

**Team**: Green Bay

**Professor**: [Kristin Tufte](https://trec.pdx.edu/research/researcher/Tufte/5273)

**Course**: Cloud and Cluster Data Management

**Project Part 2**: System Introduction

**Date**: November 4th, 2018

**Members:** Kevin Jacobson, Tim Pugh, Elijah Rich-Wimmer, Jason Larson Jr, Scot Lambert

**Selected System:** CouchDB

**1: Data Model**

**a** What is the data model of your system? Document Model

**b** Give an example of a data stored in your system and the

function to insert data.

Documents in CouchDB are stored in JSON. To interact with CouchDB we use simple http commands and the REST api it provides.

To add data we can create a database using a simple http PUT with the name for the database just after the db address, which we will call “person”. We get a positive response in JSON that it was created.



Command: curl -X PUT http://admin@127.0.0.1:5984/person

Response: {“ok”:true}

Then to add a document to the new database, we simply provide a unique key in form of http://<dbaddr>/person/<key>. And using curl we use the -d switch and include our document data:



Command and response repeated here due to screenshot turning out rather small:

Command: curl -X PUT http://admin@127.0.0.1:5984/person/1126396 -d '{"name":"Steve G. Miksell","type":"person"}'

Response: {"ok":true,"id":"1126396","rev":"1-f1e0b992924d44ea1cb6203760bc0a74"}

Note that we get a “rev” field in the response which notes a revision version. If we want to update, add, or remove, data in this document we must also supply the current “rev” value. Essentially you must first retrieve the document (the “rev” value could have been changed by someone else!), add your changes, and send the data again but also include the current “rev” value.



Command: curl -X PUT http://admin@127.0.0.1:5984/person/1126396 -d '{"\_rev":"1-f1e0b992924d44ea1cb6203760bc0a74","name":"Steve G. Miksell","type":"person","works written":[627154]}'

Response: {"ok":true,"id":"1126396","rev":"2-1f5d1da683a60589bb961383874def73"}

To view this document again we could simply perform a GET using the known key:

Command: curl -X GET http://admin@127.0.0.1:5984/person/1126396  
Response: {"\_id":"1126396","\_rev":"2-1f5d1da683a60589bb961383874def73","name":"Steve G. Miksell","type":"person","works written":[627154]}

This is a simplistic way of entering data. In practice we also want to create “design documents” which help serve as templates for validating data to be imported.

**2: Query Support**

**http://docs.couchdb.org/en/2.2.0/api/database/find.html**

**A:** Queries in CouchDB are supported by a JSON query syntax sent to an API endpoint using an http request. The language is referred to as Mango. The query is structured as a JSON request body then sent to the /db/\_find endpoint as a POST command. The body must contain a selector field to select some documents. You can also set which parts of each selected object are returned by using the ‘fields’ field. Which index the database management system should use can also be specified. The API also allows you to skip the first n results or limit the number of results returned. CouchDB supports paging with the bookmark field. Logic can be built into the selector JSON object by using special operators that are prepended with a ‘$’. The response will contain a json body of all of the selected objects with a standard http response code.

**B:**

Example Queries over the DBLP dataset using our proposed data model:

Over the person database looking for all people who are an author of “The Titanic”

Command: curl -X POST http://admin@127.0.0.1:5984/person/\_find

JSON Body:

{  
 **"selector"**: {  
 **"works\_cited"**: {**"$elemMatch"**: {$eq: “The Titanic”}

}  
 },  
 **"fields"**: ["id", "works\_cited"],  
}

Response:

{  
 **"docs"**: [  
 {  
 **"id"**: "176694",  
 **"works\_cited"**: [“The Titanic”, “My Fav Book”]  
 }

]

}

Query 2: Range query to get all people with an id between 9753 and 12343

Command: curl -X POST http://admin@127.0.0.1:5984/person/\_find

JSON Body:

{  
 **"selector"**: {  
 **"id"**: **$and**:

[{**"$gt"**: 9753}, {**$lt**: 12343}]

}  
 }  
}

Response:

{  
 **"docs"**: [  
 {  
 **"id"**: "9874",

**“name”**: Gustavo Francisco  
 **"works\_cited"**: [“The Titanic”, “My Fav Book”]  
 },

{  
 **"id"**: "9877",

**“name”**: Augustus Caesar  
 **"works\_cited"**: [“The Titanic”, “some other book”]  
 },

{  
 **"id"**: "12340",

**“name”**: Michael Jackson  
 **"works\_cited"**: [“Another book”, “My Fav Book”]  
 },

]

}

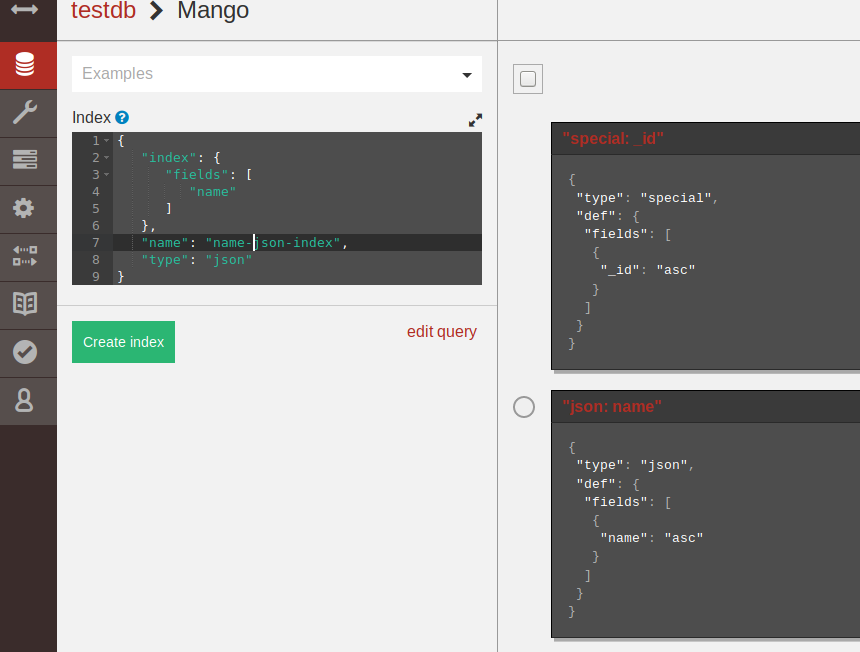
**3: Indexes**

CouchDB stores data as "documents", as one or more field/value pairs expressed as [JSON](https://en.wikipedia.org/wiki/JSON). Field values can be simple things like strings, numbers, or dates; but [ordered lists](https://en.wikipedia.org/wiki/Array_data_structure) and [associative arrays](https://en.wikipedia.org/wiki/Associative_array) can also be used. Every document in a CouchDB database has a unique id and there is no required document schema.

CouchDB guarantees [eventual consistency](https://en.wikipedia.org/wiki/Eventual_consistency) to be able to provide both availability and partition tolerance.

The stored data is structured using views. In CouchDB, each view is constructed by a [JavaScript](https://en.wikipedia.org/wiki/JavaScript) function that acts as the Map half of a [map](https://en.wikipedia.org/wiki/Map_(higher-order_function))/reduce operation. The function takes a document and transforms it into a single value that it returns. CouchDB can index views and keep those indexes updated as documents are added, removed, or updated.

See below for screen shots of index creation.

****

**4: Transactions and Concurrency Control**

Before we talk about concurrency in CouchDB, let's spend a moment talking about consistency. CouchDB is setup to support eventual consistency, as opposed to strong consistency. CouchDB favors eventual consistency so that if there is network hiccups or nodes disappear, the system can remain available. This may produce an answer, but it does not guarantee that the answer is up to date. In other words, CouchDB prefers to serve up an answer instead of waiting for the network hiccups to be resolved and providing a “consistent” answer. Thus, CouchDB favors availability over accuracy.

Let’s recall the CAP Theorem:

* Consistency: All database clients see the same data, even with concurrent updates.
* Availability: All database clients can access some version of the data
* Partition tolerance: The database can be split over multiple servers.

When dealing with a distributed system across a network, we’re only offered typically a strategy to implement two of the three options. Another item we must confront is how to make sure the data is “synced up”, or consistent. For example, let's say some new data is inserted into CouchDB, but it’s stored on some server A. A request comes in for the data but is routed through server B, which hasn’t synced up with A yet. Depending on your how important it is for your data to be exact, this can pose issues a developer needs to address. One could opt to pick availability over consistency, in which case an answer is served up but the request from server B may not wait to sync up with server A. If one chose consistency over availability, we’d get the opposing situation: we’d have to wait for server B to sync up with server A, and this will cause a bit of a delay and prevent us from fully utilizing the resources on the database. In the case of CouchDB as we stated it operates with eventual consistency. It has a B-tree storage system so that it self-balances, and provides a consistent logarithmic time for inserting data, deleting data, and searching for data.

CouchDB also uses MapReduce to get our results. This allows some parallelization of operations which enhances the operation of CouchDB.

With the above points noted, we can now discuss how CouchDB uses Multi-Version Concurrency Control to allow concurrent reading of data. If we didn’t have a concurrency control method, then we could see incomplete writes of data or data that is not consistent. When an MVCC database (like CouchDB) must work on some data that is stored, the previous version of the data is retained, and a new version is created. Since CouchDB uses an MVCC method for concurrency, we do not deal with locks like relational databases use but are typical for NoSQL distributed systems. This gives us an advantage of using available resources during larger loads on the database and allows it to run very fast. An example of how this works in practice:

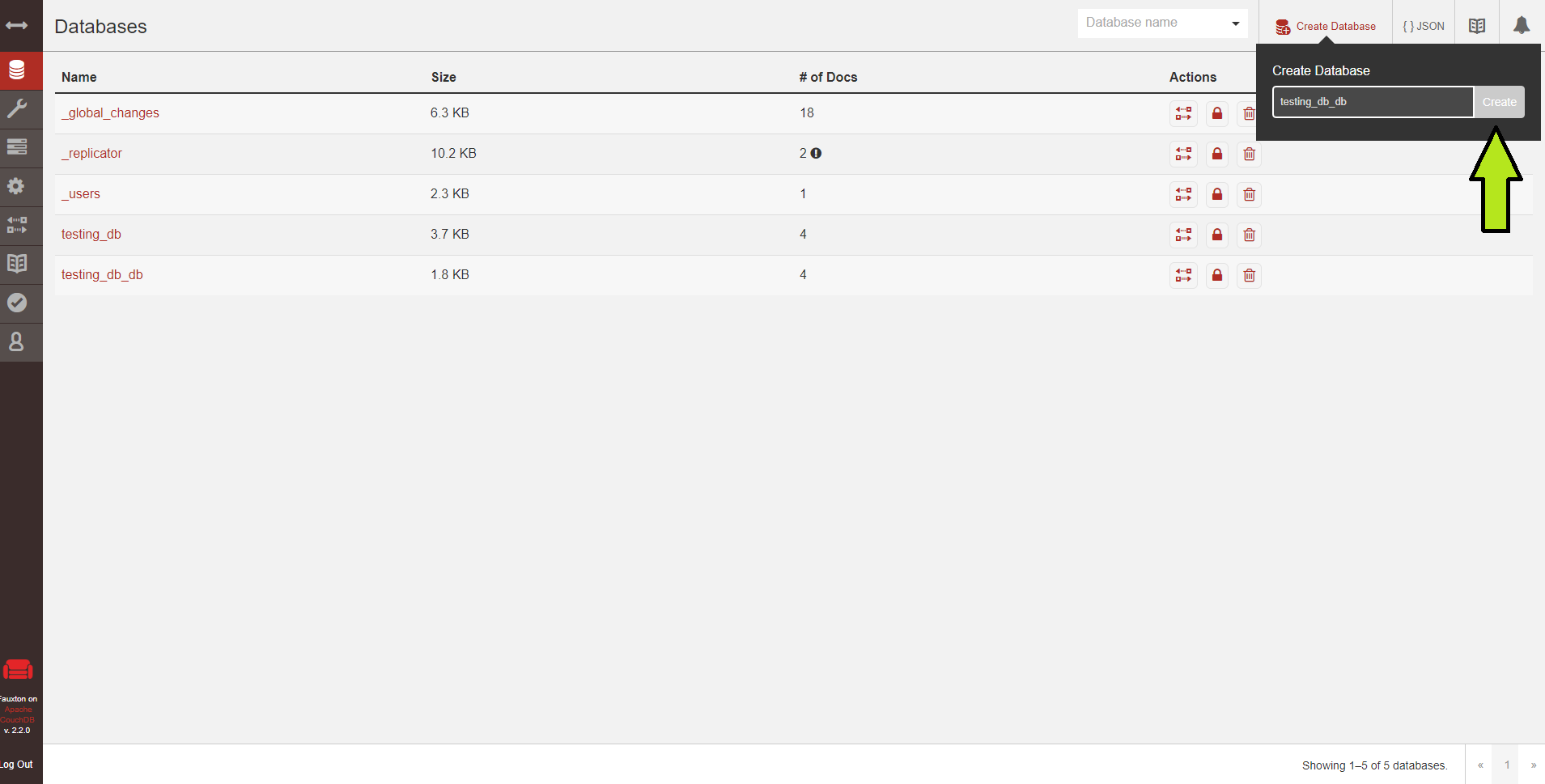
User A requests a document first and begins reading the document. User B access the document and modifies it. Since we use versioning, User A can still view their document. User C comes along and wants to read the document, and they pull the latest version (the one modified by User B), all while A is still reading the previous old version. This syncs up with the eventual consistency CouchDB has: the data will eventually be consistent but favors availability. User B was able to read and modify the document, while user A was viewing it, and made a new copy without having to wait. This concurrency allows more work to take place, which as we previously mentioned, makes good use of available resources.

It’s worth noting that with CouchDB, it does not support transactions that are “bulk”. For example, we cannot modify document X and Y in the same transaction and roll it back. We would need to complete the transaction on X than move on to the transaction of Y. All items in a transaction must modify a single document. When an error takes place, it’s not undone like one would expect with a relational database, but a new transaction takes place to provide the fix. Thus, CouchDB guarantees a transaction to succeed or fail. There is no state in between, the actions are atomic on a single document.

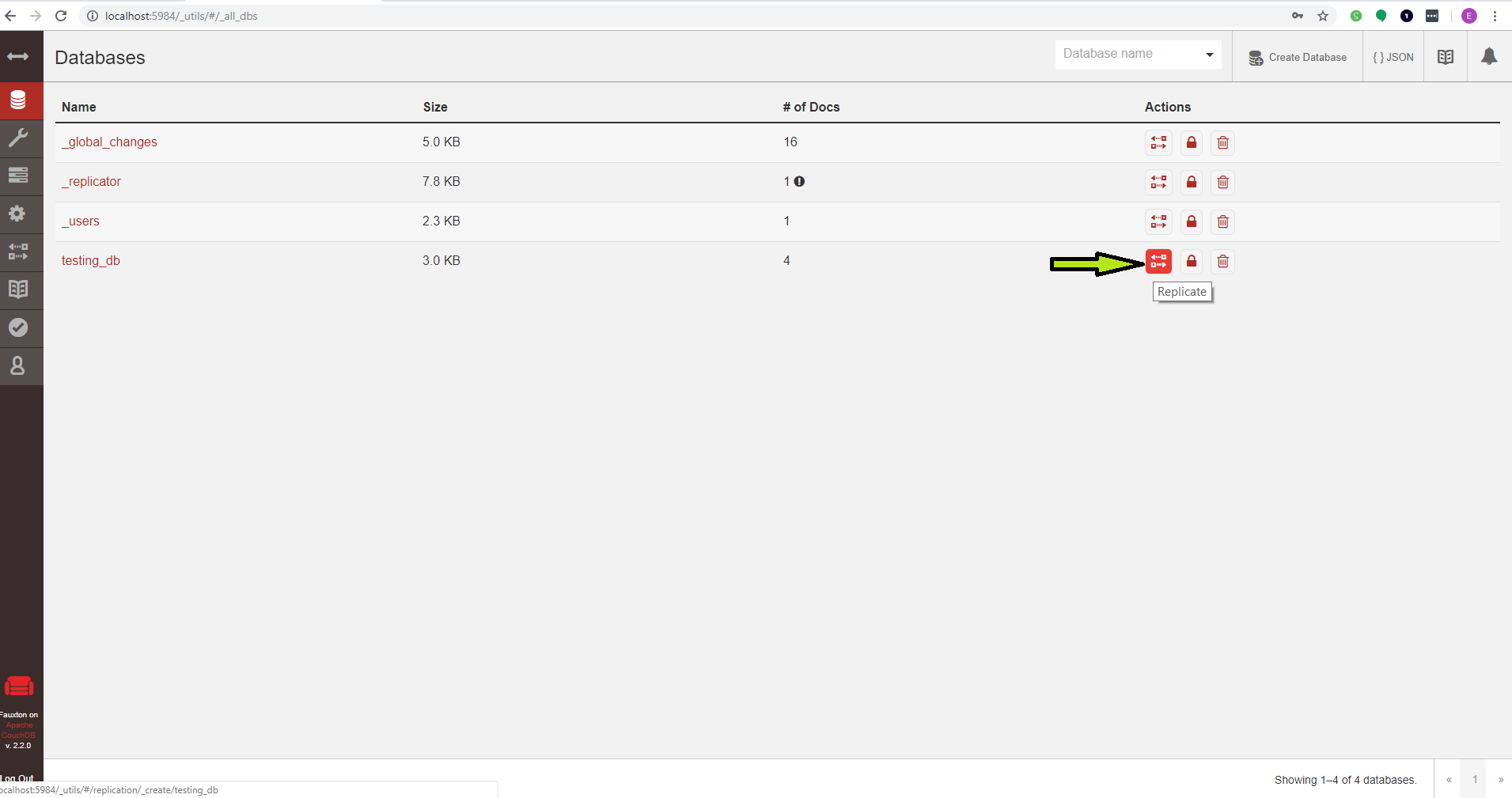
**5: Scalability**

* **Yes, this system supports replication.** *(This includes both Database replication and shard replication) The prompt wasn’t exact on what kind of replication should be described, so both techniques are described below.)*
  + Automatic replication is configured automatically as a system preference (See below with sharding).
  + Additionally, database replication can be achieved through the Scheduling Replicator.
* **The steps are as follows:**

1. Create a new database. This is only necessary if we configure the replicator to copy to a preexisting database.

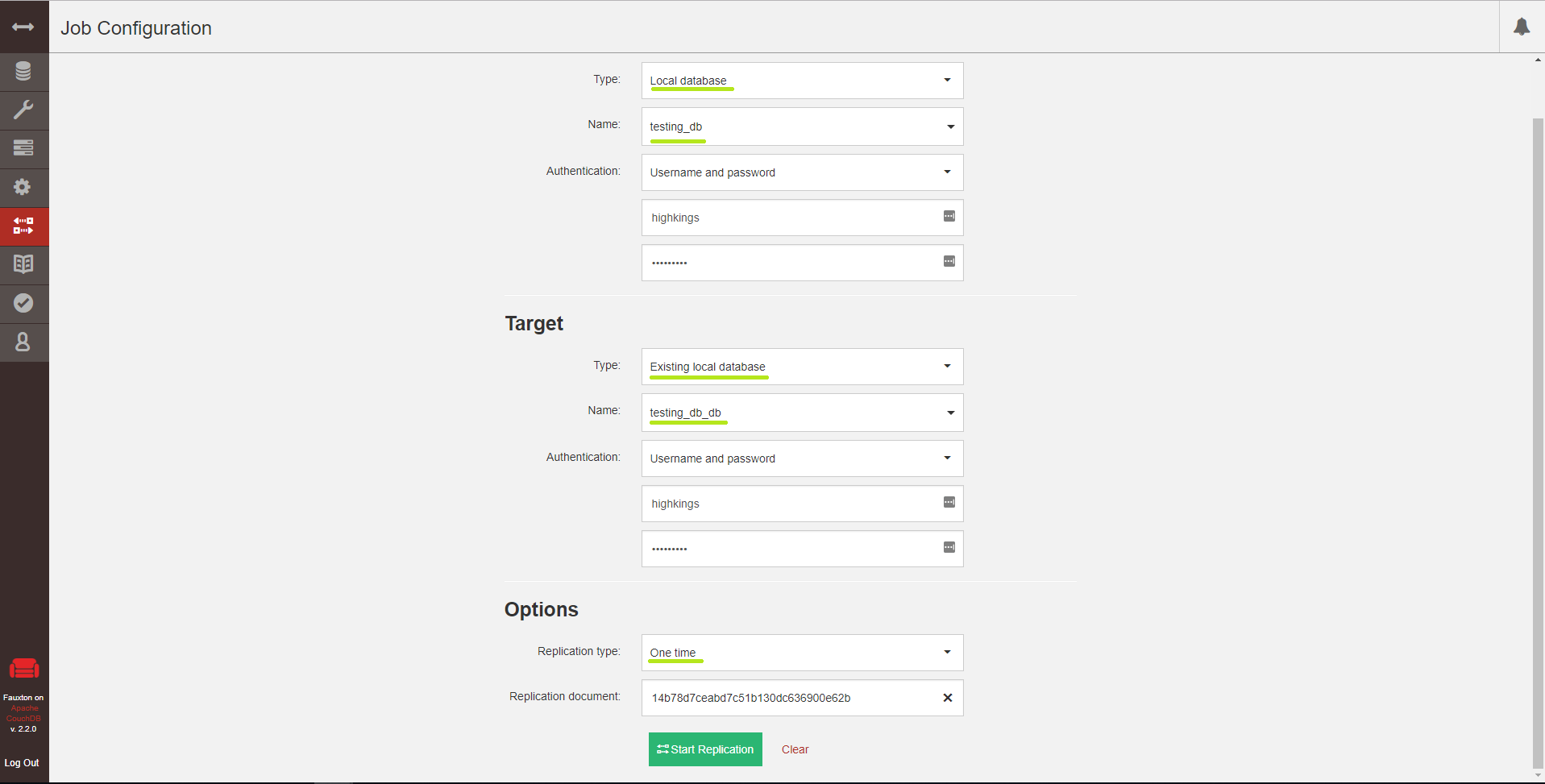
****

1. Navigate to the databases menu and click the Replicate button
   1. *(As the illustration below highlights)*

****

**3.Fill out replication details**

* In this specific example, we used a local database. However, external nodes can be set up as the target or destination in the exact same fashion. *(This was tested to confirm)*.

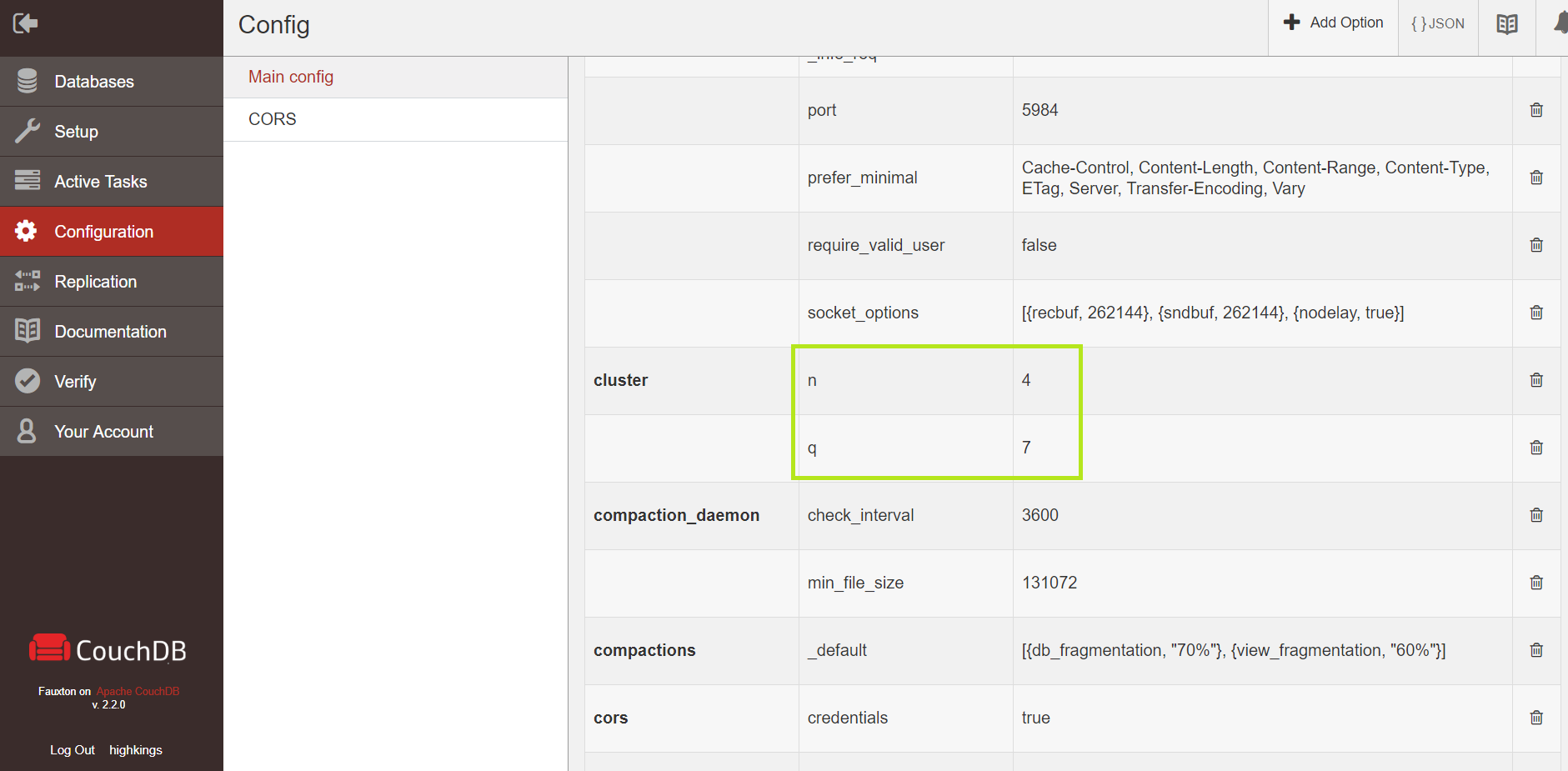
****

* **Final Results demonstration:**

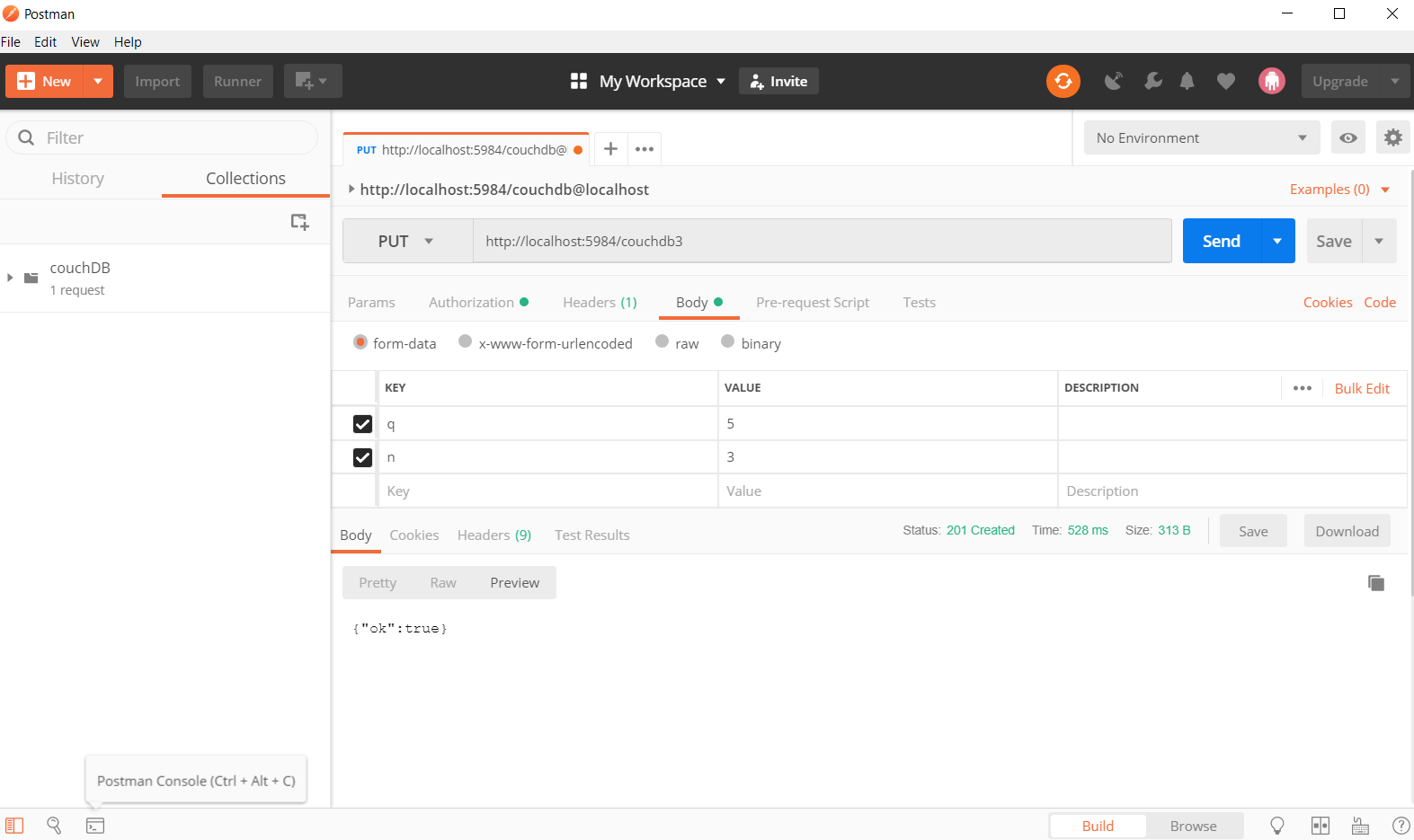
*This isn’t exactly necessary. However the following image is to demonstrate that the system successfully replicated the database in its entirety.*

****

* **Yes this system supports sharding.**
  + In our case, sharding is initialized and managed automatically. We only need to manually relocate shard balance if certain data pieces are.
    - The number of shards and shard replicas is controlled in a node’s configuration file. Every time a database is created, sharding will occur on the system following the settings previously saved on the server.
    - **For our system**: The configuration demonstrated below communicates that for every database we have created currently, there are 7 shards for this database with 4 replicas of each shard distributed across the system.



* + Additionally, manual sharding and shard replication settings may declared per database upon creation using a PUT request: as the following postman declaration demonstrates.



* + **If necessary**: to manually configure sharding for optimization, we can follow the steps as narrated by the documentation: (*Section 11.5.2: Moving a shard:*[***http://docs.couchdb.org/en/stable/cluster/sharding.html?highlight=sharding#resharding-a-database-to-a-new-q-value***](http://docs.couchdb.org/en/stable/cluster/sharding.html?highlight=sharding#resharding-a-database-to-a-new-q-value))
    - However, this step is not necessary for our initial setup.
  + Resharding a database will require a complete recreation of that database. *(Using a temporary database to hold the shards during transformation.)*

**Citations**

**Eventual Consistency.” *1.3. Eventual Consistency - Apache CouchDB 2.2 Documentation*, Apache Software Foundation, docs.couchdb.org/en/stable/intro/consistency.html. Revision b7b1f244**

**Replicator.” 1*1.3. Replicator- Apache CouchDB 2.2 Documentation*, Apache Software Foundation, docs.couchdb.org/en/stable/replication/replicator.html**

**Sharding.” 1*1.5. ShardingApache CouchDB 2.2 Documentation*, Apache Software Foundation, http://docs.couchdb.org/en/stable/cluster/sharding.html?highlight=sharding#**

**Anderson, J. Chris, et al. “Recipes.” *Recipes*, O’Reilly Media, guide.couchdb.org/draft/recipes.html.**