www.phparch.com

NOVEMBER 2024
Volume 23 - Issue 11

php[architect]

The Magazine for PHP Developers

Lounging Around with PHP

- PHP, CouchDB, and Chill
- Building Serverless PHP applications

Also Inside:

Education Station - Deploying Your Application
PHP Foundry - Old School Recipes for Output
Security Corner - Cybersecurity Awareness
Back Office - Exception Reporter Part 1
Readable Code - The Origins of PHP
Community Corner - Learning By Battling Snakes?
Finally - Taking Responsibility

Full-stack logging & observability

See the forest for the trees

Gain insights into your errors, application logs, and other event streams with a powerful query language and flexibile visualizations.

BadgerQL

```
filter event_type::str == "feature"
| stats unique(user_id::int) as count by name::str
| sort count
| limit 5
```

4 lines

Top Used Features

by # of users — Open in Query Editor

- Your Feature A → 555 users
- Your Feature B → 540 users
- Your Feature C → 404 users
- Your Feature D → 302 users
- Your Feature E→ 200 users

Turn events into Insights

Get actionable intelligence from your logs, no `tail` required. Who wants to sit around tailing logs all day? All the events you send to Honeybadger Insights can be queried, analyzed, and even turned into metrics. Application logs, clickstream data, audit trails—you name it—squeeze the value from the volume!

https://honeybadger.io

PHP, CouchDB, and Chill

Eric Van Johnson

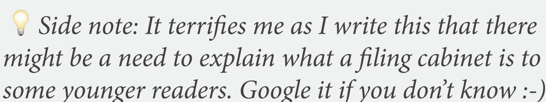
I have relationship issues. Not in my personal life. My wife and I have been married for over 35 years. No, my problems are with my data. Like many developers my age, when we wanted to store data for our application, we would reach for a database server like MySQL, Postgres, MSSQL, or some other relational database. The relational database was "the way," and I never gave it much thought once I started down that path.

Database tables with relationships to other tables require some management. If you need to delete a record with a relationship, you must delete the data with a single relation to that record. Thanks to ORMs like Laravel Eloquent and Doctrine, this isn't too difficult to handle anymore, but it is still something you must remember to set up and configure. With relational databases, there were always use cases that bugged me. For me, it's phone numbers. I remember the first time I needed to store multiple phone numbers for a user. My first approach was creating columns on my table, such as `phone` and `work_phone`. Then, I was asked to add `cell_phone`. Then, the use case came up where the user might want to add their partner's or parents' numbers. I couldn't add a column `other_number` because the user might want to define multiple numbers. As you can guess, I created another table called phone_numbers with the number type and number and migrated my existing data to it. I set up that relationship, and it worked. But now I needed to remember that this relationship existed and to ensure all of a user's phone numbers were added to my queries

The next request was to mark one of the numbers as the `primary` number. As you can probably guess, I added a column `primary` to the `phone_numbers` table. But now I also need to ensure that each user only has one primary number. It's not a major issue, but it started feeling like little paper cuts for such a relatively small dataset.

How many of us added a `twitter_handle` to our database because back in the day, what else was there to worry about? Now, a new social network pops up every other month

As developers, we can overcome any data issue with enough code, but what if you didn't have to worry about that? What if you treated your data like documents in filing cabinets?



> *Side note: It terrifies me as I write this that there might be a need to explain what a filing cabinet is to some younger readers. Google it if you don't know :-)*

Enter the world of the "Document Store." After years of working with relational databases, this simple concept was mind-bending to me. My coding brain could not comprehend such a straightforward approach: Put all your data into one document?

I was aware of document stores, specifically MongoDB. I would play with it but never understood when I would need such a solution. However, I started working for a company that was using Apache CouchDB (https://couchdb.apache.org/[1] . It was initially so frustrating, and I never got my head around CouchDB at that time. It wasn't an issue because I wasn't a developer. I just needed to make sure my team managed the servers CouchDB was running on. As luck would have it, years later, I would hire someone from that development team, and we had several conversations about document stores and CouchDB specifically. They even convinced me to add it to one of our client's infrastructure stack, and today, I genuinely appreciate what document stores and CouchDB bring to the table

An Uncomfortable Feeling

One term you will need to get comfortable with is "eventual consistency." I hear you barking, big dog, "How is that even a thing, and when would that EVER be OK?" Stick with me; we'll get there, I promise. One more note, for years, document stores were not ACID compliant. MongoDB was the first ACID complaint document store I remember hearing about. As of writing this, I now see that the CouchDB documentation has a section "ACID Properties" (https://phpa.me/couchdb-overview[2] . I haven't familiarized myself with this overview, but it was worth mentioning for those who find this important.

One of the things that keeps drawing me to CouchDB as opposed to others like MongoDB are the drivers or the lack of drivers for it. You can find packages for languages and frameworks for CouchDB, but honestly, I have yet to find much of a benefit in using them because CouchDB works off the HTTP protocol, and it's outstanding. You can talk to it directly with something as simple as `curl` in the terminal. It uses `GET`, `POST`, `PUT`, and `DELETE`. In all these cases, CouchDB will return a JSON response to you.

1 https://couchdb.apache.org/

2 https://phpa.me/couchdb-overview

Getting It Up and Running

Installing CouchDB is pretty simple. There are packages for most distros as well as Docker solutions. I am not going to get into a lot of details about the installation process, but just be aware that if you are hosting CouchDB for production or even on the public intranet, you need to take some time and read the section of CouchDB outlining important security steps (https://phpa.me/couchdb-security[3] .

For this article, I will be using a Docker solution.

```
docker run -d --name archie1 -e COUCHDB_USER=admin
  -e COUCHDB_PASSWORD=elephpant
  -p 5984:5984 apache/couchdb
```

Figure 1.


```
1  {
2      "couchdb": "Welcome",
3      "version": "3.4.2",
4      "git_sha": "6e5ad2a5c",
5      "uuid": "2bb6d31545a257811820407a9b1fdb61",
6      "features": [
7          "access-ready",
8          "partitioned",
9          "pluggable-storage-engines",
10         "reshard",
11         "scheduler"
12     ],
13     "vendor": {
14         "name": "The Apache Software Foundation"
15     }
16 }
```


Now, when we open a browser and hit the new HTTP endpoint on our server, `http://batcomputer:5984/`, we are up and running. Yes, the “batcomputer” is one of the computers in my internal lab, so don’t judge me; my wife will do it for you.

Much like packages for CouchDB, I am not aware of any GUI clients for it worth getting, but fear not because CouchDB comes with Fauxton. You can access your Fauxton install in your browser by adding the path `/_utils` to your CouchDB instance. This means our URL is now `http://batcomputer:5984/_utils/`. You will be presented with a login screen for an admin user. For us, it’s the `COUCHDB_USER` and `COUCHDB_PASSWORD` variables we passed to the docker command. Once logged in, you should see something like Figure 2.

It’s nothing super exciting, but it is not bad for a quick Docker command to get up and running. At this point, you can click around, and you’ll see a lot of things you would expect to see in a database client. You see your databases, which, at this point, we don’t have any. You see a place for settings, replication, documentation, a place to manage users, and some other things.

Since we’re here, let’s create a database by clicking on "Create Database" in the upper right and let's call it `my-podcast` as shown in Figure 3.

Non-partitioned versus Partitioned is outside the scope of this article, but chances are you only need Non-partitioned.

Figure 2.

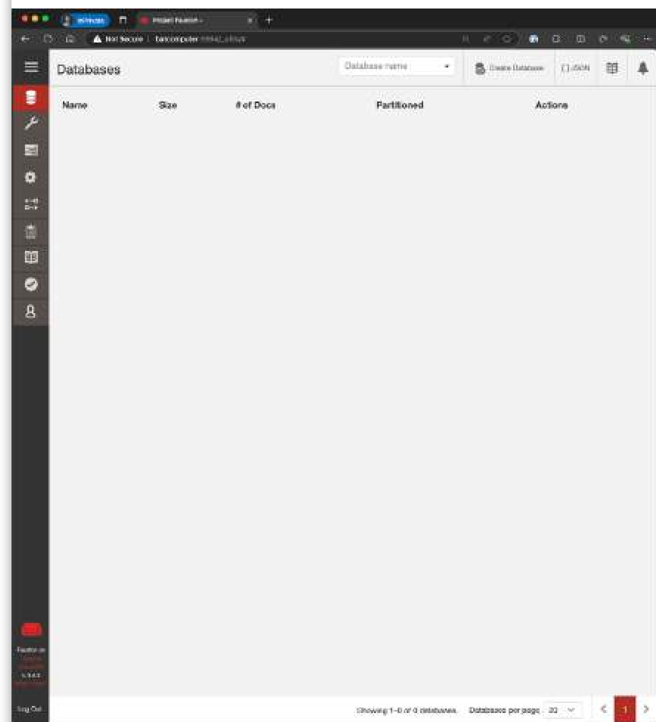
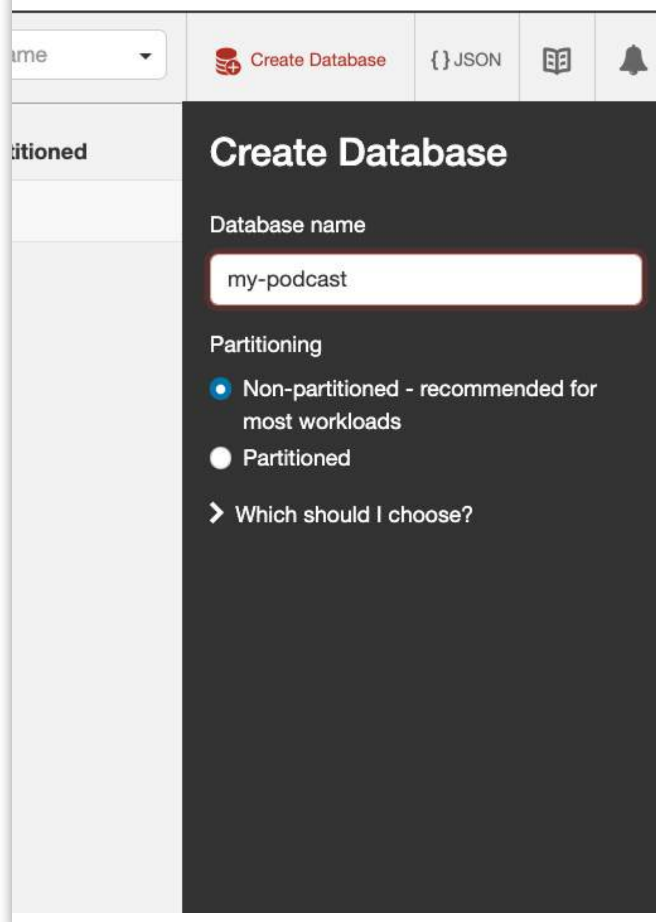


Figure 3.



3 https://phpa.me/couchdb-security

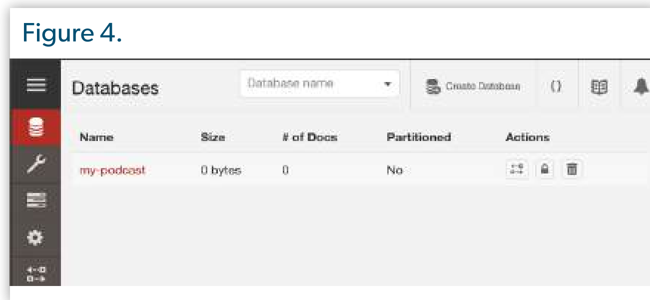
If you need Partitioned, then you understand what it is and why you need it.

No Schema

Now we have our database.

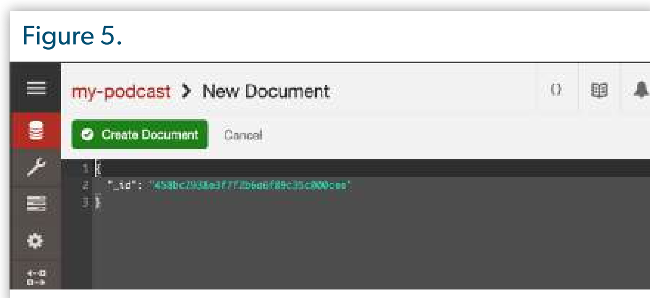
Let's click on our database and create a document. You'll notice that there is no concept of tables or making any sort of sub-division of the database. There are only documents at this level.

Figure 4.



The first thing you might notice is that it's just a JSON document. You might also notice an ID already assigned to the document. CouchDB autogenerates it, but it is not a simple increment ID. You can, however, change this id to anything you want. If you are adding a user with UUID associated with them, and this was a user document, then you might decide to change the document ID to that user's UUID. The critical thing to remember is that this ID must be unique to all the documents in the database, now and forever.

Figure 5.

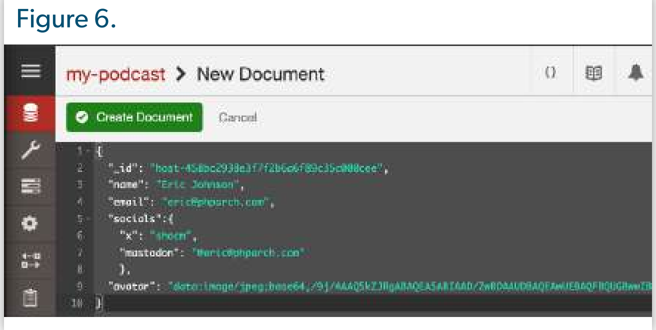


This is where the world of "no schema" really starts to hit you. You need to be a disciplined developer. It would help if you also had a general understanding of data structure and architecture. The freedom a schema-less database gives you will be unmatched for this added discipline.

I can leave this document ID and start building my JSON document. But later, you'll see that this will make it more challenging to know what's inside the document when looking at the database. This document will capture the podcast host's information, so I will use a very simple naming convention of prepending "host-" to the id and start adding the host's information as part of the JSON structure as shown in Figure 6.

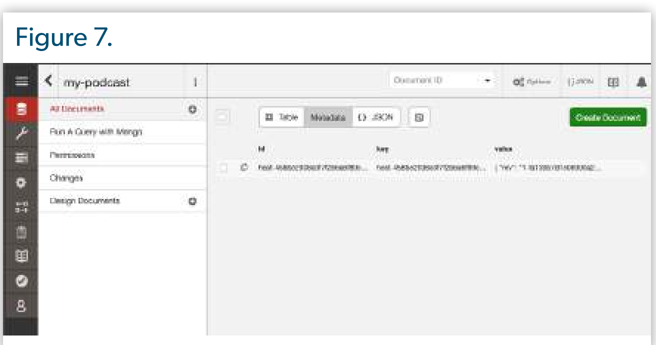
Currently, this is basic stuff. You will see that I created an object containing services and handles for `socials`. This is

Figure 6.



opposed to creating another table with a relationship in a relational database. I've also added an avatar to the record by base64 encoding an image. Now I click "Create Document," and I am done. I have my first record as seen in Figure 7.

Figure 7.

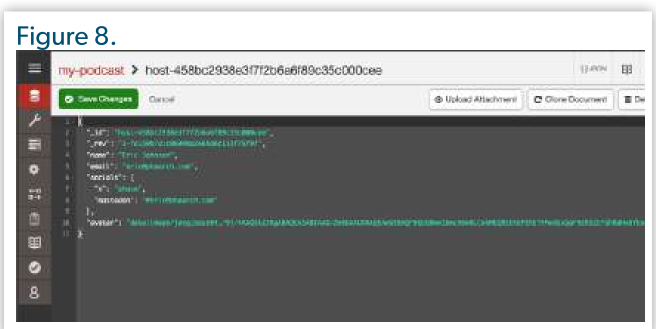


> 💡 *Sidebar: I am sure a few of you are already thinking, "This is nothing special. Most modern relational databases these days have a JSON column type that would take care of that socials section without the need for another table." This is true, but we are just getting started. Stick with me.*

A Revelation

Let's open up the record we just created.

Figure 8.



You will see that CouchDB has now added another key called `_rev`. Let's make a change to our record and see what that does. I am going to add another social handle, save the document, and open it up again.

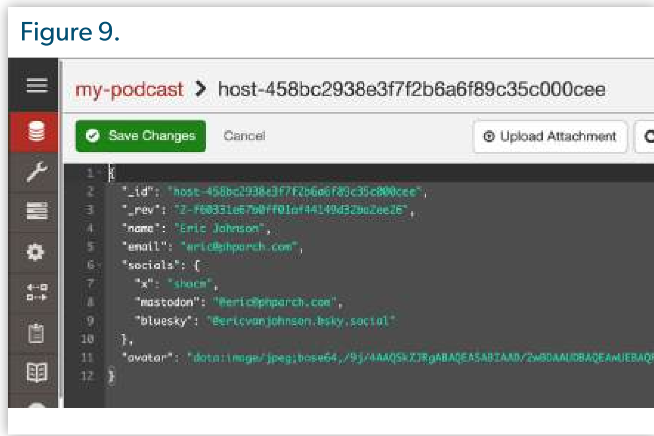


Figure 9.

You'll see that the first digit in the _rev has incremented by 1. This tells me how many times the document has been updated. And it will continue to count up every time the record is changed.

We don't have any information about the podcast itself, which would be good to track. With most databases, we would probably create another table called podcasts even if all we had at the time was this one podcast, but with CouchDB, we create a new document in the database.

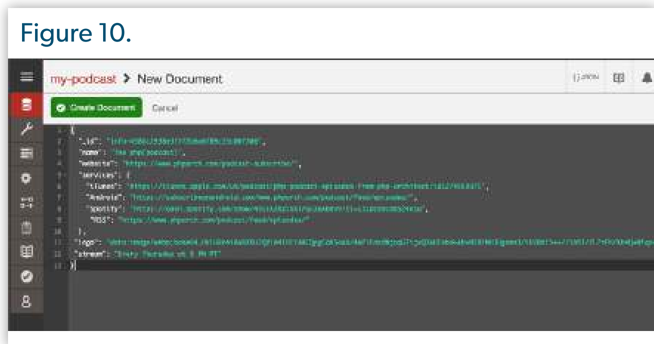


Figure 10.

I hope you are seeing some of the flexibility now. This document has its own set of data..

A new host joins the podcast. We are now at a critical stage. Do we add the second host to the same document the first host was in, or do we create another separate document? The good news and bad news is there is no wrong answer here. What I mean here is that the existing host document can be added to as much as you would like. If we want to go back to our original document, take this first host and add them to an object, and then add a second object to put the information on the new host, we can do that. This would require some coding updates in your application because the data for the first host won't be in the same place it was before; it will be a layer lower.

Personally, I would add another document, and that is what I am going to do. Like the "podcast info" document we created, the new host document does not need to match the other one. More importantly, maybe I want to start tracking both hosts' equipment and a mailing address. It is simple enough to add that information to the existing document and add it to the new document. No database migrations are needed.

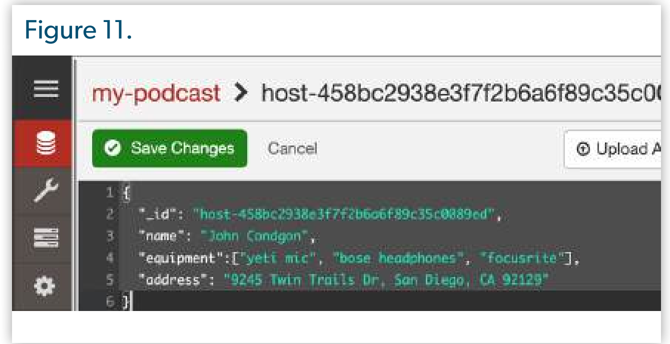


Figure 11.

CouchDB's documentation includes a section called Document Design Considerations[4] that is probably worth reading. Another section to look at is Best Practices[5] , which discusses more about the usage of CouchDB.

Things are Getting Serious

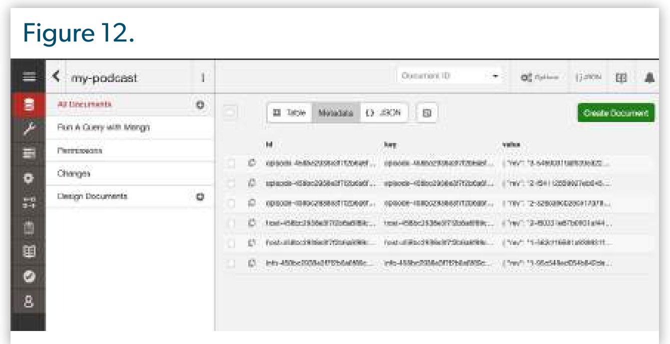


Figure 12.

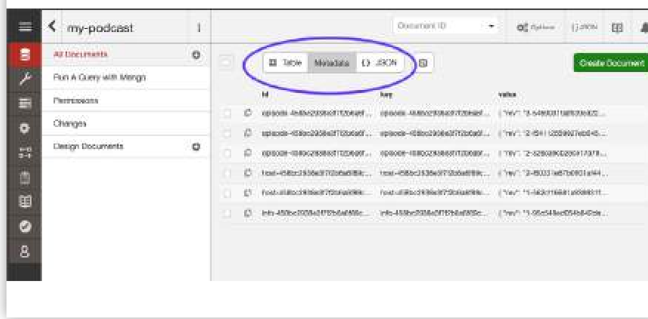
Alright, things are starting to build out. At this point, I've added some more documents. Now, our database has one podcast info document, two host documents, and three episode documents. But looking at this, I realize I should have included the episode number in the naming convention. But now I have a little problem because CouchDB doesn't like it if you mess around with the ID once the document has been created. You can try to change it if you want. First, you'll want to remove the _rev. Even then, if you change the ID, CouchDB will create a new document, and we will end up with four episode documents, with one being a clone with a different ID. Again, I will stress that you should take some time and carefully consider your naming conventions. But it's not the end of the world for us.

You may have noticed that we are looking at the "Metadata view" (Shown in Figure 13 on the next page) and that there is also a "JSON View" and "Table view." The JSON view just displays all the JSON in the documents. It's not a terrible

4 https://phpa.me/couchdb-docs

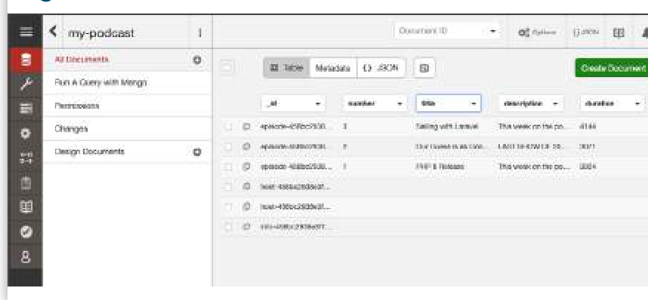
5 https://phpa.me/couchdb-best-practices

Figure 13.



option, but that is because we don't have thousands of documents of varying lengths in our database. The "Table" view is a nice option because it identifies all the keys in all our documents and lets us display them that way.

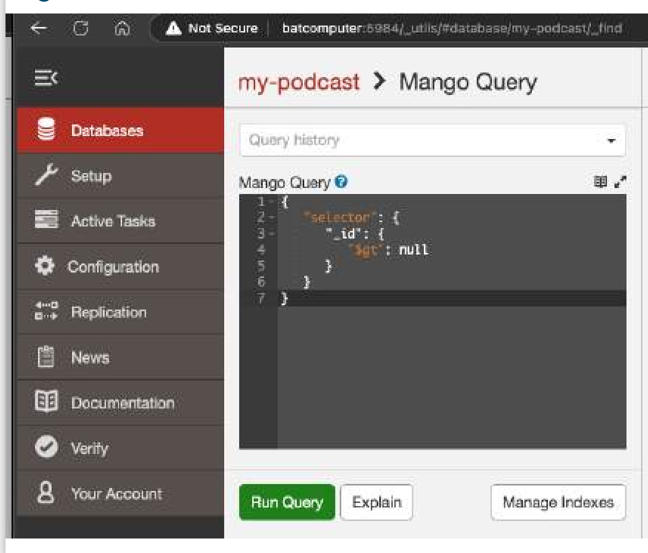
Figure 14.



This is better, but again, we don't have a lot of documents with different keys, so the list isn't that long. Also, I have records here that don't even include episode information. We will look at the section under "All Documents" called "Run A Query with Mango." Mango is the query language for CouchDB.

Let's write a Mango query that only shows episodes in order of episode number.

Figure 15.



The first thing we want to do, like a relational database, is define some indexes to help speed up the search. Let's do that first. We'll create an index for the ID itself because we know we want to search on that since it's what identifies an episode document by starting the ID with the word `episode`.

```
{
  "index": {
    "fields": ["_id"]
  },
  "name": "id-index",
  "type": "json"
}
```

Next, I will create an index for the episode number.

```
{
  "index": {
    "fields": ["number"]
  },
  "name": "number-index",
  "type": "json"
}
```

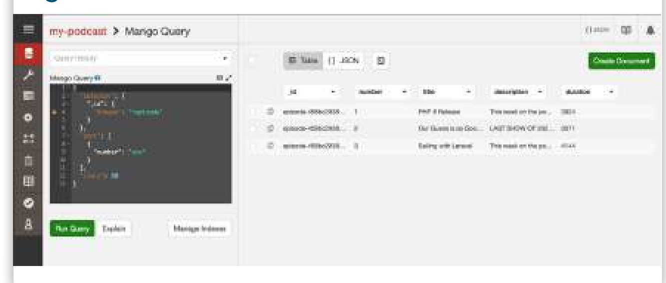
Now that I have those, I can run my mango query.

Listing 1.

```
1. {
2.   "selector": {
3.     "_id": {
4.       "$regex": "^episode"
5.     }
6.   },
7.   "sort": [
8.     {
9.       "number": "asc"
10.     }
11.   ],
12.   "limit": 50
13. }
```

And now I only see documents that start with the word "episode."

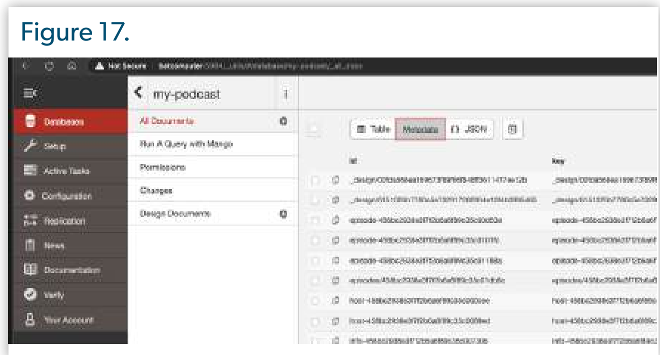
Figure 16.



This feels like progress, but you might notice something odd. If we go back and look at our complete database again, we will see new documents starting with the `_design/`. Well remember those indexes we created? They are now documents in the database as well. Also, notice the naming convention

it's using, which is prefixing the ID with the word _design and a / like a URL. It will fit nicely with a URL format if I ever need to make a curl call. I should have probably taken the same approach of using a / instead of a - to separate my ID for "host," "info," and "podcast," but honestly, I forgot about this 😁

Figure 17.

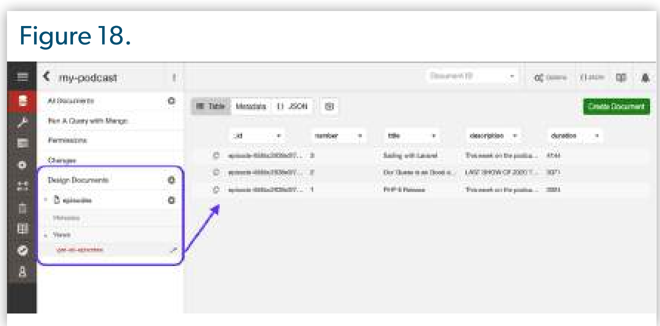


The next thing we will want to do is create our own "Design Document" view using MapReduce. I think of a design document as a stored procedure. These are just queries we want available to us. We will typically make sure we have good indexes for these queries. Let's create a design document to do what we did with Mango and return all documents with the word "episode" at the beginning of the ID.

```
function(doc) {
    if (doc._id && doc._id.startsWith('episode')) {
        emit(doc._id, doc);
    }
}
```

Now, we can run this view whenever we want a list of episodes.

Figure 18.



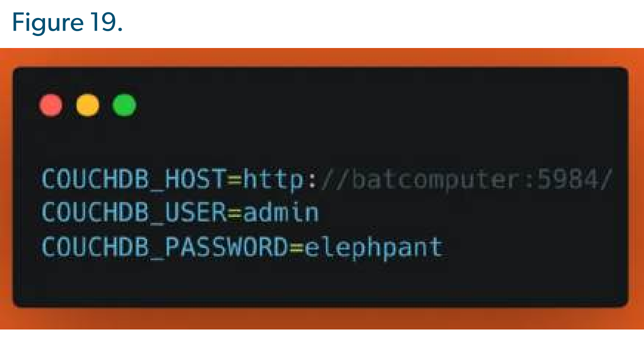
ENOUGH Already!!! What About the PHP Part?

OK, you've been patient up to this point. I've got more to cover on CouchDB, but let's get your PHP fix out of the way. Let's spin up a new Laravel app

```
laravel new couchdb
```

Now, let's add those CouchDB variables to our .env file. If this were a real application, you would probably want to add a config file and create a service to save you some coding, but we will reference the .env file everywhere we need this information.

Figure 19.



Next, we open our web route file and create a route to access all the documents in our document store using the Laravel HTTP client.

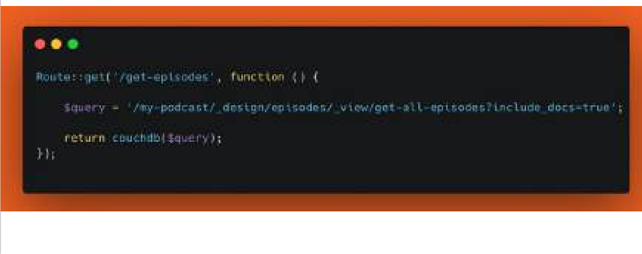
Figure 20.

If we go to the URL `https://couchdb.test/documents`, we'll get a JSON dump of all the documents in our my-podcast database.

Figure 21.

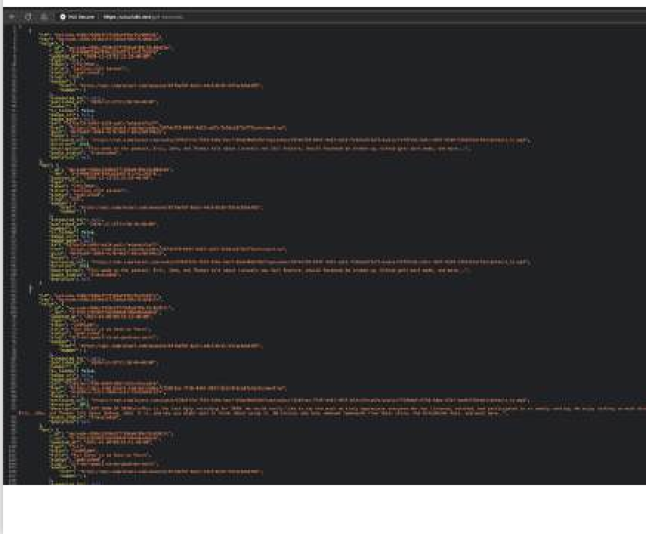
Let's add a call to the `get-all-episodes` design document view we created inside of CouchDB.

Figure 22.



Now we head over to `https://couchdb.test/get-episodes` and we see:

Figure 23.



I think you get the point. You can use `POST` to create new documents, `PUT` to update an existing document, and `DELETE` to remove a document. Before using the JSON in your application, you'll probably want to turn it into an object, an array, or a collection. That would be part of the "Service" you would create.

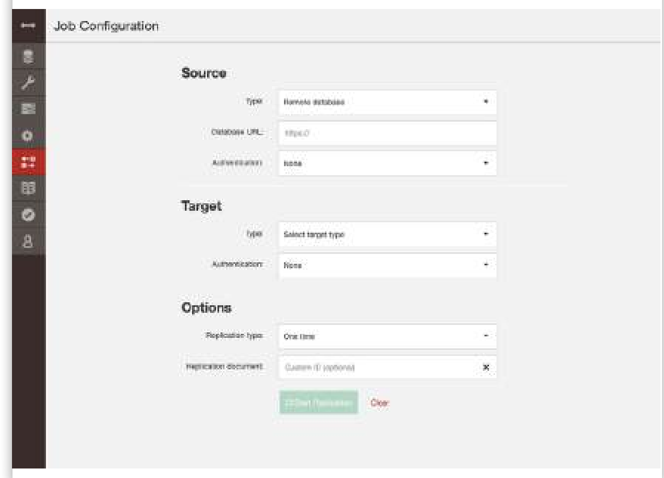
I haven't touched on it, but any URL you can hit in the browser can also be hit with a simple `curl` command. This means you can easily script things using data in your CouchDB instance, and your application will have a RESTful API interface out of the box.

Back On the Couch

We've just scratched the service of everything CouchDB can do, and there is so much more, like attaching files to a document. I know what you might think: "Fine, a schema-less database is a neat trick, but I'm not sure it's something I need." You wouldn't be wrong. At the end of the day, a database is just a database. If you are fine dealing with schemas, who am I to say you're wrong? But one relatively cool feature I skipped over was "Replication." And no, I am not talking about database clustering, although CouchDB also has that if you have a high-traffic site. And yes, I know Relational Database also have "replication," but nothing like this. Remember that term "eventual consistency" I touched on earlier? Well, hear me out.

For starters, setting up one-way or bidirectional replication between CouchDB servers is laughably easy. In the Fauxton interface we were working in earlier, there is a section for replication. You give it a source database and a target database, including the server address, tell it if it is a one-time or continuous replication, and that's it.

Figure 24.



And replication is blazing fast. We are talking seconds, not minutes. Oh, and did I mention that you can also set up this replication via `curl`, just like everything else in CouchDB? That means you can do something as simple as create a bash script that starts replication with another known server when you spin up a new server. I know you can do this because this is what I used to do. Having CouchDB run on all your servers isn't that unheard of, nor is it resource-intensive.

Regional database replication and failover are now a snap. If you want a copy in-house, just set up a replication for that. If you want one on your laptop, just set it up. It will be fine if you turn off your laptop in the afternoon or go on vacation for a week. When you get back online, the CouchDB instance will get all the new changes sent to it.

You don't even need to replicate all the databases on a server. You may have one CouchDB instance you want to replicate to a region on the other side of the world and another instance you want to replicate to your in-house sales group to run reports against. Not a problem for CouchDB.

But wait, there's one more thing I still need to mention. There's a mobile equivalent. Let me introduce you to PouchDB[6], an open-source JavaScript database inspired by Apache CouchDB. As of my writing this, PouchDB can replicate, one-way and bi-directional, with CouchDB instances. How do I know this? Because it's a very real-world example of how we use CouchDB today. Our client uses multiple mobile devices in and out of facilities, which can cause them to lose all internet connections for hours at a time. The mobile devices run an application built using the Capacitor runtime and PouchDB. When the users have access to the internet, the database is updated. If the device loses internet connectivity, the user can continue to work with the database they have locally, and when they get back online, their changes are replicated up, and any data they might have missed is replicated down. You can even add filters to replicate only a subset or a single document. It's bonkers.

Finally, Some Drawbacks

There are some drawbacks, but depending on where you stand on the topics, you might not consider them drawbacks at all.

- There is a real learning curve. It's not that the concepts are that difficult to understand, but personally, my brain just kept fighting them. I couldn't stop thinking of data as something that needed a defined schema and relationships. So, understanding how you are going to capture data is essential.
- It can be a little nerve-racking to not be able to access your data without using traditional SQL queries at first.
- It's a fast-read, slow-write database. If your database needs the same performance for reads and writes, this is probably not the solution for you.
- It doesn't support transactions in the traditional sense.
- A personal pet peeve I had to overcome was the idea of not duplicating data. In a relational database, once data is written, it should not have to be written again in another place. With CouchDB, I can find myself ignoring this rule. An example is a user who might add their addresses to their profile, and their profile is a document in CouchDB. Then, when a user orders something, you will query the user's addresses and let them select the address they want to use for the order. A common practice with CouchDB would be adding that address and other user information, like their name, to the document representing that order. I sometimes will reference the actual user's document in CouchDB as well in the order document. Still, I am replicating the information even though I already have the information in the user document. One way to look at this is that I don't need the user document for the function of the order itself. If all I have is the order document, I have all the information I need about that order in the order's document to fulfill that task.
- Data integrity was another big one for me to come to peace with. You can no longer depend on your database server to be the last line of defense against having poor data added to it. That is now in the hands of your developers, who must ensure they are validating all the data correctly. If a phone number shouldn't have a letter in it, it will be up to your developers to watch for that, catch it, and handle it correctly.

I truly encourage you to give CouchDB a look. As with most things, the right tool for the right job. I still don't have a clear threshold for when I choose CouchDB over a relational database, but it's always in play when starting a new project. A good video to check out is IBM's CouchDB Explained[7].

Eric Van Johnson is the co-CEO of PHP Architect, LLC. An organizer of San Diego PHP (SDPHP) and podcaster with php[podcast], PHPUgly, and PHPRoundtable. A husband, father, and enjoyer of scotch and baseball. You can reach him on X as `@shocm` *or on Mastodon as* `@eric@phparch.socal` *@shocm*

6 https://pouchdb.com/

7 https://www.youtube.com/watch?v=aOE90VAVOcU