



---

# So You'd Like to Migrate From a Relational Database to CouchDB?

---

By *Till Klampaeckel*

*August 2010*

(Original Blogposts were written October 2009, November 2009 and  
May 2010)

## Content

Introduction	3
Part I — First things first!	3
Part II — Basic Operations and Building a Small Wrapper for CouchDB	7
Part III — Basic View Functions in CouchDB	11

---

## Couchio

The folks here at Couchio would like to thank Till for his informative blog posts on CouchDB. We hope that providing this documentation will help others that are looking to use CouchDB. Tills's original blogposts are at <http://till.klampaeckel.de/blog/archives/74-PHP-So-you-d-like-to-migrate-from-MySQL-to-CouchDB-Part-I.html>.

This work is licensed under the Creative Commons Attribution 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

## Introduction

This white paper was created from a series of blogposts that I did discussing migrations from MySQL, or a SQL-database generally, to CouchDB. First I'll start off by introducing CouchDB — from a PHP side, then I'll demo a couple basic use cases, and discuss migrations from MySQL. Next I will focus on create, read, update and delete operations in CouchDB. I will also introduce my nifty PHP CouchDB called [ArmChair!](#) ArmChair is my own very simple and (hopefully) easy-to-use approach to accessing CouchDB from PHP. In the final section I will discuss the basic view functions in CouchDB.

My idea is to introduce CouchDB to a world where database-driven development generally refers to MySQL. By no means is this meant to be disrespectful to MySQL, or SQL-databases in general. However, I'm a firm believer in *using the right tool for the job*.

## Part I — First things first!

First off, before using CouchDB and maybe eventually replacing MySQL with it, we need to ask ourselves the “**Why?**”- question. And in order to be capable of more than a well-educated guess we need to familiarize ourselves with CouchDB basics.

### Basics

- Document-oriented and schema-less storage.
- Erlang (for rock-solid-scaling-goodness).
- RESTful HTTP-API (we'll get to that).
- Everything is JSON - request data, storage, response!

### Document-oriented

In a document-oriented as to opposed to a relational store, the data is not stored in table, where data is usually broken down into fields. In a document-oriented store each record is stored along side and can have its own characteristics — properties of any kind.

As an example, consider these two records:

```
Till Klampaeckel, Berlin  
Till Klampaeckel, till@php.net, Berlin, Germany
```

In a relational store, we would attempt to break down, or normalize, the data. Which means that we would probably create a table with the columns name, email, city and country.

Consider adding another record:

```
Till Klampaeckel, +49110, till@some.jabber  
(Just fyi — this is not my real phone number!)
```

Looking for an intersection in the records, the name is the only thing this record has in common with the previous two. With a relational database, we would either have to add a column for phone number and *chat*, or we would start splitting off the data into multiple tables (e.g. a table called phone and one called chat) in order to get grip. With a document-oriented database — such as CouchDB — this is not an issue. We can store any data, constraints do not apply.

## Erlang

Erlang was invented a while ago, by Ericsson, when it was still sans Sony. In a nutshell, Erlang's true strength is reliability and stability. It also manages to really utilize all the resources modern hardware has to offer since it's a master of parallelization. CouchDB is written in Erlang, and also accepts view code written in Erlang. More on views later.

## RESTful HTTP-API

For starters, a lot of HTTP-APIs claim to be RESTful, most of them are not. HTTP has so called request verbs (DELETE, GET, HEAD, POST, PUT among them) and a lot of APIs don't use them to the fullest extend, or rather not all. Instead, most APIs are limited GET and maybe use a little POST. An example of such an API is the Flickr API. Most of us are familiar with GET and POST already. For example, when you opened the web page to this blog entry, the browser made a GET-request. If you decide to post a comment later on — you guessed it, that's a POST-request. Aside from its basic yet powerful nature, HTTP is interesting in particular because it is a common denominator in many programming languages. Whatever you use — C#, PHP, Python, Ruby — these languages know how to *talk* HTTP. And even better — most of them ship pretty comfortable wrappers.

## JSON

JSON — It is a godsend for those of us who never liked XML. It's very lightweight, yet we are able to represent lists and objects, integers, strings — most data types you would want to use. A clear disadvantage of JSON is that it lacks validation (think DTD), and of course comments — ha, ha!

## Why, oh why?

So along with “Why?”, we should consider the following:

- Does it make sense?
- Is CouchDB (really) the better fit for my application?
- What is my #1 problem in MySQL, and how does CouchDB solve it?

And if we are still convinced to migrate all of our data, we'll need to decide on an access wrapper.

## It's all HTTP, right?

By now, everyone has heard that CouchDB has a RESTful HTTP-API. But what does that imply? It means, that we won't need to build a new extension in PHP to be able to use it. There's already either `ext/socket` or `ext/curl` — often both — in 99% of all PHP installs out there. Which means that PHP is more than ready to talk to CouchDB — right out of the box. Since I mentioned JSON before — today `ext/json` is available in most PHP installs as well. If however we happen to be one of the few unfortunates who don't have and cannot get this extension, we should use `Services_JSON` instead.

## Install it!

CouchDB installations are available for OS X, Windows, and in most Linux and Unix Distributions.

Mac OS X:

CouchDBX, the one-click CouchDB package for OS X <http://janl.github.com/couchdbx/>

Windows:

CouchDB Installer is at <http://people.apache.org/~mhammond/dist/>

Ubuntu Desktop 9.10+:

CouchDB is pre-installed on Ubuntu Desktop, and can be accessed according to the desktopcouch specification. <http://freedesktop.org/wiki/Specifications/desktopcouch/Documentation/ViewingMyData>

Ubuntu, Debian:

```
apt-get install couchdb
```

Fedora, CentOS:

```
yum install couchdb
```

FreeBSD:

```
cd /usr/ports/databases/couchdb && make install clean
```

### Raw, or comfort?

In general, my next question would be how comfortable it is to interact with CouchDB. What I mean is the following — consider this `curl` request from the shell:

```
shell% curl -X PUT http://localhost:5984/mydb/mydoc -d '{"foo":"bar"}'
```

This request creates a document with an id called `mydoc` in a database called `mydb`. Simple and straightforward. If I translate the above into PHP (using `curl`), it will look similar to the following:

```
<?php
$data = array('foo' => 'bar');
$json = json_encode($data);

$file = tmpfile();
fwrite($file, $json);
fseek($file, 0);

$ch = curl_init();

curl_setopt($ch, CURLOPT_URL, "http://localhost:5984/mydb/mydoc");
curl_setopt($ch, CURLOPT_HEADER, 0);
curl_setopt($ch, CURLOPT_PUT, true);
curl_setopt($ch, CURLOPT_INFILE, $file);
curl_setopt($ch, CURLOPT_INFILESIZE, strlen($json));

$result = curl_exec($ch);

fclose($file);
curl_close($ch);

echo $result;
?>
```

That's a lot of code, and also somewhat nasty looking, but that's because different limitations apply!

- A PUT request through PHP's `curl` extension needs a file handle to work with — there's no way to push a simple string into it.
- `ext/curl` itself is not exactly straightforward to use. (No offense, and strictly IMHO, of course!)

### Alternatives?

Of course there are more than a few ways to comfortably access CouchDB from PHP.

## HTTP\_Request2 vs. Zend\_Http\_Client

Both solutions are *full blown* PHP5 and offer an object oriented approach. My personal favorite is

[HTTP\\_Request2](#), but [Zend\\_Http\\_Client](#) is a close second. Both components are easily structured and simple to use HTTP client libraries. Both are more than capable to access CouchDB.

The equivalent to the `curl` command with `HTTP_Request2`:

```
<?php
require_once 'HTTP/Request2.php';

$data = array('foo' => 'bar');
$json = json_encode($data);

$req = new HTTP_Request2('http://localhost:5984/mydb/mydoc');
$res = $req->setMethod(HTTP_Request2::METHOD_PUT)
    ->setBody($json)
    ->send();

echo $res->getBody();
?>
```

## phpillow

`phpillow` is Kore Nordmann's CouchDB wrapper. You can get it from [his website](#), a basic example looks like this:

```
<?php
require 'phpillow/autoload.php';

$data = array('foo' => 'bar');
$json = json_encode($data);

phpillowConnection::createInstance('localhost', 5894);
phpillowConnection::setDatabase('mydb');
$db = phpillowConnection::getInstance();
$db->put(phpillowConnection::getDatabase() . 'mydoc', $json);
?>
```

`phpillow` adds another layer to what `HTTP_Request2` and `Zend_Http_Client` provide. And it really excels when the access classes (`phpillowDocument`, `phpillowView`, etc.) are used. It also allows you to define model classes (e.g. by extending `phpillowDocument`) and defining a set of rules for document data validation, document id generation, etc. Love it, or hate it, `phpillow` is probably not what you want to use for your first steps into CouchDB, but you want to check it out when you build a more complex application.

## Quo vadis, wrapper?

One of the things I noticed about CouchDB, especially when you use one of the client libraries I mentioned earlier, is that you tend to re-build it in every project. To this date, I unfortunately know of no *awesome* PHP wrapper for CouchDB which I'd recommend to everyone in every situation. Even my own approach is very much tailored to our own use case and in my opinion doesn't qualify to meet the needs of a wider audience. Which leads to a small dilemma — the danger one runs into is that whenever we attempt to create a wrapper that's flexible enough to serve a greater audience, we may end up making it more complex to work with CouchDB through the wrapper than it really has to be. But in the end — be that as it may — this is the extreme strength of CouchDB. With very little knowledge of HTTP, we are productive right away. No training and books necessary.

## Conclusion

In this section, I've highlighted the current PHP approach to CouchDB. I tried to keep things simple for now and will dive into basic usage and more examples in the next section.

## Part II — Basic Operations and Building a Small Wrapper for CouchDB

### Getting data into CouchDB

As I explained before — CouchDB is all HTTP. But for starters, we'll take a shortcut here and briefly talk about CouchDB's nifty administration frontend called Futon. Think of Futon as a *really* easy to use phpMyAdmin. And by default Futon is included in your CouchDB installation.

### Features

Futon will allow you to virtually do anything you need:

- create a database
- create a document
- create a view
- build the view
- *view* ;-) (documents, views)
- update all of the above
- delete all of the above
- replicate with other CouchDB servers

Assuming the installation completed successfully, and CouchDB runs on `127.0.0.1:5984`, the URL to your very own Futon is `http://127.0.0.1:5984/_utils/`. And it doesn't hurt to create a bookmark while you are at it.

### Why GUI?

Purists will always argue that using a GUI is slower than, for example, hacking your requests from the shell. That may be correct once you are a *level 99 CouchDB super hero*, but since we all start at level 1, Futon is a great way to interact with CouchDB to learn the basics. And besides, even *level 99 CouchDB super heroes* sometimes like to just click and see, and not type in *crazy* hacker commands to get it done. I'll encourage everyone to check it out.

### Read, write, update and delete.

Sometimes also referred to as CRUD (create, read, update, delete) — read, write, update and delete are the basics most web applications do. Since most of you have done a “*write a blog in X*”-tutorial before — and I personally am tired of writing blogs in different languages or with different backends — let's use another example. Think about a small guestbook application, it does all of the above — a fancy guest will even do update. For the sake of simplicity, I'll skip on the frontend in this example and we'll work on the backend and essentially create a small wrapper class for CouchDB.

### Operations

By now — “CouchDB is all HTTP” — should sound all familiar. So in turn, all these CRUD operations in CouchDB translate to the following HTTP request methods:

- write/create - **PUT** or **POST**
- read - **GET**
- update - **PUT** or **POST**
- delete - **DELETE**

### On Write

Whenever you supply an ID of a new document along with the document, you should use **PUT**. When you don't care about the document ID, use **POST** instead, and CouchDB will generate a unique ID for you. This unique ID **will not look like an autoincremented integer**, but [we should not cling to this concept](#) anyway. Without diving into too advanced context now, but the `auto_increment` feature in MySQL is a little flawed in general and in a distributed context especially. More on this (maybe) in a later part of this series — in the mean-time, check out Joshua Schachter's post.

### On Update

By default, CouchDB keeps a revision ID of each document. To many this is a pretty cool feature — out of the box, so to speak. But there are two very important and fundamental things to be aware of.

- CouchDB will keep previous revisions of a document around until you run `compact` on the database. Think of `compact` as a housekeeping chore. It will wipe your database clean of previous revisions and even optimize the indices (in case you had a lot of *data changing operations* in the mean time). CouchDB revisions are especially important in a distributed context (think replication — more on this later) and while it's cool to have them, they should not be used as a feature or be exposed to the user of your application.
- In case we decide to update a document, we always have to provide the previous (or current) revision of the document. This sounds strange, but the reasons are simple — in case another update gets in between all we have to do is provide the necessary interfaces and workflows in our application to alert the user and avoid a possible conflict.

### CouchDB and the HTTP Standard

CouchDB's API adheres to the above in 99.99999999% of the time. And it only breaks the pattern once. The exception to the rule is that when you bulk request multiple documents, which is strictly speaking a **GET** operation CouchDB will allow you to *post* in this case. The reason for this is that the length of a **GET** request is limited (by RFC) and if we exceeded this by requesting too many document IDs, we would hit a hard limit. To get around this, the operation is **POST** — but more on this later.

### Requirements

For the following examples we assume a working CouchDB installation and a database called *guestbook*. No admins are set up — we can read and write without permission.

For simplicity, we imagine a form with the following fields:

- author
- entry

... and in addition to those two keys that may be populated by the user we add:

- type (always: `guestbook`)
- added (a date of some kind)

... the last two are not absolutely necessary, but will come handy in future iterations of this tutorial.

Also, on the tech side, we need a webserver with PHP and `HTTP_Request2` (`pear install HTTP_Request2-alpha`) in your `include_path`. :-)

### Let's get to the Code

For this example, I invented a new wrapper for CouchDB called [ArmChair](#)! `ArmChair` is a very short (and simple) PHP class. It uses `HTTP_Request2` for all HTTP operations. `ArmChair` extends the `HTTP_Request2`

class, which is why the methods `setUri()`, `send()`, `setBody()`, etc. are all available in `$this` context. And in case I lost anyone with the above, they need to read up on [object oriented programming and PHP](#) before they continue. ;-)

The current version of `ArmChair` is 0.1.0 and **not meant for production use**. It will evolve as this series continues. Make sure to stay in the [0.1.0 tag](#) on Github so the code I use in this tutorial, matches the *release*. So together with the example `curl` requests for the most basic operations, the code from `ArmChair`.

### Creating an entry

Here's a simple POST request to add an entry in our guestbook:

```
curl -X POST http://127.0.0.1:5984/guestbook \
  -d '{"entry":"Great guestbook","author":"till","type":"guestbook",
    "added":"2009-11-02 22:22:00"}'
```

`ArmChair`'s version:

```
<?php
$armchair = new ArmChair('http://127.0.0.1:5984/guestbook');

$entry = array(
    'entry' => 'Great guestbook!',
    'author' => 'Till',
    'type' => 'guestbook',
    'date' => '2009-11-02 22:22:00',
);

$document = $armchair->addDocument($entry);
var_dump($document);
```

The relevant source code:

```
public function addDocument(array $data)
{
    if (isset($data['_id'])) {
        $id = urlencode($data['_id']);
        unset($data['_id']);
        $this->setUri($this->server . '/' . $id);
        $this->setMethod(HTTP_Request2::METHOD_PUT);
    } else {
        $this->setUri($this->server);
        $this->setMethod(HTTP_Request2::METHOD_POST);
    }
    $this->setBody(json_encode($data));

    $response = $this->send();
    return $this->parseResponse($response);
}
```

In a nutshell, when we supply an `_id` key, we get to select the ID the document is saved as (PUT). Otherwise, CouchDB will take care of it (POST). In our example, we create a document with a *random* ID.

### Reading all entries

```
curl -X GET http://127.0.0.1:5984/guestbook/_all_docs
```

(Note: `_all_docs` is not exactly suitable for production. Instead, we should create a view to retrieve all documents later. Since this is only part II of my series, I'll skip on views here and will get to it later.)

ArmChair's version:

```
<?php
$armchair = new ArmChair('http://127.0.0.1:5984/guestbook');
$documents = $armchair->get();
var_dump($documents);
```

The relevant source code:

```
public function get($id = null)
{
    if ($id === null) {
        $this->setUri($this->server . '/_all_docs');
    } else {
        $id = urlencode($id);
        $this->setUri($this->server . '/' . $id);
    }
    $this->setMethod(HTTP_Request2::METHOD_GET);
    $response = $this->send();
    return $this->parseResponse($response);
}
```

In simple words — we can use this method to retrieve all documents, as a single one. It all depends on if we supply an `$id` parameter when we call `$armchair->get()`.

### Deleting an entry

```
curl -X DELETE http://127.0.0.1:5984/guestbook/ID?rev=1-ea2f7bfea40efed2bcd89a8f17e903bb
```

ArmChair's version of delete:

```
<?php
$armchair = new ArmChair('http://127.0.0.1:5984/guestbook');
$armchair->deleteDocument('ID', '1-ea2f7bfea40efed2bcd89a8f17e903bb');
```

The source code:

```
public function deleteDocument($id, $rev)
{
    $id = trim($id);
    if (empty($id)) {
        return false;
    }
    $id = urlencode($id);
    $rev = urlencode($rev);
    $this->setUri($this->server . '/' . $id . '?rev=' . $rev);
    $this->setMethod(HTTP_Request2::METHOD_DELETE);

    $response = $this->send();
    return $this->parseResponse($response);
}
```

### Complete

Again, the complete source code is available on [Github](#).

### Disclaimer

And again. This class is not very robust (in terms of error handling, etc.), or feature complete. Of course anyone may use this as an entry to CouchDB and to get started, but it not be used in production — at least not until I say so. ;-)

### Conclusion

I encourage everyone to browse the **ArmChair** code on Github. Besides creating, reading and deleting, it supports updating documents.

The next section will focus on view creation — in order to pass on `/_all_docs` and play with the equivalents of `LIMIT` and `ORDER BY` in CouchDB.

### Recap

Part I introduced the CouchDB basics which included basic requests using PHP and cURL. Part II focused on create, read, update and delete operations in CouchDB. I also introduced my nifty PHP CouchDB wrapper called **ArmChair!** ArmChair is my own very simple and (hopefully) easy-to-use approach to accessing CouchDB from PHP. The objective is to develop it with each part of this series to make it a more comprehensive solution.

## Part III — Basic View Functions in CouchDB

Part III will target basic view functions in CouchDB — think of views as a **WHERE**-clause in MySQL. They are similar, but also not. :-)

### Map-Reduce-Thingy

If you read up on CouchDB before coming to this blog, you have probably heard of map-reduce. There, or maybe elsewhere. A lot of people attribute Google's success to map-reduce. Because they are able to process a lot of data in parallel (across multiple cores and/or machines) in *relatively* little time. I guess the PageRank in Google Search or Google Analytics are examples of where it *could* be used. In the following, I'll try to explain what map-reduce is. For people without a science degree. (And that includes me!)

### Map

Generally, map-reduce is a way to process data. It's made off two things: map and reduce. The idea is that the map-function is very robust and it allows data to be broken up into smaller pieces so it can be processed in parallel. In most cases the order data is processed in doesn't really matter. What generally counts is that it is processed at all. And since map allows us to run the processing in parallel, it's easier to scale out. (That's the secret sauce!) And when I write scale-out, I don't suggest building a cluster of 1000 servers in order to process a couple thousand documents. It's already sufficient in this case to utilize all cores in my own computer when the map task is run in parallel. In CouchDB, the result of map is a list of keys and values.

### Reduce

Reduce is called once the map-part is done. It's an optional step in terms of CouchDB — not every map requires a reduce to follow.

### Real World Example

- take a simple photo application (such as flickr) with comments
- use map to sort through the comments and *emit* the names of users who left one
- use reduce to only get unique *references* and see how many comments were left by these users

In SQL:

```
SELECT user, count(*) FROM comments GROUP BY user
```

### Why the Fuzz?

Just so people don't feel offended. Map-reduce is slightly more complicated than my example SQL-query but it's also not some secret-super-duper thing. Its strength is really parallelization which requires the ability to break data into chunks to process them. The end.

### An Example

My example is a photo service. I have two users — myself and *your* mom! ;-) We both upload pictures.

### Setup

My documents may look like the following:

```
{
  "_id" : "1",
  "type" : "photo",
  "title" : "A pretty cool photo",
  "description" : "This is a pretty f-ing cool photo",
  "user" : "till"
}

{
  "_id" : "2",
  "type" : "photo",
  "title" : "Another pretty cool photo",
  "description" : "This is just another pretty f-ing cool photo",
  "user" : "till"
}

{
  "_id" : "3",
  "type" : "photo",
  "title" : "A picture",
  "description" : "Not so cool, but still alright",
  "user" : "your mom"
}

{
  "_id" : "randomness",
  "type" : "comment",
  "photo" : "2",
  "text" : "My photo",
  "user" : "till"
}
```

### Map-Only

And here's a view `map`-function to get all my photos:

```
function(doc) {
  if (doc.type == 'photo') {
    emit(doc.user, null);
  }
}
```

Embed it in a document like this:

```
{
  "_id" : "_design/lookup",
  "views" : {
    "by_user" : {
      "map" : "function(doc) { if (doc.type == 'photo') { emit(doc.user, null); } }"
    }
  }
}
```

What does the request look like?

```
curl http://localhost:5984/photos/_design/lookup/_view/by_user?key="till"
```

And just like that, you wrote a `map`-function in CouchDB.

The equivalent in SQL:

```
SELECT * FROM photos WHERE user = 'till'
```

### Map-Reduce - Get a list of users

Map:

```
function (doc) {
  if (doc.user) {
    emit(doc.user, null);
  }
}
```

This basically gets us a list with "till", "till" and "your mom". The assumption here is that each user has uploaded at least one picture — that may be a little flawed but works great for my example. :-)

In order to *unique* the value, we use the following `reduce`:

```
function (keys, values) {
  return true;
}
```

And here's the request:

```
curl http://localhost:5984/photos/_design/lookup/_view/userlist?group=true
```

(Note: The `reduce` doesn't do much but it allows us to `group=true`.)

And here's the SQL:

```
SELECT distinct user FROM comments
```

### Number of photos by user

Let's assume you need your users and the number of photos they uploaded:

Map:

```
function (doc) {
  if (doc.type == 'photo') {
    emit(doc.user, 1);
  }
}
```

Reduce:

```
function (keys, values) {  
  return sum(values);  
}
```

Request:

```
curl http://localhost:5984/photos/_design/lookup/_view/count
```

SQL (you've seen it before):

```
SELECT user, count(*) FROM comments GROUP BY user
```

## Writing Views

Writing JSON in Futon is pretty tedious. I wish I could say I like it, but I don't. I also avoid creating views through code. The easiest — when you don't want to rely on another tool — is to write the view code in your favorite editor/IDE and then copy it into Futon. When you hit “save document” and it doesn't work, Futon<sup>H^H^H^H^H^H</sup>CouchDB will complain. ;-)

## CouchApp

In case you'd like to up the bar a little — meet [CouchApp](#)! The following guide (thanks, [Jan](#)) shows you how to install CouchApp and how to use it. The biggest advantage of using CouchApp is that you'll be able to add your views to version control and so on. Something a lot of people value. ;-) Only a minor issue is that you won't have to use Futon to fiddle with the views but instead it's semi-integrated with your favorite editor/IDE, etc..

## Setup

```
sudo easy_install -U couchapp  
mkdir project  
cd project  
couchapp init
```

## Create a View

```
couchapp generate view till-and-your-mom  
nano views/till-and-your-mom/map.js  
nano views/till-and-your-mom/reduce.js  
couchapp push . http://localhost:5984/db
```

## Conclusion

And without further ado — that was a little introduction to map-reduce. I'd love to say something like, “More complex examples next time!”, but it's really so simple because CouchDB is cool like that. For further reading, I'd recommend “[Views for SQL Jockeys](#)”.