching layers—and still end up with something that’s true to CouchDB’s APIs.

**CouchDB’s Weaknesses**

CouchDB’s mapreduce-based views, while novel, can’t perform all the fancy data slicing you’d expect from a relational database.

In fact, you shouldn’t be running ad hoc queries *at all* in production.

Also, CouchDB’s replication strategy isn’t always the right choice.

CouchDB replication is all or nothing, meaning all replicated servers will have the same contents.

There is no sharding to distribute content around the datacenter.

The principal reason for adding more CouchDB nodes is not to spread the data around so much as to increase throughput for read and write operations.

Finally

CouchDB’s attention to robustness in the face of uncertainty makes it a great choice if your system must stand up to the harsh realities of the wild Internet.

By leveraging standard webisms like HTTP/REST and JSON, CouchDB fits in easily wherever web technologies are prevalent, which is increasingly everywhere.

Inside the walled garden of a datacenter, CouchDB can still make sense if you commit to managing conflicts when they arise or if you pursue an alternative implementation like BigCouch, but don’t expect to get sharding right out of the box.

Other features include ease of backups, binary attachments to documents, and a system for developing and deploying web apps directly through CouchDB with no other middleware. (Try CouchDB for your next data-driven web app; you won’t be disappointed! )

References:

Seven Databases in Seven Weeks

Also See: http://en.wikipedia.org/wiki/CouchDB