Why CouchDB?

Comment on topic or styleApache CouchDB is one of a new breed of database management systems. This chapter explains why there's a need for new systems as well as the motivations behind building CouchDB.

Comment on topic or styleAs CouchDB developers we're naturally very excited to be using CouchDB. In this chapter we'll share with you the reasons for our enthusiasm. We'll show you how CouchDB's schema-free document model is a better fit for common applications, how the built-in query engine is a powerful way to use and process your data, and how CouchDB's design lends itself to modularization and scalability.

Relax

Comment on topic or styleIf there's one phrase to describe CouchDB it is *relax*. It is in the title of this book, it is the byline to CouchDB's official logo and when you start CouchDB you see:

```
Apache CouchDB has started. Time to relax.
```

Comment on topic or styleWhy is relaxation important? Developer productivity roughly doubled in the last five years. The chief reason for the boost are more powerful tools that are easier to use. Take Ruby on Rails as an example. It is an infinitely complex framework, but pretty easy to get going with. Rails is a success story because of the core design focus on ease of use. This is one reason why CouchDB is relaxing: learning CouchDB, understanding its core concepts should feel natural to most everybody who has been doing any work on The Web. And it is still pretty easy to explain to nontechnical people.

Comment on topic or styleGetting out of the way when creative people try to build specialized solutions is in itself a core feature and one thing that CouchDB aims to get right. We found existing tools too cumbersome to deal with during development or in production and decided to focus on making CouchDB easy, even a pleasure, to use. The Getting Started chapter and the The Core CouchDB API chapter will demonstrate the intuitive HTTP-based REST API.

Comment on topic or styleAnother area of relaxation for CouchDB users is the production setting. If you have a live running application, CouchDB again goes out of its way to avoid troubling you. Its internal architecture is fault tolerant, failures occur in a controlled environment and are dealt with gracefully. Single problems do not cascade through an entire server system but stay isolated in single requests.

Comment on topic or styleCouchDB's core concepts are simple (yet powerful) and well understood. Operations teams (if you have one, otherwise, that's you) do not have to fear random behaviour and untraceable errors. If anything should go wrong, you can pretty easily find out what the problem is, but these situations are rare.

Comment on topic or styleCouchDB is also designed to handle varying traffic gracefully. Say you have web site experiencing a sudden spike in traffic. CouchDB generally absorbs a lot of concurrent requests without falling over; instead, it takes a little more time for each request to finish, but they all get answered. When the spike is over, CouchDB will work faster again.

Comment on topic or styleThe third area of relaxation is growing and shrinking the underlying hardware of your application. This is commonly referred to as scaling. CouchDB enforces a set of limits on the programmer. On first look, CouchDB might seem inflexible at times, but some things are simply left out by design for the simple reason that if CouchDB would support them, it would allow a programmer to create applications that can't deal with scale. Where scaling, again, can mean growing up or shrinking down on hardware. We'll explore the whole matter of scaling CouchDB in *Part IV Deploying CouchDB*.

Comment on topic or styleIn a nutshell: CouchDB doesn't let you do what would get you in trouble later on. This sometimes means unlearning best-practices you might have picked up in your current or past work. The Recipes chapter contains a list of common tasks and how to solve them in CouchDB.

A Different Way to Model Your Data

Comment on topic or styleWe believe that CouchDB will drastically change the way that you build document-based applications. CouchDB combines an intuitive document storage model with a powerful query engine in a way that's so simple you'll probably be tempted to ask "Why has no one built something like this before?"

Comment on topic or style"Django may be built *for* the Web, but CouchDB is built *of* the Web. I've never seen software that so completely embraces the philosophies behind HTTP. CouchDB makes Django look old-school in the same way that Django makes ASP look outdated."

— *Jacob Kaplan-Moss, Django Developer*

Comment on topic or styleCouchDB's design borrows heavily from Web architecture and the concepts of resources, methods and representations. It augments this with powerful ways to query, map, combine and filter your data. Add fault-tolerance, extreme scalability, and incremental replication, and CouchDB defines a sweet spot for document databases.

A Better Fit for Common Applications

Comment on topic or styleWe write software to improve our lives and the lives of others. Usually this involves taking some mundane information such as contacts, invoices or receipts and manipulating it using a computer application. CouchDB is a great fit for common applications like this because it embraces the natural idea of evolving, self-contained documents as the very core of its data model.

Self-Contaned Data

Comment on topic or styleAn invoice contains all the pertinent information about a single transaction; the seller, the buyer, the date, and a list of the items or services sold. As shown in Figure 1-1, there's no abstract reference on this piece of paper that points to some other piece of paper with the seller's name and address. Accountants appreciate the simplicity of having everything in one place. And given the choice, programmers appreciate that, too.

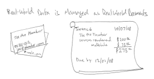


Figure 1-1: Self-contained documents

Comment on topic or styleYet this is exactly how we model our data in a relational database! Each invoice is stored in a table as a row that refers to other rows in other tables: one row for seller information, one for the buyer, one row for each item billed, and yet more rows still to describe the item details, manufacturer details and so on and so forth.

Comment on topic or styleThis isn't meant as a detraction of the relational model, which is widely applicable and extremely useful for a number of reasons. Hopefully, though, it illustrates the point that sometimes your model may not “fit” your data in the way you’d like.

Comment on topic or styleLet’s take a look at the humble contact database, to illustrate a different way of modeling data, one that more closely “fits” its real-world counterpart: a pile of business cards. Much like our invoice example, a business card contains all the important information, right there on the cardstock. We call this “self contained” data, and it’s an important concept in understanding document databases like CouchDB.

Syntax and Semantics

Comment on topic or styleMost business cards contain roughly the same information: someone’s identity, an affiliation, and some contact information. While the exact form of this information can vary between business cards, the general information being conveyed remains the same and we’re easily able to recognize it as a business card. In this sense, we can describe a business card as a *real-world document*.

Comment on topic or styleJan’s business card might contain a phone number but no fax whereas Chris’s business card contains both a phone number and fax. Jan does not have to make his lack of a fax machine explicit by writing something as ridiculous as “Fax: None” on the business card. Instead, by simply omitting a fax number it’s implied that he doesn’t have one.

Comment on topic or styleWe can see that real-world documents of the same type, such as business cards, tend to be very similar in *semantics*, the sort of information they carry, but can vary hugely in *syntax*, how that information is structured. As human beings, we’re naturally comfortable dealing with this kind of variation.

Comment on topic or styleWhile a traditional relational database requires you to model your data *up front*, CouchDB’s schema-free design unburdens you with a powerful way to aggregate your data *after the fact*, just like we do with real-world documents. We’ll look in depth at how to design applications with this underlying storage paradigm.

Building blocks for larger systems

Comment on topic or styleCouchDB is a storage system useful on its own. You can build many applications with the tools CouchDB gives you. But CouchDB is designed with a bigger picture in mind. Its components can be used as building blocks for systems larger and more complex that solve the storage problem in slightly different ways.

Comment on topic or styleWhether you need a system that’s crazy fast that isn’t too concerned with reliability (think logging) or one that guarantees storage in two or more physically separated locations for reliability, but you’re willing to take a performance hit, since reliability is more important, CouchDB lets you build these systems.

Comment on topic or styleThere are a multitude of knobs you could turn to make a system work better in one area, but you’ll affect another area while doing so. One first example would be the CAP theorem discussed in the next chapter. To give you an idea of other things that affect a storage systems see Figure 1-8 and Figure 1-9:

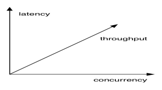


Figure 1-9: Throughput, Latency & Concurrency

Comment on topic or styleBy reducing latency for a given system (and that is true not only for storage systems), you affect concurrency and throughput capabilities.

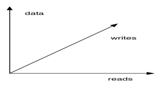


Figure 1-8: Scaling: Read Requests, Write Requests or Data

Comment on topic or styleWhen you want to scale out, there are three distinct issues to deal with: Scaling read requests, write requests and data. Orthogonal to all three and to the items shown above are many more attributes like reliability or simplicity. You can draw many of these graphs that show different features or attributes pull into different directions and thus shape the system they describe.

Comment on topic or styleCouchDB is very flexible and gives you enough building blocks to make system shaped to suit your exact problem. That's not saying that CouchDB can be bent to solve any problem: CouchDB is no silver bullet, but in the space of data storage it can get you a long way.

CouchDB Replication

Comment on topic or styleCouchDB replication is one of these building blocks. Its fundamental function is to synchronize two or more CouchDB databases. This may sound simple, but the simplicity is key to allowing replication to solve a number of problems: reliably synchronize databases between multiple machines for redundant data storage; distributing data to a cluster of CouchDB instances that share a subset of the total number of requests that hit the cluster (load balancing); distributing data between physically-apart locations, like one office in New York and another one in Tokyo.

Comment on topic or styleCouchDB replication uses the same REST API all clients use. HTTP is ubiquitous and well understood. Replication works incrementally, that is, if during replication anything goes wrong, like a dropping network connection, it will pick up where it left off the next time it runs. It also only transfers data that is needed to synchronize databases.

Comment on topic or styleA core assumption CouchDB makes is that things can go wrong, it is designed for graceful error recovery instead of assuming all will be well. The replication system's incremental design shows that best. The ideas behind "things that can go wrong" are embodied in Fallacies of Distributed Computing:

1. Comment on topic or style The network is reliable.
2. Comment on topic or style Latency is zero.
3. Comment on topic or style Bandwidth is infinite.
4. Comment on topic or style The network is secure.
5. Comment on topic or style Topology doesn't change.
6. Comment on topic or style There is one administrator.
7. Comment on topic or style Transport cost is zero.
8. Comment on topic or style The network is homogeneous.

Comment on topic or styleExisting tools often try to hide that there is a network and that any or all of the above conditions don't exist for a particular system. This usually results in fatal error scenarios when finally something goes wrong. Instead CouchDB doesn't try to hide the network, it just handles errors gracefully and lets you know when actions on your end are required.

Local Data is King

Comment on topic or styleCouchDB takes quite a few *lessons learned* from The Web, but there is one thing that sucks about the web: latency. Whenever you have to wait for an application to respond or a website to render you almost always wait for a network connection that isn't as fast as you want it at that point. Waiting a few seconds instead of milliseconds greatly influences user experience and thus user-satisfaction.

Comment on topic or styleWorse: what do you do when you are offline. This happens all the time, your DSL or cable provider has issues, your iPhone, G1 or Blackberry has *no bars*. No connectivity, no way to get to your data.

Comment on topic or styleCouchDB can solve this scenario as well and this is where scaling is important again. This time it is scaling down. Imagine CouchDB installed on phones and other mobile devices that can synchronize data with centrally hosted CouchDB's when they are on a network. The synchronization is not bound by user interface contraints like sub-second response times. It is easier to tune for high bandwidth and higher latency than for low bandwidth and very low latency. Mobile applications can then use the local CouchDB to fetch data and since no remote networking is required for that, latency is low by default.

Comment on topic or styleBut CouchDB on a phone, can it be done? Erlang, CouchDB's implementation language has been designed to run on embedded devices magnitudes smaller and less powerful than today's phones.

Wrapping Up

Comment on topic or styleThe next chapter further explores the distributed nature of CouchDB. We should have given you enough bites to whet your interest. Let's go!

Eventual Consistency

Comment on topic or styleIn the previous chapter, we saw that CouchDB's flexibility allows us to evolve our data as our applications grow and change. In this chapter we'll explore how working "with the grain" of CouchDB promotes simplicity in our applications and helps us naturally build scalable, distributed systems.

Working With The Grain

Comment on topic or styleA *distributed system* is a system which operates robustly over a wide network. A particular feature of network computing is that network links can potentially disappear and there are plenty of strategies for managing this type of network segmentation. CouchDB's differs from others by accepting eventual consistency, as opposed to putting absolute consistency ahead of raw availability, like RDBMS or Paxos. What they have in common is an awareness that data acts differently when many people are accessing it simultaneously. Their approaches differ in which aspects of *consistency*, *availability*, or *partition tolerance* they prioritize.

Comment on topic or styleEngineering distributed systems is tricky. Many of the caveats and "gotchas" you will face over time aren't immediately obvious. We don't have all the solutions and CouchDB isn't a panacea, but when you work with CouchDB's grain rather than against it, the path of least resistance leads you to naturally scalable applications.

Comment on topic or styleOf course, building a distributed system is only the beginning. A website with a database that is only available half the time is next to worthless. Unfortunately, the traditional relational database approach to consistency makes it very easy for application programmers to rely on global state, global clocks, and other high-availability no-noes, without even realizing that they're doing so. Before examining how CouchDB promotes scalability, we'll look at the constraints faced by a distributed system. After we've seen the problems that arise when parts of your application can't rely on being in constant contact with each other, we'll see that CouchDB provides an intuitive and useful way for modeling applications around high-availability.

The CAP Theorem

Comment on topic or styleThe CAP Theorem describes a few different strategies for distributing application logic across networks. CouchDB's solution uses replication to propagate application changes across participating nodes. This is a fundamentally different approach from consensus algorithms and relational databases, which operate at different intersections of consistency, availability, and partition tolerance.

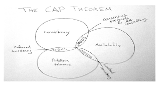


Figure 2-1: The CAP theorem

Comment on topic or styleThe CAP theorem, shown in Figure 2-1, identifies three distinct concerns:

Consistency

> Comment on topic or style All database clients see the same data, even with concurrent updates.

Availability

> Comment on topic or style All database clients are able to access some version of the data.

Partition tolerance

> Comment on topic or style The database can be split over multiple servers.

Comment on topic or stylePick two.

Comment on topic or styleWhen a system grows large enough that a single database node is unable to handle the load placed on it, a sensible solution is to add more servers. When we add nodes, we have to start thinking about how to partition data between them. Do we have a few databases that share exactly the same data? Do we put different sets of data on different database servers? Do we only let certain database servers write data and let others handle the reads?

Comment on topic or styleRegardless of which approach we take, the one problem we'll keep bumping into is that of keeping all these database servers in synchronization. If you write some information to one node, how are you going to make sure that a read request to another database server reflects this newest information? These events might be milliseconds apart. Even with a modest collection of database servers, this problem can become extremely complex.

Comment on topic or styleWhen it’s absolutely critical that all clients see a consistent view of the database, the users of one node will have to wait for any other nodes to come into agreement before being able to read or write to the database. In this instance, we see that *availability* takes a back-seat to *consistency*. However, there are situations where availability trumps consistency:

Comment on topic or style“Each node in a system should be able to make decisions purely based on local state. If you need to do something under high load with failures occurring and you need to reach agreement, you’re lost… If you’re concerned about scalability, any algorithm that forces you to run agreement will eventually become your bottleneck. Take that as a given.”

— *Werner Vogels, Amazon CTO and Vice President*

Comment on topic or styleIf availability is a priority, we can let clients write data to one node of the database without waiting for other nodes to come into agreement. If the database knows how to take care of reconciling these operations between nodes, we achieve a sort of “eventual consistency” in exchange for high-availability. This is a surprisingly applicable trade-off for many applications.

Comment on topic or styleUnlike traditional relational databases, where each action performed is necessarily subject to database-wide consistency checks, CouchDB makes it really simple to build applications that sacrifice immediate consistency in exchange for the huge performance improvements that come with simple distribution.

Local Consistency

Comment on topic or styleBefore we attempt to understand how CouchDB operates in a cluster, it’s important that we get to grips with the inner workings of a single CouchDB node. The CouchDB API is designed to provide a convenient, but thin, wrapper around the database core. By taking a closer look at the structure of the database core, we’ll have a better understanding of the API that surrounds it.

The Key to Your Data

Comment on topic or styleAt the heart of CouchDB is a powerful *B-Tree* storage engine. A B-Tree is a sorted data structure that allows for searches, insertions, and deletions in logarithmic time. As Figure 2-2 illustrates, CouchDB uses this B-Tree storage engine for all internal data, documents, and views. If we understand one, we will understand them all.

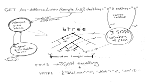


Figure 2-2: Anatomy of a View Request

Comment on topic or styleCouchDB uses MapReduce to compute the results of a view. MapReduce makes use of two functions, “map” and “reduce”, which are applied to each document in isolation. Being able to isolate these operations means that view computation lends its self to parallel and incremental computation. More importantly, because these functions produce key/value pairs, CouchDB is able to insert them into the B-Tree storage engine, sorted by key. Lookups by key, or key range, are extremely efficient operations with a B-Tree, described in *big O notation* as *O(log N)* and *O(log N + K)* respectively.

Comment on topic or styleIn CouchDB, we access documents and view results by key or key range. This is a direct mapping to the underlying operations performed on CouchDB’s B-Tree storage engine. Along with document inserts and updates, this direct mapping is the reason we describe CouchDB’s API as being a thin wrapper around the database core.

Comment on topic or styleBeing able to access results by key alone is a very important restriction, because it allows us to make huge performance gains. As well as the massive speed improvements, we can partition our data over multiple nodes, without affecting our ability to query each node in isolation. *BigTable*, *Hadoop*, *SimpleDB*, and *memcached* restrict object lookups by key for exactly these reasons.

No Locking

Comment on topic or styleA table in a relational database is a single data structure. If you want to modify a table, say to update a row, the database system must ensure that nobody else is trying to update that row and that nobody can read from that row while it is being updated. The common way to handle this uses what's known as a *lock*. If multiple clients want to access a table, the first client gets the lock, making everybody else wait. When the first client's request is processed, the next client is given access while everybody else waits, and so on. This serial execution of requests, even when they arrived in parallel, wastes a significant amount of your server's processing power. Under high load, a relational database can spend more time figuring out who is allowed to do what, and in which order, than it does doing any actual work.

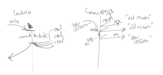


Figure 2-3: MVCC means no locking

Comment on topic or styleInstead of locks, CouchDB uses *Multi Version Concurrency Control (MVCC)* to manage concurrent access to the database. Figure 2-3 visualizes the differences between MVCC and traditional locking mechanisms. MVCC means that CouchDB can run at full speed, all the time, even under high load. Requests are run in parallel, making excellent use of every last drop of processing power your server has to offer.

Comment on topic or styleDocuments in CouchDB are versioned, much like they would be in a regular version control system such as *Subversion*. If you want to change a value in a document, you create an entire new version of that document and save it over the old one. After doing this, you end up with two versions of the same document, one old and one new.

Comment on topic or styleHow does this offer an improvement over locks? Consider a set of requests wanting to access a document. The first request reads the document. While this is being processed, a second request changes the document. Since the second request includes a completely new version of the document, CouchDB can simply append it to the database without having to wait for the read request to finish.

Comment on topic or styleWhen a third request wants to read the same document, CouchDB will point it to the new version that has just been written. During this whole process, the first request could still be reading the original version.

Comment on topic or styleA read request will always see the most recent snapshot of your database.

Validation

Comment on topic or styleAs application developers, we have to think about what sort of input we should accept and what we should reject. The expressive power to do this type of validation over complex data *within* a traditional relational database leaves a lot to be desired. Fortunately, CouchDB provides a powerful way to perform per-document validation from within the database.

Comment on topic or styleCouchDB can validate documents using JavaScript functions similar to those used for MapReduce. Each time you try to modify a document, CouchDB will pass the validation function a copy of the existing document, a copy of the new document, and a collection of additional information, such as user authentication details. The validation function now has the opportunity to approve or deny the update.

Comment on topic or styleBy working with the grain and letting CouchDB do this for us, we save ourselves a tremendous amount of CPU cycles that would otherwise have been spent serializing object graphs from SQL, converting them into domain objects and using those objects to do application level validation.

Distributed Consistency

Comment on topic or styleMaintaining consistency within a single database node is relatively easy for most databases. The real problems start to surface when you try to maintain consistency between multiple database servers. If a client makes a write operation on server *A*, how do we make sure that this is consistent with server *B*, or *C*, or *D*? For relational databases, this is a very complex problem with entire books devoted to its solution. You could use multi-master, master/slave, partitioning, sharding, write-through caches, and all sorts of other complex techniques.

Incremental Replication

Comment on topic or styleBecause CouchDB operations take place within the context of a single document, if you want to use two database nodes you no longer have to worry about them staying in constant communication. CouchDB achieves *eventual consistency* between databases by using incremental replication, a process where document changes are periodically copied between servers. We are able to build what's known as a *shared nothing* cluster of databases where each node is independent and self-sufficient, leaving no single point of contention across the system.

Comment on topic or styleNeed to scale out your CouchDB database cluster? Just throw in another server.

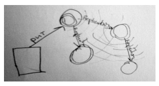


Figure 2-4: Incremental replication between CouchDB nodes
Comment on topic or styleAs illustrated in Figure 2-4, with CouchDB's incremental replication you can synchronize your data between any two databases, however you like, and whenever you like. After replication, each database is able to work independently.

Comment on topic or styleYou could use this feature to synchronize database servers within a cluster or between data centers using a job scheduler such as *cron*, or you could use it to synchronize data with your laptop for offline work as you travel. Each database can be used in the usual fashion and changes between databases can be synchronized later in both directions.

Comment on topic or styleWhat happens when you change the same document in two different databases and want to synchronize these with each other? CouchDB's replication system comes with automatic conflict detection *and* resolution. When CouchDB detects that a document has been changed in both databases, it flags this document as being in conflict, much like they would be in a regular version control system.

Comment on topic or styleThis isn't as troublesome as it might first sound. When two versions of a documents conflict during replication, the *winning* version is saved as the most recent version in the document's history. Instead of throwing the *losing* version away, as you might expect, CouchDB saves this as a previous version in the document's history, so that you can access it if you need to. This happens automatically and consistently, so both databases will make exactly the same choice.

Comment on topic or styleIt is up to you to handle conflicts in a way that makes sense for your application. You can leave the chosen document versions in place, revert to the older version, or try to merge the two versions and save the result.

Case Study

Comment on topic or styleGreg Borenstein, a friend and coworker, built a small library for converting Songbird playlists to JSON objects and decided to store these in CouchDB as part of a backup application. The completed software uses CouchDB's MVCC and document revisions to ensure that Songbird playlists are backed up robustly between nodes.

Comment on topic or styleSongbird is a free software media player with an integrated Web browser, based on the Mozilla XULRunner platform. Songbird is is available for Microsoft Windows, Apple Mac OS X, Solaris, and Linux.

Comment on topic or styleLet's examine the workflow of the Songbird backup application, first as a user backing up from a single computer, and then using Songbird to synchronize playlists between multiple computers. We'll see how document revisions turn what could have been a hairy problem into something that *just works*.

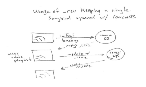


Figure 2-5: Backing up to a single database
Comment on topic or styleThe first time we use this backup application, we feed our playlists to the application and initiate a backup. Each playlist is converted to a JSON object and handed to a CouchDB database. As illustrated in Figure 2-5, CouchDB hands back the document ID and revision of each playlist as it's saved to the database.

Comment on topic or styleAfter a few days, we find that our playlists have been updated and we want to back up our changes. After we have fed our playlists to the backup application it fetches the latest versions from CouchDB, along with the corresponding document revisions. When the application hands back the new playlist document, CouchDB requires that the document revision is included in the request.

Comment on topic or styleCouchDB then makes sure that the document revision handed to it in the request matches the current revision held in the database. Because CouchDB updates the revision with every modification, if these two are out of synchronization it suggests that someone else has made changes to the document in between us requesting it from the database and sending our updates. Making changes to a document after someone else has modified it without first inspecting those changes is usually a bad idea.

Comment on topic or styleForcing clients to hand back the correct document revision is the heart of CouchDB's optimistic concurrency.

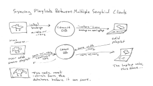


Figure 2-6: Synchronizing between two databases
Comment on topic or styleWe have a laptop we want to keep synchronized with our desktop computer. With all our playlists on our desktop the first step is to "restore from backup" onto our laptop. This is the first time we've done this so afterwards our laptop should hold an exact replica of our desktop

playlist collection.

Comment on topic or styleAfter editing our Argentine Tango playlist on our laptop to add a few new songs we've purchased we want to save our changes. The backup application replaces the playlist document in our laptop CouchDB database and a new document revision is generated. A few days later we remember our new songs and want to copy the playlist across to our desktop computer. As illustrated in Figure 2-6, the backup application copies the new document and the new revision to the desktop CouchDB database. Both CouchDB databases now have the same document revision.

Comment on topic or styleBecause CouchDB tracks document revisions it ensures that updates like these will only work if they are based on current information. If we had made modifications to the playlist backups between synchronization, things wouldn't go so smoothly.

Figure 2-7: Synchronization conflicts between two databases

Comment on topic or styleWe back up some changes on our laptop and forget to synchronize. A few days later we're editing playlists on our desktop computer, make a backup, and want to synchronize this to our laptop. As illustrated in Figure 2-7, when our backup application tries to replicate between the two databases, CouchDB sees that the changes being sent from our desktop computer are modifications of out-of-date documents and helpfully informs us that there has been a conflict.

Comment on topic or styleRecovering from this error is easy to accomplish from an application perspective. Just download CouchDB's version of the playlist and provide an opportunity to merge the changes or save local modifications into a new playlist.

Wrapping Up

Comment on topic or styleCouchDB's design borrows heavily from Web architecture and the lessons learnt deploying massively distributed systems on that architecture. By understanding why this architecture works the way it does, and by learning to spot what parts of your application can be easily distributed and what parts can not, you'll enhance your ability to design distributed and scalable applications, with CouchDB or without it.

Comment on topic or styleWe've covered the main issues surrounding CouchDB's consistency model and hinted at some of the benefits to be had when you work *with* CouchDB and not against it. But enough theory, let's get up and running and see what all the fuss is about!

Getting Started

Comment on topic or styleIn this chapter, we'll take a quick tour of CouchDB's features, familiarizing ourselves with *Futon*, the built-in administration interface. We'll create our first document and experiment with CouchDB views. Before we start, skip to the installation Appendix for your operating system. You will need to follow these instructions, and get CouchDB installed, before you can progress.

All Systems are Go!

Comment on topic or styleWe'll have a very quick look at CouchDB's barebone *Application Programming Interface (API)* by using the command line utility *curl*. Please note that this is only one way of talking to CouchDB. We will show you plenty more throughout the rest of the book. What's interesting about `curl` is that it gives you control over raw HTTP requests and you can see exactly what is going on "underneath the hood" of your database.

Comment on topic or styleMake sure CouchDB is still running and do:

```
curl http://127.0.0.1:5984/
```

Comment on topic or styleThis issues a `GET` request to your newly installed CouchDB instance.

Comment on topic or styleThe answer back should look something like:

```
{"couchdb":"Welcome","version":"0.9.0"}
```

Comment on topic or styleNot all that spectacular, CouchDB is saying "hello" with the running version number.

Comment on topic or styleThe `curl` command issues `GET` requests by default. You can issue `POST` requests using `curl -X POST`. To make it easy to work with our terminal history we usually use `-X` option even when issuing `GET` requests. If we want to send a `POST` next time, all we have to change is the verb.

Comment on topic or styleHTTP does a bit more under the hood than you can see in the examples here. If you're interested in every last detail that goes over the wire, pass in the `-v` option like `curl -vX GET` which will show you the server `curl` tries to connect to, the request headers it sends and response headers it receives back. Great for debugging!

Comment on topic or styleNext, we can get a list of databases:

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

Comment on topic or styleAll we added to the previous request is the `_all_dbs` string.

Comment on topic or styleThe response should look like:

```
[]
```

Comment on topic or styleOh, that's right, we didn't create any databases yet! All we see is an empty list.

Comment on topic or styleLet's create a database:

```
curl -X PUT http://127.0.0.1:5984/baseball
```

Comment on topic or styleCouchDB will reply with:

```
{"ok":true}
```

Comment on topic or styleRetrieving the list of databases again shows some useful results this time:

```
curl -X GET http://127.0.0.1:5984/_all_dbs

["baseball"]
```

Comment on topic or styleBefore we go on, we should mention *JavaScript Object Notation (JSON)*, the data format CouchDB speaks. JSON is a lightweight data interchange format based on JavaScript syntax. Because JSON is natively compatible with JavaScript your Web browser is an ideal client for CouchDB.

Comment on topic or styleBrackets (`[]`) represent ordered lists and curly braces (`{}`) represent key/value dictionaries. Keys must be strings, delimited by quotes (`"`), and values can be strings, numbers, booleans, lists or key/value dictionaries. For a more detailed description of JSON, see the JSON Primer appendix.

Comment on topic or styleLet's create another database:

```
curl -X PUT http://127.0.0.1:5984/baseball
```

Comment on topic or styleCouchDB will reply with:

```
{"error":"db_exists"}
```

Comment on topic or styleWe already have a database with that name so CouchDB will respond with an error. Let's try again with a different database name:

```
curl -X PUT http://127.0.0.1:5984/plankton
```

Comment on topic or styleCouchDB will reply with:

```
{"ok":true}
```

Comment on topic or styleRetrieving the list of databases yet again now shows some useful results:

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

Comment on topic or styleCouchDB will respond with:

```
["baseball", "plankton"]
```

Comment on topic or styleTo round things off, let's delete the second database:

```
curl -X DELETE http://127.0.0.1:5984/plankton
```

Comment on topic or styleCouchDB will reply with:

```
{"ok":true}
```

Comment on topic or styleThe list of databases is now the same as it was before:

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

Comment on topic or styleCouchDB will respond with:

```
["baseball"]
```

Comment on topic or styleFor brevity, we’ll skip working with documents, as the next section covers a different and potentially easier way of working with CouchDB that should provide experience with this. As we work through the example, keep in mind that under the hood everything is being done by the application exactly like you have been doing here manually. Everything is done using `GET`, `PUT`, `POST` and `DELETE` with a URI.

Welcome to Futon

Comment on topic or styleAfter having seen CouchDB’s raw API, let’s get our feet wet by playing with Futon, the built-in administration interface. Futon provides full access to all of CouchDB’s features and makes it easy getting to grips with some of the more complex ideas involved. With Futon we can create and destroy databases, view and edit documents, compose and run MapReduce views, and trigger replication between databases.

Comment on topic or styleTo load Futon in your browser, visit:

```
http://127.0.0.1:5984/_utils/
```

Comment on topic or styleIf you’re running version `0.9` or later you should see something similar to Figure 3-1. In later chapters we’ll focus on using CouchDB from server-side languages such as Ruby and Python. As such, this chapter is a great opportunity to showcase an example of natively serving up a dynamic Web application using nothing more than CouchDB’s integrated Web server, something you may wish to do with your own applications.

Figure 3-1: The Futon welcome screen

Comment on topic or styleThe first thing we should do with a fresh installation of CouchDB is run the test suite to verify that everything is working properly. This assures us that any problems we may run into aren’t due to bothersome issues with our setup. By the same token, failures in the Futon test suite are a red flag, telling us double check our installation before attempting to use a potentially broken database server, saving us the confusion when nothing seems to be working quite like we expect!

Comment on topic or styleSome common network configurations cause the replication test to fail when accessed via the `localhost` address. You can fix this by accessing CouchDB via `http://127.0.0.1:5984/_utils/`.

Comment on topic or styleNavigate to the test suite by clicking *Test Suite* on the Futon side bar, then click *run all* at the top of the main frame to kick things off. Figure 3-2 shows the Futon test suite running some tests.

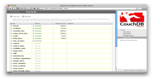


Figure 3-2: The Futon test suite running some test

Comment on topic or styleBecause the test suite is run from the browser, not only does it test that

CouchDB is functioning properly, it also verifies that your browser's connection to the database is properly configured, which can be very handy for diagnosing misbehaving proxies or other HTTP middleware.

Comment on topic or styleIf the test suite has an inordinate number of failures, you'll need to see the troubleshooting section in the Installing from Source appendix for the next steps to fix your installation.

Comment on topic or styleNow that the test suite is finished, you've verified that your CouchDB installation is successful and you're ready to see what else Futon has to offer.

Your First Database and Document

Comment on topic or styleCreating a database in Futon is simple. From the overview page click *Create Database*. When asked for a name, enter `hello-world` and click the *Create* button.

Figure 3-3: An empty database in Futon
Comment on topic or styleAfter your database has been created Futon will display a list of all its documents, as in Figure 3-3. This list will start out empty, so let's create a our first document. Click the *Create Document* link and then the *Create* button in the pop-up that gets shown. Make sure to leave the document ID blank and CouchDB will generate a UUID for you.

Comment on topic or styleFor demoing purposes, having CouchDB assign a UUID is fine. When you write your first programs, we recommend assigning your own UUIDs. If your rely on the server to generate the UUID and you end up making two POST requests because the first POST request bombed out, you might generate two docs and never find out about the first one since only the second one will be reported back. Generating your own UUIDs makes sure that you'll never end up with duplicate documents.

Comment on topic or styleFuton will display the newly created document, with its `_id` and `_rev` as the only fields. To create a new field, click the *Add Field* button. We'll call the new field `hello`. Click the green tick icon (or hit the *Enter* key) to finalize creating the `hello` field. Double-click the hello field's value (default `null`) to edit it.

Comment on topic or styleIf you try to enter `world` as the new value you'll get an error when you click the value's green tick icon. CouchDB values must be entered as valid JSON. Instead, enter `"world"` (with quotes) because this is a valid JSON string you should have no problems saving it. You could experiment with using other JSON values, for example `[1, 2, "c"]` or `{"foo":"bar"}`. Once you've entered your values into the document, make a mental note of its `_rev` attribute and click *Save Document*. The result should look something like Figure 3-4.

Figure 3-4: A "hello world" document in Futon
Comment on topic or styleYou'll notice that the document's `_rev` has changed. We'll go into more detail about this in later chapters, but for now, the important thing to note is that `_rev` acts like a safety feature when saving a document. As long as you and CouchDB agree on the most recent `_rev` of a document, you can successfully save your changes.

Comment on topic or styleFuton also provides a way to display the underlying JSON data, which can be more compact and easier to read depending on what sort of data you are dealing with. To see the JSON version of our Hello World document, click the *Source* tab. The result should look something like Figure 3-5.

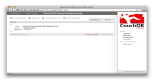


Figure 3-5: The JSON source of a “hello world” document in Futon

Running a Query Using MapReduce

Comment on topic or styleTraditional relational databases allow you to run any queries you like as long as your data is structured correctly. Instead, CouchDB uses predefined *map* and *reduce* functions in a style known as MapReduce. These functions provide great flexibility because they can adapt to variations in document structure, and indexes for each document can be computed independently and in parallel. The combination of a map- and a reduce function is called a *view* in CouchDB terminology.

Comment on topic or styleFor experienced relational database programmers, MapReduce can take some getting used to. Rather than declaring which rows from which tables to include in a result set and depending on the database to determine the most efficient way to run the query, reduce queries are based on simple range requests against the indexes generated by your map functions.

Comment on topic or styleMap functions are called once with each document as the argument. The function can choose to skip the document altogether or emit one or more view rows as key/value pairs. Map functions may not depend on any information outside of the document. This independence is what allows CouchDB views to be generated incrementally and in parallel.

Comment on topic or styleCouchDB views are stored as rows which are kept sorted by key. This makes retrieving data from a range of keys efficient even when there are thousands or millions of rows. When writing CouchDB map functions, your primary goal is to build an index that stores related data under nearby keys.

Comment on topic or styleBefore we can run an example MapReduce view, we’ll need some data to run it on. We’ll create documents carrying the price of various supermarket items as found at different stores. Let’s create documents for apples, oranges, and bananas. (Allow CouchDB to generate the `_id` and `_rev` fields.) Use Futon to create documents that have a final JSON structure that looks like this:

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "apple",
    "prices" : {
        "Fresh Mart" : 1.59,
        "Price Max" : 5.99,
        "Apples Express" : 0.79
    }
}
```

Comment on topic or styleThis document should look like Figure 3-6 when entered into Futon.

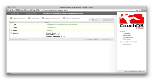


Figure 3-6: An example document with apple prices in Futon

Comment on topic or styleOkay, now that's done, let's create the document for oranges:

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "orange",
    "prices" : {
        "Fresh Mart" : 1.99,
        "Price Max" : 3.19,
        "Citrus Circus" : 1.09
    }
}
```

Comment on topic or styleAnd finally, the document for bananas:

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "banana",
    "prices" : {
        "Fresh Mart" : 1.99,
        "Price Max" : 0.79,
        "Banana Montana" : 4.22
    }
}
```

Comment on topic or styleImagine we're catering a big luncheon, but the client is very price sensitive. To find the lowest prices we're going to create our first view, which shows each fruit sorted by price. Click *hello-world* to return to the hello-world overview, and then from the *select view* menu choose *Temporary view…* to create a new view. The result should look something like Figure 3-7.

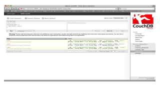


Figure 3-7: A temporary view in Futon

Comment on topic or styleEdit the map function, on the left, so that it looks like the following:

```
function(doc) {
    var store, price, value;
    if (doc.item && doc.prices) {
        for (store in doc.prices) {
            price = doc.prices[store];
            value = [doc.item, store];
            emit(price, value);
        }
    }
}
```

Comment on topic or styleThis is a *JavaScript* function that CouchDB runs for each of our documents as it computes the view. We'll leave the reduce function blank for the time being.

Comment on topic or styleClick *Run* and you should see result rows like in Figure 3-8, with the various items sorted by price. This map function could be even more useful if it grouped the items by type, so that all the prices for bananas were next to each other in the result set. CouchDB's key sorting system allows any valid JSON object as a key. To learn more, see the View Collation table on *page 225*. In this case, we'll emit an array of `[item, price]` so that CouchDB groups by item type and price.

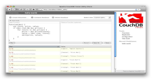


Figure 3-8: The results of running a view in Futon

Comment on topic or styleLet’s modify the view function so that it looks like this:

```
function(doc) {
    var store, price, key;
    if (doc.item && doc.prices) {
        for (store in doc.prices) {
            price = doc.prices[store];
            key = [doc.item, price];
            emit(key, store);
        }
    }
}
```

Comment on topic or styleIn this function, we first check that the document has the fields we want to use. CouchDB recovers gracefully from a few isolated map function failures, but when a map function fails regularly (due to a missing required field or other JavaScript exception), CouchDB shuts off its indexing to prevent any further resource usage. For this reason, it’s important to check for the existence of any fields before you use them. In this case, our map function will skip the first “hello world” document we created without emitting any rows or encountering an errors. The result of this query should look something like Figure 3-9.

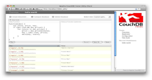


Figure 3-9: The results of running a view after grouping by item type and price

Comment on topic or styleOnce we know we’ve got a document with an item type and some prices we iterate over the item’s prices and emit key/values pairs. The key is an array of the item and the price and forms the basis for CouchDB’s sorted index. In this case the value is the name of the store where the item can be found for the listed price.

Comment on topic or styleView rows are sorted by their keys, in this example: first by item, then by price. This method of complex sorting is at the heart of creating useful indexes with CouchDB.

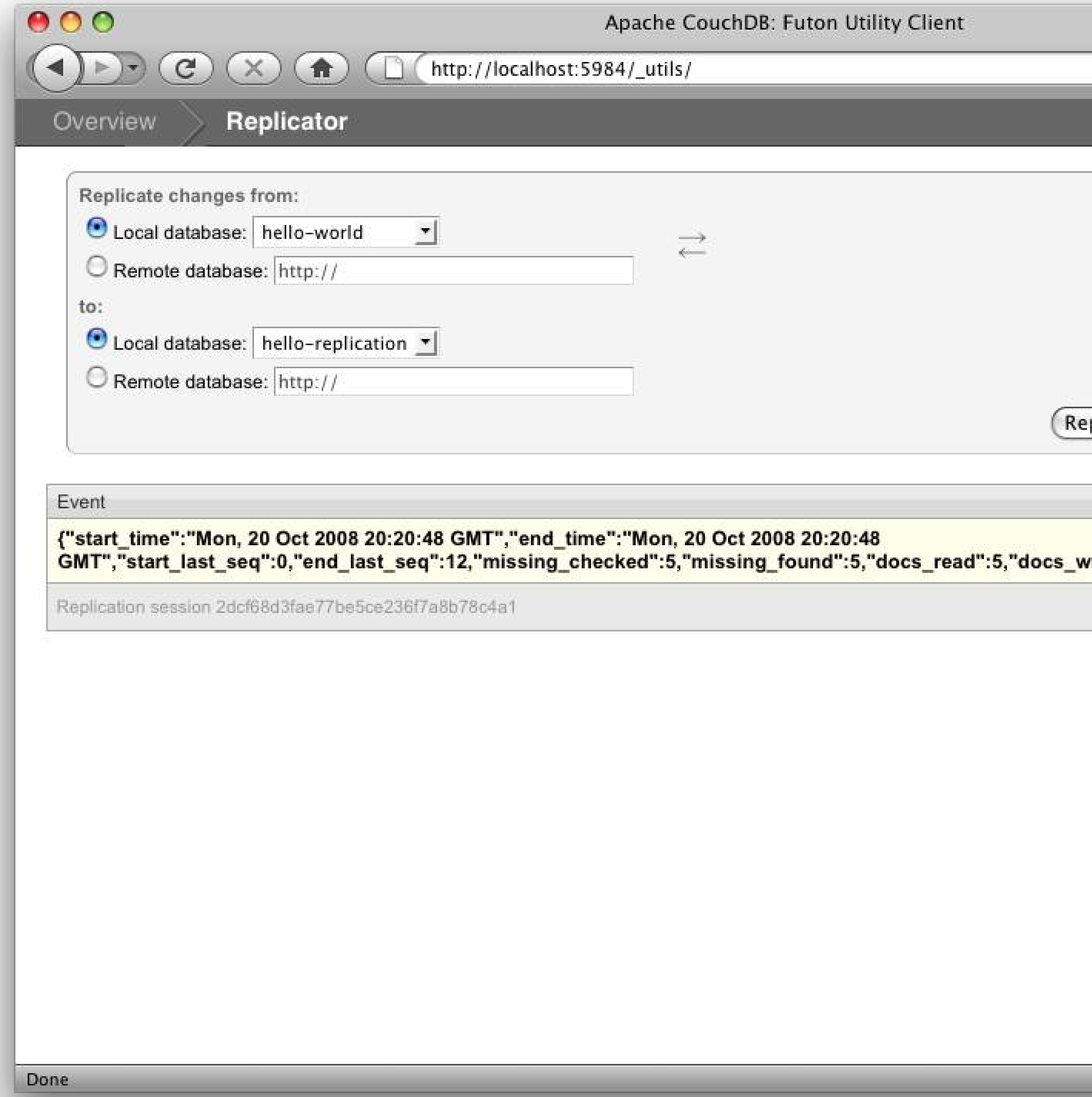
Comment on topic or styleMapReduce can be challenging, especially if you’ve spent years working with relational databases. The important thing to keep in mind is that map functions give you an opportunity to sort your data using any key you choose, and that CouchDB’s design is focused on providing fast, efficient access to data within a range of keys.

Triggering Replication

Comment on topic or styleFuton can trigger replication between two local databases, between a local and remote database, or even between two remote databases. We’ll show you how to replicate data from one local database to another, which is a simple way of making backups of your databases as we’re working through the examples.

Comment on topic or styleFirst we’ll need to create an empty database to be the target of replication. Return to the overview and create a database called `hello-replication`. Now click *Replicator* in the sidebar and choose `hello-world` as the source and `hello-replication` as the target. Click

Replicate to replicate your database. The result should look something like Figure 3-10.


Apache CouchDB: Futon Utility Client
http://localhost:5984/_utils/
Overview
Replicator
Rep
Done

Replicate changes from:
[✓ Local database: hello-world]
Remote database: http://
to:
[✓ Local database: hello-replication]
Remote database: http://

Event

```
{"start_time":"Mon, 20 Oct 2008 20:20:48 GMT","end_time":"Mon, 20 Oct 2008 20:20:48
GMT","start_last_seq":0,"end_last_seq":12,"missing_checked":5,"missing_found":5,"docs_read":5,"docs_w
```

Replication session 2dcf68d3fae77be5ce236f7a8b78c4a1

Figure 3-10: Running database replication in Futon

Comment on topic or styleFor larger databases, replication can take much longer. It is important to leave the browser window open while replication is taking place. As an alternative you can trigger replication via *curl* or some other HTTP client that can handle long-running connections. If your client closes the connection before replication finishes you'll have to re-trigger it. Luckily, CouchDB's replication can take over from where it has left off instead of starting from scratch.

Wrapping Up

Comment on topic or styleNow that we've seen most of Futon's features, you'll be prepared to dive in and inspect your data as we build up our example application in the next few chapters. Futon's pure JavaScript approach to managing CouchDB shows how it's possible to build a fully featured Web application using only CouchDB's HTTP API and integrated Web server.

Comment on topic or styleBut before we get there, we'll have another look at CouchDB's HTTP API; now with a magnifying glass. Let's *curl* on the couch and relax ahead.

The Core API

Comment on topic or styleThis chapter explores the CouchDB at minute detail. It shows all the nitty-gritty and clever bits. We show you best practices and guide you around common pitfalls.

Comment on topic or styleWe start out by revisiting the basic operations we ran in the last chapter, looking behind the scenes. We also show what Futon needs to do behind it's user interface to give us the nice features we've seen earlier.

Comment on topic or styleThis chapter is both an introduction to the core CouchDB API as well as a reference. If you can't remember how to run a particular request or why some parameters are needed you can always come back here and look things up (we are probably the heaviest user of this chapter).

Comment on topic or styleWhile explaining the API bits and pieces, we sometimes need to take a larger detour to explain the reasoning for a particular request. This is a good opportunity for us to tell you why CouchDB works the way it works.

Comment on topic or styleThe API can be subdivided in the following sections. We'll explore them individually:

- Comment on topic or style Server Info
- Comment on topic or style Databases
- Comment on topic or style Documents
- Comment on topic or style Replication

Server Info

Comment on topic or styleThis one is basic and simple. It can serve as a sanity check to see if CouchDB is running at all. It can also act as a safety guard for libraries that require a certain version of CouchDB. We're using the `curl` utility again.

```
curl http://127.0.0.1:5984/
```

Comment on topic or styleCouchDB replies, all excited to get going:

```
{"couchdb":"Welcome","version":"0.9.0"}
```

Comment on topic or styleYou get back a JSON string, that, if parsed into a native object or data structure of your programming language gives you access to the welcome string and version information.

Comment on topic or styleThis is not terribly useful, but it illustrates nicely the way of interacting with CouchDB. You send an HTTP request and you receive a JSON string in the HTTP response as a result.

Databases

Comment on topic or styleNow lets do something a little more useful: creating databases. For the strict, CouchDB is a *database management system* (DMS). That means it can hold multiple *databases*. A database is a bucket that holds "related data". We'll explore later what that means exactly. In practice, the terminology is overlapping, often people refer to a DMS as "a database" and also a database within the DMS as "a database". We might follow that slight oddity, don't get confused by it, in general, it should be clear from the context if we talk about the whole of CouchDB or a single database within CouchDB.

Comment on topic or styleNow lets make one! We want to store our favorite music albums and we creatively give our database the name `albums`. Note that we're now using the `-X` option again to tell `curl` to send a `PUT` request instead of the default `GET` request.

```
curl -X PUT http://127.0.0.1:5984/albums
```

Comment on topic or styleCouchDB replies:

```
{"ok":true}
```

Comment on topic or styleThat's it. You created a database and CouchDB told you that all went well. What happens if you try to create a database that already exists? Let's try to create that database again:

```
curl -X PUT http://127.0.0.1:5984/albums
```

Comment on topic or styleCouchDB replies:

```
{"error":"file_exists","reason":"The database could not be created, the file
already exists."}
```

Comment on topic or styleWe get back an error. This is pretty convenient. We also learn a little bit about how CouchDB works. CouchDB stores each database in a single file. Very simple. This has some consequences down the road, but we skip on details for now and explore the underlying storage system the The Power of B-Trees appendix.

Comment on topic or styleLet's create another database, this time with `curl`'s `-v` (for "verbose") option. The verbose option tells `curl` to show us not only the essentials – the HTTP response body, but all the underlying request and response details:

```
curl -vX PUT http://127.0.0.1:5984/albums-backup
```

Comment on topic or style`curl` elaborates:

```
* About to connect() to 127.0.0.1 port 5984 (#0)
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
> PUT /albums-backup HTTP/1.1
> User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l
zlib/1.2.3
> Host: 127.0.0.1:5984
> Accept: */*
>
< HTTP/1.1 201 Created
< Server: CouchDB/0.9.0 (Erlang OTP/R12B)
< Date: Sun, 05 Jul 2009 22:48:28 GMT
< Content-Type: text/plain;charset=utf-8
< Content-Length: 12
< Cache-Control: must-revalidate
<
{"ok":true}
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Comment on topic or styleWhat a mouthful. Let's step through this line by line to understand what's going on and find out what's important. Once you've seen this output a few times, you're able to spot the important bits more easily.

```
* About to connect() to 127.0.0.1 port 5984 (#0)
```

Comment on topic or styleThis is `curl` telling us that is going to establish a *TCP* connection to the CouchDB server we specified in our request URI. Not at all important, only when debugging networking issues.

```
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 (127.0.0.1) port 5984 (#0)
```

Comment on topic or style`curl` tells us it successfully connected to CouchDB. Again, not important if you don't try to find problems with your network.

Comment on topic or styleThe following lines are prefixed with `>` and `<` characters. `>` means the line was sent to CouchDB verbatim (without the actual `>`). `<` means the line was sent back to `curl` by CouchDB.

```
> PUT /albums-backup HTTP/1.1
```

Comment on topic or styleThis initiates an HTTP request. It's *method* is `PUT`, the *URI* is `/albums-backup` and the HTTP version is `HTTP/1.1`. There is also `HTTP/1.0` which is simpler in some cases, but for all practical reasons, you should be using `HTTP/1.1`.

Comment on topic or styleNext, we see a number of *request headers*. These are used to provide additional details about the request to CouchDB.

```
 > User-Agent: curl/7.16.3 (powerpc-apple-darwin9.0) libcurl/7.16.3 OpenSSL/0.9.7l
zlib/1.2.3
```

Comment on topic or styleThe `User-Agent` header tell CouchDB which piece of client software is doing the HTTP request. We don't learn anything new, it's `curl`. This header is often useful in web development when there are known errors in client implementations that a server might want to prepare the response for. It also helps to determine, which platform a user is on. This information can be used for technical and statistical reasons. For CouchDB, this header is irrelevant.

```
> Host: 127.0.0.1:5984
```

Comment on topic or styleThis header is required by HTTP 1.1, it tells the server the host name that came with the request.

```
> Accept: */*
```

Comment on topic or styleThe `Accept` header tells CouchDB that `curl` accepts any media type. We'll look into why this is useful a little later.

```
>
```

Comment on topic or styleAn empty line denotes that the request headers are finished now and the rest of the request contains data we're sending to the server. In this case, we're not sending any data, so the rest of the curl output is dedicated to the HTTP response.

```
< HTTP/1.1 201 Created
```

Comment on topic or styleThe first line of CouchDB's HTTP response includes the HTTP version information (again, to acknowledge, that the requested version could be processed). An HTTP *status code* and a *status code message*. Different requests trigger different response codes. There's a whole range of them telling the client (`curl` in our case) what effect the request had on the server. Or, if an error occurred what kind of error. RFC 2616, the HTTP 1.1 specification defines clear behaviour for response codes. CouchDB fully follows the RFC.

Comment on topic or styleThe *201 Created* status code tells the client that the resource that the request was made against was successfully created. No surprise here, but if you remember that we got an error message when we tried to create this database twice you now know that this response also included a different response code. Acting upon responses based on response codes is a common practice. For example, all response codes of 400 or larger tell you that some error occurred. If you want to shortcut your logic and immediately deal with the error, you could just check a `> 400` response code.

```
< Server: CouchDB/0.9.0 (Erlang OTP/R12B)
```

Comment on topic or styleThe `Server` header is good for diagnostics, it tells us which CouchDB version and which underlying Erlang version you are talking to. In general, you can ignore this header, but it is good to know it is there if you need it.

```
< Date: Sun, 05 Jul 2009 22:48:28 GMT
```

Comment on topic or styleThe `Date` header tells you the time of the server. Since client- and server-time are not necessary synchronized, this header is purely informational. You shouldn't build any critical application logic on top of this!

```
< Content-Type: text/plain;charset=utf-8
```

Comment on topic or styleThis header tells you which mime type the HTTP response body is and its encoding. We already know CouchDB returns JSON strings. The appropriate `Content-Type` header is `application/json`. Why do we see text/plain? This is where pragmatism wins over purity. Sending an `application/json` `Content-Type` header will make a browser offer you the returned JSON for download instead of just displaying it. Since it is extremely useful to be able to test CouchDB from a browser, CouchDB sends a `text/plain` content type, so all browser will display the JSON as text.

Comment on topic or styleThere a some browser extensions that make your browser JSON-aware, but they are not installed by default.

Comment on topic or styleDo you remember the `Accept` request header and how it is set to `\*/\*` to express interest in any mime type? If you send `Accept: application/json` in your request, CouchDB knows that you can deal with a pure JSON response with the proper `Content-Type` header and will use it instead of `text/plain`.

```
 < Content-Length: 12
```

Comment on topic or styleThis simply tells us how many bytes the response body has.

```
 < Cache-Control: must-revalidate
```

Comment on topic or styleThis tells you or any proxy server between CouchDB and you to not cache this response.

```
 <
```

Comment on topic or styleThis empty line tells us we're done with the response headers and what follows now is the response body.

```
{"ok":true}
```

Comment on topic or styleWe've seen this before.

```
* Connection #0 to host 127.0.0.1 left intact
* Closing connection #0
```

Comment on topic or styleThe last two lines are `curl` telling us that it kept the TCP connection it opened in the beginning open for a moment, but then closed it after it received the entire response.

Comment on topic or styleThroughout the book, we'll show more requests with the `-v` option, but we'll omit some of the headers we've seen here and include only those that are important for the particular request.

Comment on topic or styleCreating databases is all fine, but how do we get rid of one? Easy, just change the HTTP method.

```
> curl -vX DELETE http://127.0.0.1:5984/albums-backup
```

Comment on topic or styleThis deletes a CouchDB database. The request will remove the file that the database contents are stored in. There is no "are you sure?" safety-net or any "empty the trash" magic you've got to do in order to delete a database. Use this command with care. Your data will be deleted without chances of bringing it back easily if you don't have a backup copy of it.

Comment on topic or styleThis section went knee deep into HTTP and set the stage for discussing the rest of the core CouchDB API. Next stop: Documents.

Documents

Comment on topic or styleDocuments are CouchDB's central data structure. The idea behind a document is, unsurprisingly, that of a real-world document. A sheet of paper like an invoice, a recipe, or a business card. We already learned that CouchDB uses the JSON format to store documents. Let's see how this storing works at the lowest level.

Comment on topic or styleEach document in CouchDB has an *id*. This id is unique per database. You are free to choose any string to be the id, but for best results, we recommend a UUID (or GUID), a Universally (or Globally) Unique IDentifier. UUIDs are random numbers that have such a low collision probability that everybody can make thousands of UUIDs a minute for millions of years without ever creating a duplicate. This is a great way to ensure two independent people cannot create two different documents with the same id. Why should you care what somebody else is doing? For one, that somebody else could be you at a later time or on a different computer; secondly, CouchDB replication lets you share documents with others and using UUIDs ensures that it all works. But more on that later, lets make some documents.

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d
'{"title":"There is Nothing Left to Lose","artist":"Foo Fighters"}'
```

Comment on topic or styleCouchDB replies:

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"1-2902191555"}
```

Comment on topic or styleThe `curl` command appears complex, let's break it down. First `-X PUT` tells `curl` to make a PUT request. It is followed by the URL that specifies your CouchDB IP address and port. The resource part of the URL `/albums/6e1295ed6c29495e54cc05947f18c8af` specifies the location of a document inside our *albums* database. The wild collection of numbers and characters is a UUID. This UUID is your document's id. Finally, the `-d` flag tells curl to use the following string and use it as the body for the PUT request. The string is a simple JSON structure including `title` and `artist` attributes with their respective values.

Comment on topic or styleIf you don't have a UUID handy, you can ask CouchDB to give you one (in fact, that is what we did just now without showing you). Simply send a GET request to `/_uuids`.

```
curl -X GET http://127.0.0.1:5984/_uuids
```

Comment on topic or styleCouchDB replies:

```
{"uuids":["6e1295ed6c29495e54cc05947f18c8af"]}
```

Comment on topic or styleVoilá, a UUID. If you need more than one, you can pass in the `?count=10` HTTP parameter to request 10 UUIDs, or really, any number you need.

Comment on topic or styleTo double-check if CouchDB isn't lying about having saved your document (it usually doesn't :), try to retrieve it by sending a GET request.

```
curl -X GET http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af
```

Comment on topic or styleWe hope you see a pattern here. Everything in CouchDB has an address, a URI; and you use the different HTTP methods to operate on these URIs.

Comment on topic or styleCouchDB replies:

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"1-2902191555","title":"There is
Nothing Left to Lose","artist":"Foo Fighters"}
```

Comment on topic or styleThis looks a lot like the document you asked CouchDB to save, which is good. But you should notice that CouchDB added two fields to your JSON structure. The first is `_id` which holds the UUID we asked CouchDB to save our document under. We always know the id of a document if it is included, this is very convenient.

Comment on topic or styleThe second field is `_rev`. It stands for *revision*.

Revisions

Comment on topic or styleIf you want to change a document in CouchDB, you don't tell it to go and find a field in a specific document and insert a new value. Instead, you load the full document out of CouchDB, make your changes in the JSON structure (or object, when you are doing actual programming) and save back the entire new revision (or version) of that document back into CouchDB. Each revision is identified by a new `_rev` value.

Comment on topic or styleIf you want to update or delete a document, CouchDB expects you to include the `_rev` field of the revision you wish to change. When CouchDB accepts the change, it will generate a new revision number. This mechanism ensures that, in case somebody else made a change unbeknownst to you before you got to request the document update, CouchDB will not accept your update because you are likely to overwrite data you didn't know existed. Or simplified: Whoever saves a change to a document first, wins. Let's see what happens if we don't provide a `_rev` field (which is equivalent to providing a outdated value).

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d
'{"title":"There is Nothing Left to Lose","artist":"Foo Fighters","year":"1997"}'
```

Comment on topic or styleCouchDB replies:

```
{"error":"conflict","reason":"Document update conflict."}
```

Comment on topic or styleIf you see this, go and add the latest revision number of your document to the JSON structure:

```
curl -X PUT http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af -d
'{"_rev":"1-2902191555","title":"There is Nothing Left to Lose","artist":"Foo
Fighters","year":"1997"}'
```

Comment on topic or styleNow you see why it was handy that CouchDB returned that `_rev` when we made the initial request. CouchDB replies:

```
{"ok":true,"id":"6e1295ed6c29495e54cc05947f18c8af","rev":"2-2739352689"}
```

Comment on topic or styleCouchDB accepted your write and it also generated a new revision number. The revision number is the md5 hash of the transport representation of a document with an `N-` prefix denoting the number of times a document got updated. This is useful for replication. See the Conflict Management chapter.

Comment on topic or styleThere are multiple reasons why CouchDB uses this revision system that is also called Multi Version Concurrency Control (MVCC). They all work hand-in-hand and this is a good opportunity to explain some of them.

Comment on topic or styleOne of the aspects of the HTTP protocol that CouchDB uses is that it is *stateless*. What does that mean? When talking to CouchDB you need to *make requests*. *Making a request* includes opening a network connection to CouchDB, exchanging bytes and closing the connection. This is done every time you make a request. Other protocols allow you to open a connection, exchange bytes, keep the connection open, exchange more bytes later, maybe depending on the bytes you exchanged at the beginning, and eventually close the connection. Holding a connection open for later use requires the server to do extra work. One common pattern is that for the lifetime of a connection, the client has a consistent and static view on the data the server. Managing huge amounts of parallel connections is a significant amount of work. HTTP connections are usually short-lived and making the same guarantees is a lot easier. As a result, CouchDB can handle many more concurrent connections.

Comment on topic or styleAnother reason is that this model is simpler conceptually and as a consequence easier to program. CouchDB uses less code to make this work and less code is always good as the ratio of defects per lines of code is static.

Comment on topic or styleThe revision system also has positive effects on replication and storage mechanisms, but we'll explore them later in the book.

A Word of Warning

Comment on topic or styleThe terms *version* and *revision* might sound familiar (if you are programming without version control, drop this book right now and start learning one of the popular systems). Using new versions for document changes works a lot like version control, but there's an important difference: CouchDB does *not* guarantee that older versions are kept around.

Documents in Detail

Comment on topic or styleNow let's have a closer look at our document creation requests with the `curl -v`-flag that was helpful when we explored the database API earlier. This is also a good opportunity to create more documents that we can use in later examples.

Comment on topic or styleWe'll add some more of our favorite music albums. Get a fresh UUID from the `/_uuids` resource. If you don't remember how that works, you can look it up a few pages back.

```
curl -vX PUT http://127.0.0.1:5984/albums/70b50bfa0a4b3aed1f8aff9e92dc16a0 -d
'{"title":"Blackened Sky","artist":"Biffy Clyro","year":2002}'
```

Comment on topic or styleBy the way, if you happen to know more information about your favorite albums, do not hesitate to add more properties. And don't worry about not knowing all the info for all the albums, CouchDB's schemaless documents can contain whatever you know. After all, you should relax and not worry about data.

Comment on topic or styleNow with the `-v` option, CouchDB's reply with only the important bits looks like this:

```
> PUT /albums/70b50bfa0a4b3aed1f8aff9e92dc16a0 HTTP/1.1
>
< HTTP/1.1 201 Created
< Location: http://127.0.0.1:5984/albums/70b50bfa0a4b3aed1f8aff9e92dc16a0
< Etag: "1-2248288203"
<
{"ok":true,"id":"70b50bfa0a4b3aed1f8aff9e92dc16a0","rev":"1-2248288203"}
```

Comment on topic or styleWe're getting back the `201 Created` HTTP status code in the response headers as we've seen earlier when we created a database. The `Location` header gives us a full URL to our newly created document. And there's a new header; meet Mr. `Etag`. An Etag in HTTP-speak identifies a specific version of a resource. In this case, it identifies a specific version (the first one) of our new document. Sounds familiar? Yes, conceptually, an Etag is the same as a CouchDB document revision number and it shouldn't come as a surprise that CouchDB uses revision numbers for Etags. Etags are useful for caching infrastructures, we'll learn how to use them in *Part V Scaling CouchDB*.

Attachments

Comment on topic or styleCouchDB documents can have attachments just like an email message can have attachments. An attachment is identified by a name and includes its mime type (or content type) and the number of bytes the attachment contains. Attachments can be any data. It is easiest to think about attachments as files attached to a document. These files can be text, images, Word documents, music or movie files. Let's make one.

Comment on topic or styleAttachments get their own URL where you can upload data. Say we want to add the album artwork to the `6e1295ed6c29495e54cc05947f18c8af` document ("There is Nothing Left to Lose") and let's also say the artwork is in a file artwork.jpg in the current directory:

```
> curl -vX PUT
http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af/artwork.jpg?rev=2-
2739352689 --data-binary @artwork.jpg -H "Content-Type: image/jpg"
```

Comment on topic or styleThe `-d@` option tells `curl` to read a file's contents into the HTTP request body. We're using the `-H` option to tell CouchDB that we're uploading a JPG file. CouchDB will keep this information around and will send the appropriate header when requesting this attachment; in case of an image like this, a browser e.g. will render the image instead of offering you the data for download. This will come handy later. Note that you need to provide the current revision number of the document you're attaching the artwork to, just as if you would update the document. Because, after all, attaching some data is changing the document.

Comment on topic or styleIf you now point your browser to `http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af/artwork.jpg `, you should see your artwork image.

Comment on topic or styleIf you request the document again, you'll see a new member `_attachments`:

```
curl http://127.0.0.1:5984/albums/6e1295ed6c29495e54cc05947f18c8af
```

Comment on topic or styleCouchDB replies:

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"3-131533518","title":"There is
Nothing Left to Lose","artist":"Foo Fighters","year":"1997","_attachments":
{"artwork.jpg":{"stub":true,"content_type":"image/jpg","length":52450}}}
```

Comment on topic or style`_attachments` is a list of keys and values where the values are JSON objects containing the attachment meta data. `stub=true` tells us, that this entry is just the metadata. If we use the `?attachments=true` HTTP option when requesting this documents, we'd get a base64 encoded string containing the attachment data.

Comment on topic or styleWe'll have look at more document request options later as we explore more features of CouchDB. Like replication, the next topic.

Replication

Comment on topic or styleCouchDB replication is a mechanism to synchronize databases. Much like `rsync` (if you are familiar with that) synchronizes two directories locally or over a network, replication synchronizes two databases locally or remote.

Comment on topic or styleIn a simple POST request you tell CouchDB the *source* and the *target* of a replication and CouchDB will go ahead and figure out which documents and new document revisions are on *source* that are not yet on *target* and will proceed to move the missing documents and revisions over.

Comment on topic or styleWe'll have an in-depth look replication later in the book; in this chapter we'll just show you how to use it.

Comment on topic or styleFirst, we'll create a target database. Note that CouchDB won't automatically create a target database for you and will return a replication failure, if the target doesn't exist (likewise for the source, but that mistake isn't as easy to make :).

```
curl -X PUT http://127.0.0.1:5984/albums-replica
```

Comment on topic or styleNow we can use the database `albums-replica` as a replication target:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"albums","target":"albums-replica"}'
```

Comment on topic or styleCouchDB replies (this time we formatted the output so you read it more easily):

```
{
  "history": [
    {
      "start_last_seq": 0,
      "missing_found": 2,
      "docs_read": 2,
      "end_last_seq": 5,
      "missing_checked": 2,
      "docs_written": 2,
      "doc_write_failures": 0,
      "end_time": "Sat, 11 Jul 2009 17:36:21 GMT",
      "start_time": "Sat, 11 Jul 2009 17:36:20 GMT"
    }
  ],
  "source_last_seq": 5,
  "session_id": "924e75e914392343de89c99d29d06671",
  "ok": true
}
```

Comment on topic or styleCouchDB maintains a *session history* of replications. The response for a replication request contains the history entry for this *replication session*. It is also worth noting that the request for replication will stay *open* until replication closes. If you have a lot of documents, it'll take a while until they are all replicated and you wont get back the replication response until all documents are replicated. It is important to note that replication only replicates the database as it was at the point in time when replication was started. So, any additions, modifications, or deletions subsiquent to the start of replication will not be replicated.

Comment on topic or styleWe'll punt on the details again, the `"ok": true` at the end tells us all went well. If you now have a look at the `albums-replica` database, you should see all the documents that you created in the `albums` database. Neat, eh?

Comment on topic or styleWhat you just did is called *local replication* in CouchDB terms. You created a local copy of a database. This is useful for backups or to keep snapshots of a specific state of your data around for later. You might want to do this if you are developing your applications but want to be able to roll back to a stable version of your code and data.

Comment on topic or styleThere are more types of replication, useful in other situations. The `source` and `target` members of our replication request are actually links (like in HTML) and so far we've seen links relative to the server we're working on (hence *local*). You can also specify a remote database as the target:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"albums","target":"http://127.0.0.1:5984/albums-replica"}'
```

Comment on topic or styleUsing a local `source` and a remote `target` database is called *push replication*. We're pushing changes to a remote server.

Comment on topic or styleSince we don't have a second CouchDB server around just yet, we'll just use the absolute address of our single server, but you should be able to infer from this, that you can put any remote server in there.

Comment on topic or styleThis is great for sharing local changes with remote servers or buddies next door.

Comment on topic or styleYou can also use a remote `source` and a local `target` to do a *pull* replication. This is great for getting the latest changes from a server that is used by others.

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"http://127.0.0.1:5984/albums-replica","target":"albums"}'
```

Comment on topic or styleFinally, you can run *remote replication* which is mostly useful for management operations:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d
'{"source":"http://127.0.0.1:5984/albums","target":"http://127.0.0.1:5984/albums-
replica"}'
```

Comment on topic or styleCouchDB basks itself in having a *RESTful API* and these replication requests don't look very RESTy to the trained eye, what's up with that? While CouchDB's core database, document & attachment API is RESTful, not all of CouchDB's API is and the replication API is one example. There are more, as we'll see later in the book.

Comment on topic or styleWhy are there RESTful an non-RESTful APIs mixed up here? Have the developers been too lazy to go REST all the way? Remember, REST is an architectural style that lends itself to certain architectures (such as the the CouchDB document API). But it is not a one-size-fits-all. Triggering an event like replication does not make a whole lot of sense in the REST world. It is more like a traditional remote procedure call. And there is nothing wrong with this.

Comment on topic or styleWe very much believe in the "use the right tool for the job"-philospohy and REST does not fit every job. For support we refer to Leonard Richardson and Sam Ruby who wrote the "REST bible", er, the canonical reference and they share our view.

Wrapping Up

Comment on topic or styleThis is still not the full CouchDB API, but we discussed the essentials in great detail. We’re going to fill in the blanks as we go. For now, we believe you’re ready to start building CouchDB applications.

Design Documents

Comment on topic or style*Design documents* are a special type of CouchDB document which contains application code. Because it runs inside a database, the application API is highly structured. We’ve seen JavaScript views, and other functions in the previous chapters. In this section we’ll take a look at the function APIs, and talk about how functions in a design document are related within applications.

Document Modeling

Comment on topic or styleThere are two main kinds of documents, in my experience. The first kind is like something a word processor would save, or a user profile. With that sort of data, you want to denormalize as much as you possibly can. Basically you want to be able to load the document in one request, and get something that makes sense enough to display.

Comment on topic or styleThere is a technique for creating "virtual" documents by using views to collate data together. You could use this to store each attribute of your user profiles in a different document, but I wouldn’t recommend it. Virtual documents are useful in cases where the presented view will be created by merging the work of different authors - for instance the reference example, a blog post and it’s comments in one query. CouchDB Joins, by Christopher Lenz, covers this in more detail.

Comment on topic or styleThis "virtual document" idea takes us to the other kind of document - the event log. Use this in cases where you don’t trust user input, or need to trigger an asynchronous job. This records the user action as an event, so only minimal validation needs to occur at save-time. It’s when you load the doc for further work that you’d check for complex relational-style constraints.

Comment on topic or styleYou can treat documents as state machines, with a combination of user input and background processing managing document state. You’d use a view by state to pull out the relevant document - changing its state would move it in the view.

Comment on topic or styleThis approach is also useful for logging - combined with the batch=ok performance hint, CouchDB should make a fine log store, and reduce views are ideal for finding things like average response time or highly-active users.

The Query server

Comment on topic or styleCouchDB’s default query server (the software package that executes design document functions) is written in JavaScript, but there are views servers available for nearly any language you can imagine. Implementing a new language is a matter of handling a few JSON commands from a simple line-based program.

Comment on topic or styleIn this section, we’ll review existing functionality like map reduce views, update validation functions, and show and list transforms. We’ll also briefly describe capabilities available on CouchDB’s roadmap, like replication filters, update handlers for parsing non-JSON input, and a rewrite handler for making application URLs more palatable. Since CouchDB is an open-source project we can’t really say when each planned feature will become available, but it’s our hope that everything described here is available by the time you read this. We’ll make it clear in the text when we're talking about things that aren't yet in CouchDB trunk.

Comment on topic or styleIn this chapter we'll look at the design document, and how it fits into the CouchDB architecture for serving applications. The principles covered here apply whatever your scale…. integrated … standalone …

Applications are Documents

Comment on topic or styleCouchDB is designed to work best when there is a one-to-one correspondence between applications and design documents.

Comment on topic or styleA **design document** is a CouchDB document with an `id` that begins with `_design/`. For instance, the example blog application, Sofa, is stored in a design document with the id `_design/sofa`. Design documents are just like any other CouchDB document: they replicate along with the other documents in their database, and track edit conflicts with the `rev` parameter.

Comment on topic or styleAs we've seen, design documents are normal JSON document, denoted by the fact that their document id is prefixed with `_design/`.

Comment on topic or styleCouchDB looks for views and other application functions here. The static HTML pages of our application are served as attachments to the design document. Views and validations, however, aren't stored as attachments, rather, they are directly included in the design document's JSON body.

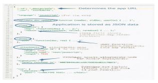


Figure: Anatomy of our Design Doc

Comment on topic or styleCouchDB's Map Reduce queries are stored in the `views` field. This is how Futon displays and allows you to edit Map Reduce queries. View indexes are stored on a per design doc basis, according to a fingerprint of the function's text contents. This means that if you edit attachments, validations, or any other non-view (or language) fields on the design doc, the views will not be regenerated. However, if you change a map or a reduce function, the view index will be deleted and a new index built for the new view functions.

Comment on topic or styleCouchDB has the capability to render responses in formats other than raw JSON. The design doc fields `show` and `list` contain functions used to transform raw JSON into HTML, XML or other Content-Types. This allows CouchDB to serve Atom feeds without any additional middleware. The `show` and `list` functions are a little like "actions" in traditional web frameworks, they run some code based on a request, and render a response. However, they differ in that they may not have side-effects. This means that they are largely restricted to handling GET requests, but it also means they can be cached by proxies like Varnish.

Comment on topic or styleBecause application logic is contained in a single document, code upgrades can be accomplished with CouchDB replication. This also opens the possibility for a single database to host multiple applications. The interface a newspaper editor needs is vastly different from what a reader desires, although the data is largely the same. They can both be hosted by the same database, in different design documents.

Comment on topic or styleA CouchDB database can contain many design documents. Example design doc ids are:

```
_design/calendar
_design/contacts
```

```
_design/blog
_design/admin
```

Comment on topic or styleIn the full CouchDB URL structure, you’d be able to GET the design doc JSON at URLs like:

```
http://localhost:5984/mydb/_design/calendar
http://localhost:5984/mydb/_design/contacts
http://localhost:5984/mydb/_design/blog
http://localhost:5984/mydb/_design/admin
```

Comment on topic or styleWe show this to note that design documents have a special case, as they are the only documents whose URLs can be used with a literal slash. We’ve done this because nobody likes to see `%2F` in their browser’s location bar. In all other cases a slash in a document id must be escaped when used in a URL. For instance the document id `movies/jaws` would appear in the url like this: `http://localhost:5984/mydb/movies%2Fjaws`

Comment on topic or styleWe’ll build the first iteration of the example application without using `show` or `list` because writing Ajax queries against the JSON API is a better way to teach CouchDB as a database. The APIs we explore in the first iteration are the same APIs you’d use to analyze log data, archive assets, or manage persistent queues.

Comment on topic or styleIn the second iteration, we’ll upgrade our example blog so that it can function with client-side JavaScript turned off. For now, sticking to Ajax queries gives more transparency into how CouchDB’s JSON / HTTP API works. JSON is a subset of JavaScript, so working with it in JavaScript keep the impedance mismatch low, while the browser’s *XMLHttpRequest (XHR)* object handles the HTTP details for us.

Comment on topic or styleCouchDB uses the `validate_doc_update` function to prevent invalid or unauthorized document updates from proceeding. We use it in the example application to ensure that blog posts can only be authored by logged in users. CouchDB’s validation functions also can’t have any side-effects, and have the opportunity to block not only end user document saves, but also replicated documents from other nodes. We’ll talk about validation in depth in the third part of the book.

Comment on topic or styleThe raw images, JavaScript, CSS, and HTML assets needed by Sofa are stored in the `_attachments` field, which is interesting in that by default it shows only the stubs, rather than the full content of the files. Attachments are available on all CouchDB documents, not just design documents, so asset-management applications have as much flexibility as they could need. If a set of resources is required for your application to run, they should be attached to the design doc. This means a new user can easily bootstrap your application on an empty database.

Comment on topic or styleThe other fields in the design doc shown above (and in the design docs we’ll be using) are put there by CouchApp as a convenience. The `signatures` field allows as to avoid updating attachments that have not changed between the disk and the database - it does this by comparing file content hashes. The `lib` field is used to hold additional JavaScript code and JSON data to be inserted at deploy time into view, show, and validation functions.

A Basic Design Document

Comment on topic or styleIn the next section we’ll get into advanced techniques for working with design documents, but before we finish here, let’s look at a very basic design document. All we’ll do is define a single view, but it should be enough to show you how design documents sit in the larger system.

Comment on topic or styleFirst, add the following text (or something like it) to a text file called `mydesign.json` using your editor:

```
{
  "_id" : "_design/example",
  "views" : {
    "foo" : {
      "map" : "function(doc){ emit(doc._id, doc._rev)}"
    }
  }
}
```

Comment on topic or styleNow use curl to `PUT` the file to CouchDB (we'll create a database first for good measure):

```
curl -X PUT http://127.0.0.1:5984/basic
curl -X PUT http://127.0.0.1:5984/basic/_design/example -d @mydesign.json
```

Comment on topic or styleFrom the second request, you should see a response like:

```
{"ok":true,"id":"_design/example","rev":"1-230141dfa7e07c3dbfef0789bf11773a"}
```

Comment on topic or styleNow we can query the view we've defined, but before we do that, we should add a few documents to the database so we have something to view. Running the following command a few times will add empty documents:

```
curl -X POST http://127.0.0.1:5984/basic -d '{}'
```

Comment on topic or styleNow to query the view:

```
curl http://127.0.0.1:5984/basic/_design/example/_view/foo
```

Comment on topic or styleThis should give you a list of all the documents in the database (except the design doc). You've created and used your first design document!

Looking to the Future

Comment on topic or styleThere are other design document functions which are being introduced at the time of this writing, `_update` and `_filter` which we aren't convering in depth here. Filter functions are covered in the Change Notifications chapter. Imagine a web service (A) that POSTs an XML-blob at a URL of your choosing, when particular events occur. Paypal's instant payment notification is one of these. With an `_update` handler you can POST these directly at CouchDB and it can parse the XML into a JSON document and save it. The same goes for CSV, multipart form, or any other format.

Comment on topic or styleThe bigger picture we're working on is like an app server, but different in one crucial regard: rather than let the developer do whatever they want (loop of a list of docids and make queries, make queries based on the results of other queries, etc) we're defining "safe" transformations, like view, show, list and update. By safe we mean that they have well known performance characteristics, and otherwise fit into CouchDB's architecture in a streamlined way.

Comment on topic or styleThe goal here is to provide a way to build standalone apps that can also be easily indexed by search engines and used via screen-readers. Hence the push for plain-old html. You can pretty much rely on JS getting executed (except when you can't). Having HTML resources means CouchDB is suitable for public-facing web apps.

Comment on topic or styleOn the horizon are a rewrite handler and a database event handler, as they seem to flesh out the application capabilities nicely. A rewrite handler would allow your application to present it's own URL space, which would make integration into existing systems a bit easier. An event handler would allow you to run asynchronous processes when the database changes, so that for instance, a document update can trigger a workflow, multi-document validation, or message queue.

Finding Your Data With Views

Comment on topic or styleViews are useful for many purposes:

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

What is a View?

Comment on topic or styleLet's go through the different use-cases. First: Extracting data that you might need for a special purpose in a specific order. For the front page we want a list of blog post titles sorted by date. We'll work with a set of example documents as we walk through how views work. These are abridged versions of the documents we used in the Design Documents chapter, but they really could be the same.

Example Documents

```
{
  "_id":"biking",
  "_rev":"AE19EBC7654",

  "title":"Biking",
  "body":"My biggest hobby is mountainbiking. The other day...",
  "date":"2009/01/30 18:04:11"
}

{
  "_id":"bought-a-cat",
  "_rev":"4A3BBEE711",

  "title":"Bought a Cat",
  "body":"I went to the the pet store earlier and brought home a little kitty...",
  "date":"2009/02/17 21:13:39"
}

{
  "_id":"hello-world",
  "_rev":"43FBA4E7AB",

  "title":"Hello World",
  "body":"Well hello and welcome to my new blog...",
  "date":"2009/01/15 15:52:20"
}
```

Comment on topic or styleThree will do for the example. Note that the documents are sorted by `"_id"`, which is how they are stored in the database. Now we define a view. The Getting Started chapter showed you how to create a view in Futon, the CouchDB administration client. If you can't remember how to do it, go back to page XY. Bear with us without an explanation while we show you some code.

A Basic Map Function

```
function(doc) {
  if(doc.date && doc.title) {
    emit(doc.date, doc.title);
  }
}
```

Comment on topic or styleThis is a *map function* and it is written in JavaScript. If you are not familiar with JavaScript but have used C or any other C-like language such as Java, PHP or C#, this should look familiar. It is a simple function definition.

Comment on topic or styleYou provide CouchDB with view functions as strings stored inside the `views` field of a design document. You don't run it yourself. Instead, when you *query your view*, CouchDB takes the source code and runs it for you on every document in the database your view was defined in. You *query your view* to retrieve the *view result*.

Comment on topic or styleAll map functions have a single parameter `doc`. This is a single document in your database. Our map function checks if our document has a `date` and a `title` attribute — luckily all of our documents have them — and then calls the built-in `emit()` function with these two attributes as arguments.

Comment on topic or styleThe `emit()` function always takes two arguments: The first is `key` and and the second is `value`. The `emit(key, value)` function creates an entry in our *view result*. One more thing, the `emit()` function can be called multiple times in the map function to create multiple entries in the view results from a single document, but we are not doing that yet.

View Results

```
| key                   | value
|----------------------------------------------
|"2009/01/15 15:52:20" | "Hello World"
|"2009/01/30 18:04:11" | "Biking"
|"2009/02/17 21:13:39" | "Bought a Cat"
```

Comment on topic or styleCouchDB takes whatever you pass into the `emit()` function and puts it into a list. Each row in that list includes our `key` and `value`. More importantly, the list is sorted by `key`, by `doc.date` in our case. The most important feature of a view result, is that it is sorted by `key`. We will come back to that over and over again to do neat things. Stay tuned.

Comment on topic or styleIf you read carefully over the last few paragraphs, one clause stands out: “when you query your view, CouchDB takes the source code and runs it for you on every document in the database”. If you have a lot of documents, that takes quite a bit of time and you might wonder if it is not horribly inefficient to do this. Yes it would be, but CouchDB is designed to avoid any extra costs: it only runs through all documents once, when you *first* query your view. If a document is changed, the map function is only run once, to recompute the keys and values for that single document.

Comment on topic or styleThe view result is stored in a B-tree, just like the structure which is responsible for holding your documents. View B-trees are stored in their own file, so that for high-performance CouchDB usage, you can keep views on their own disk. The B-tree provides very fast lookups of rows by key, as well as efficient streaming of rows in a key range. In our example, a single view can answer all questions that involve time: “Give me all the blog posts from last week” or “last month” or “this year”. Pretty neat. Read more about how CouchDB’s B-trees work in the The Power of B-Trees appendix.

Comment on topic or styleWhen we query our view, we get back a list of all documents sorted by date, each row also includes the post title so we can construct links to posts. The listing/figure above is just a graphical representation of the view result. The actual result is JSON-encoded, and contains a little more metadata.

Actual View Result

```
{
  "total_rows": 3,
  "offset": 0,
  "rows": [
    {
      "key": "2009/01/15 15:52:20",
      "id": "hello-world",
      "value": "Hello World"
    },

    {
      "key": "2009/02/17 21:13:39",
      "id": "bought-a-cat",
      "value": "Bought a Cat"
    },

    {
      "key": "2009/01/30 18:04:11",
      "id": "biking",
      "value": "Biking"
    }
  ]
}
```

Comment on topic or styleNow, we lied again, the actual result is not as nicely formatted and doesn’t include any superfluous whitespace or newlines, but this is better for you (and us!) to read and understand. And hey, where does that `"id"` member in the result rows comes from, that wasn’t there before. Well spotted again, we omitted this earlier to avoid confusion. CouchDB automatically includes the document id of the document that created the entry in the view result. We’ll use this as well, when constructing links to the blog post pages.

Efficient Lookups

Comment on topic or styleLet’s move on to the second use-case for views: “building efficient indexes to find documents by any value or structure that resides in them”. We already explained the efficient indexing but we skipped a few details. This is a good time to finish this discussion as we are looking at map functions that are a little more complex.

Comment on topic or styleFirst, back to the B-trees! We explained that the B-tree that backs the key-sorted view result is only built once, when you first query a view and all subsequent queries will just read the B-tree instead of executing the map function for all documents again. What happens though, when you change a document, or add a new one or delete one? Easy: CouchDB is smart enough to find the rows in the view result that were created by a specific document. It marks them *invalid* to have them no longer show up in view results. If the document was deleted, we’re good, the resulting B-tree reflects the state of the database. If a doc got updated, the new doc is run through the map function and the resulting new lines are inserted into the B-tree at the correct spots; new documents are handled in the same way. The Power of B-Trees appendix demonstrates that a B-tree is a very efficient data structure for our needs and the crash-only design of CouchDB databases is carried over to the view indexes as well.

Comment on topic or styleTo add one more to the efficiency discussion, usually multiple documents get updated between view queries. The mechanism explained in the previous paragraph gets applied to all changes in the database since the last time the view got query in a batch operation which makes things even faster and is generally better use of your resources.

Find One

Comment on topic or styleOn to more complex map functions. We said "find documents by any value or structure that resides in them". We already explained how to extract a value to sort a list of views by (our `date` field). The same mechanism is used for fast lookups. The URI to query to get a view’s result is `/database/_design/designdocname/_view/viewname`. This gives you a list of all rows in the view. We only have three documents so things are small, but with thousands of documents, this can get long. You can add *view parameters* to the URI to constrain the result set. To find a single document, say we know the date of a blog post would be `/blog/_design/docs/_view/by_date?key="2009/01/30 18:04:11"` to get the "Biking" blog post. Remember that you can place whatever you like in the `key` parameter to the `emit()` function. Whatever you put in there, we can now use to look up exactly — and fast.

Comment on topic or styleNote that in the case where multiple rows have the same key (perhaps we design a view where the key is the name of the post’s author), key queries can return more than one row.

Find Many

Comment on topic or styleWe talked about "getting all posts for last month" (it’s February now), this is as easy as `/blog/_design/docs/_view/by_date?startkey="2009/01/01 00:00:00"&endkey="2009/02/00 00:00:00"`. The `startkey` and and `endkey` parameters specify an inclusive range on which we can search.

Comment on topic or styleTo make things a little nicer and to prepare for a future example, we are going to change the format of our date field. Instead of a string, we are going to use an array, where individual members are part of a timestamp in decreasing significance. This sounds fancy, but it is rather easy. Instead of

```
{
  "date": "2009/01/31 00:00:00"
}
```

Comment on topic or stylewe use

```
"date": [2009, 1, 31, 0, 0, 0]
```

Comment on topic or styleOur map function does not have to change for this, but our view result looks a little different.

New View Results

```
| key                      | value
|-----------------------------------------------
|[2009, 1, 15, 15, 52, 20] | "Hello World"
|[2009, 2, 17, 21, 13, 39] | "Bought a Cat"
|[2009, 1, 30, 18,  4, 11] | "Biking"
```

Comment on topic or styleAnd our queries change to `/blog/_design/docs/_view/by_date?key=[2009, 1, 1, 0, 0, 0]` and `/blog/_design/docs/_view/by_date?key=[2009, 01, 31, 0, 0, 0]` For all you care, this is just a change in syntax, not meaning. But it shows you the power of views. Not only can you construct an index with scalar values like strings and integers, you can also use JSON structures as keys for your views. Say we tag our documents with a list of tags and want to see all tags, but we don't care for documents that have not been tagged.

A Document Snippet With Tags

```
{
  ...
  tags: ["cool", "freak", "plankton"],
  ...
}
```

A Document Snippet Without Tags

```
{
  ...
  tags: [],
  ...
}
```

A Contrived Map Function

```
function(doc) {
  if(doc.tags.length > 0) {
    for(var idx in doc.tags) {
      emit(doc.tags[idx], null);
    }
  }
}
```

Comment on topic or styleThis shows a few new things. You can have conditions on structure (`if(doc.tags.length > 0)`) instead of just values. This is also an example of how a map function call `emit()` multiple times per document. And finally, you can pass `null` instead of a value to the `value` parameter; and the same is true for the `key` parameter. We'll see in a bit how that is useful.

Reversed Results

Comment on topic or styleTo retrieve view results in reverse order, use the `descending=true` query parameter. If you are using a `startkey` parameter, you will encounter that CouchDB returns different rows or no rows at all. What's up with that?

Comment on topic or styleIt's pretty easy to understand when you see how view query options work under the hood. A view is stored in a tree structure for fast lookups. Whenever you query a view, this is how CouchDB operates:

1. Comment on topic or style Start reading at the top, or at the position that `startkey` specifies, if present.
2. Comment on topic or style Return one row at a time until the end or we hit `endkey`, if present.

Comment on topic or styleIf you specify `descending=true`, the reading direction is reversed and *not* the sort order of the rows in the view. In addition, the same two step procedure is followed.

Comment on topic or styleSay you have a view result that looks like this:

```
| key | value |
|-------------|
|  0  | "foo" |
|  1  | "bar" |
|  2  | "baz" |
|-------------|
```

Comment on topic or styleHere are potential query options: `?startkey=1&descending=true`. What will CouchDB do? See above: Jump to `startkey` which is the row with the key `1` and start reading backwards until it hits the end of the view. So the particular result woud be

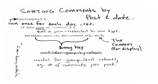
```
| key | value |
|-------------|
|  1  | "bar" |
|  0  | "foo" |
|-------------|
```

Comment on topic or styleThis is very likely not what you want. To get the rows with the indexes `1` and `2` in reverse order, you need to switch the `startkey` to `endkey`: `endkey=1&descending=true`

```
| key | value |
|-------------|
|  2  | "baz" |
|  1  | "bar" |
|-------------|
```

Comment on topic or styleNow that looks a lot better. CouchDB started reading at the bottom of the view and went backwards until it hit `endkey`.

The View to Get Comments for Posts



```
function(doc) {
  if (doc.type == "comment") {
    emit([doc.post_id, doc.created_at], doc);
  }
};
```

SORTING Comments by Post & date.
run once for each doc - req.
not a join - restricted to one type.
Array key
~~useful for group-by reduce~~
useful for group-level reduces, eg. # of comments per post.
THE COMMENT (for display)
funny notation

Figure 6-1: Comments map function

Comment on topic or styleWe use an array key here to support the `group_level` reduce query parameter. CouchDB's views are stored in the Btree file-structure (which will be described in more detail in the advanced views section). Because of the way Btree's are structured, we can cache the intermediate reduce results in the non-leaf nodes of the tree, so that reduce queries can be computed along arbitrary key ranges in logarithmic time.

Comment on topic or styleIn the blog app, we use `group_level` reduce queries to compute the count of comments both on a per-post and total basis, achieved by querying the same view index with different methods. With some array keys, and assuming each key has the value `1`:

```
["a","b","c"]
["a","b","e"]
["a","c","m"]
["b","a","c"]
["b","a","g"]
```

Comment on topic or styleThe reduce view:

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

Comment on topic or stylereturns the total number of rows between the start and end key. So with `startkey=["a","b"]&endkey=["b"]` (which includes the first 3 of the above keys) the result would equal `3`. The effect is to count rows. If you'd like to count rows without depending on the row value, you can switch on `rereduce` parameter:

```
function(keys, values, rereduce) {
  if (rereduce) {
    return sum(values);
  } else {
    return values.length;
  }
}
```

Comment on topic or styleThis is the reduce view used by the example app to count comments, while utilizing the map to output the comments, which are more useful than just `1` over and over. It pays to spend some time playing around with Map and Reduce functions. Futon is alright for this, but doesn't give full access to all the query parameters. Writing your own test code for views in your language of choice is a great way to explore the nuances and capabilities of CouchDB's incremental Map Reduce system.

Comment on topic or styleAnyway… with a `group_level` query you're basically running a series of reduce range queries. One for each *group* that shows up at the *level* you query. Let's reprint the key list from above, grouped at level `1`:

```
["a"]   3
["b"]   2
```

Comment on topic or styleAnd at `group_level=2`:

```
["a","b"]   2
["a","c"]   1
["b","a"]   2
```

Comment on topic or styleUsing the parameter `group=true` behaves as though it were `group_level=Exact`, so in the case of our current example, it would give the number 1 for each key, as there are no exactly duplicated keys.

Comment on topic or styleSetup comment view query code in `post.html`

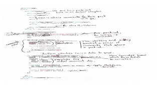


Figure 6-2: Comment display Javascript

Reduce / Rereduce

Comment on topic or styleWe briefly talked about the `rereduce` parameter to your reduce function earlier. We'll explain what's up with it in this section. By know you should have learned that your view result is stored in b-tree index structure for efficiency. The existence and use of the `rereduce` parameter is tightly coupled to how the b-tree index works.

Comment on topic or styleConsider this map result:

Example View Result (mmmh, food)

```
"afrikan", 1
"afrikan", 1
"chinese", 1
"chinese", 1
"chinese", 1
"chinese", 1
"french", 1
"italian", 1
"italian", 1
"spanish", 1
"vietnamese", 1
"vietnamese", 1
```

Comment on topic or styleWhen we want to find out how many dishes are there per origin, we can re-use the simple reduce function from above:

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

Comment on topic or styleThe following image shows a simplified version of what the b-tree index looks like. We abbreviated the key strings.

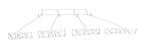


Figure 6-3: The B-Tree Index

Comment on topic or styleThe view result is what CS grads call an "pre-order" walk through the tree. We look at each element in each node starting from the right. Whenever we see that there is a sub-node to descend into, we descend and start reading the elements in that sub-node. When we walked through the entire tree, we’re done.

Comment on topic or styleYou can see that CouchDB stores both keys and values inside each leaf node. In our case it is simply always `1`, but you might have a value where you count other results and then all rows have a different value. What’s important is that CouchDB runs all elements that are within a node into the reduce function (setting the `rereduce` parameter to `false`) and stores the result inside the parent node along with the edge to the sub-node. In our case, each edge has a `3` representing the reduce value for the node it points to.

Comment on topic or styleIn reality nodes have a little over 1600 elements in them. CouchDB computes the result for all the elements in multiple iterations over the elements in a single node, not all at once (which would be disastrous for memory consumption).

Comment on topic or styleNow lets see what happens when we run a query. We want to know how many `"chinese"` entries we have. The query option is simple: `?key="chinese"`.

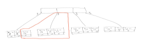


Figure 6-4: The B-Tree Index Reduce Result
Comment on topic or styleCouchDB detects that all values in the on sub-node include the `"chinese"` key. It concludes that it can just take the `3` value associated with that node to compute the final result. It then finds the node left to it and sees that it’s a node with keys outside the requested range (`key=` requests a range where the beginning and the end are the same value). It concludes that it has to use the "chinese"-element’s value and the other node’s value and run them through the reduce function with the `rereduce` parameter set to `true`.

Comment on topic or styleThe reduce function effectively calculates `3 + 1` on query time and returns the desired result. Here is some pseudocode that show the last invocation of the reduce function with actual values:

The Result is 4

```
function(null, [3, 1], true) {
  return sum([3, 1]);
}
```

Comment on topic or styleNow we said your reduce function must actually reduce your values. If you see the b-tree it should become obvious what happens when you don’t reduce your values. Consider the following map result and reduce function. This time we want to get a list of all the unique labels in our view.

```
"abc", "afrikan"
"cef", "afrikan"
"fhi", "chinese"
"hkl", "chinese"
"ino", "chinese"
"lqr", "chinese"
"mtu", "french"
"owx", "italian"
"qza", "italian"
"tdx", "spanish"
"xfg", "vietnamese"
"zul", "vietnamese"
```

Comment on topic or styleWe don’t care for the key here and only list all the labels we have. Our reduce function removes duplicates:

Don’t use this, it’s an example broken on purpose

```
function(keys, values, rereduce) {
  var unique_labels = {};
  values.forEach(function(label) {
    if(!unique_labels[label]) {
      unique_labels[label] = true;
    }
  });

  return unique_labels;
}
```

Comment on topic or styleLet’s translate this to our b-tree diagram:

Figure 6-5: An Overflowing Reduce Index

Comment on topic or styleWe hope you get the picture. The way the b-tree storage works means that if you don’t actually reduce your data in the reduce function, you end up having CouchDB to copy huge amounts of data around that grow linearly, if not faster with the number of rows in your view.

Comment on topic or styleCouchDB will be able to compute the final result, but only for views with a few rows. Anything larger will experience a ridiculously slow view build time. To help with that, CouchDB since version 0.10.0 will throw an error if your reduce function does not reduce it’s input values.

Comment on topic or styleSee the <viewsforsqljockeys,Views for SQL Jockeys Chapter> for an example of how to compute unique lists with views.

Lessons learned

- Comment on topic or style If you don’t use they key field in the map function you are probably doing it wrong.
- Comment on topic or style If you are trying to make a list of values unique in the reduce functions, you are probably doing it wrong.
- Comment on topic or style If you don’t reduce your values to a single scalar value or a small fixed-sized object or array with a fixed number of scalar values of small sizes, you are probably doing it wrong.

Summary

Comment on topic or style*Map functions* are side-effect-free functions which take a document as argument and emit key/value pairs. CouchDB stores the *emitted rows* by constructing a sorted B-Tree index, so row lookups by key, as well as streaming operations across a range of rows, can be accomplished in a small memory and processing footprint, while writes avoid seeks. Generating a view takes `O(N)`, where `N` is the total number of rows in the view. However, querying a view is very quick, as the Btree remains shallow even when it contains many many keys.

Comment on topic or style*Reduce functions* operate on the sorted rows emitted by map view functions. CouchDB’s reduce functionality takes advantage of one of the fundamental properties of B-tree indexes: for every leaf node (a sorted row), there is a chain of internal nodes reaching back to the root. Each leaf node in the B-tree carries a few rows (on the order of tens, depending on row size), and each internal node may link to a few leaf nodes or other internal nodes.

Comment on topic or styleThe reduce function is run on every node in the tree, in order to calculate the final reduce value. The end result is a reduce function which can be incrementally updated upon changes to the map function, while recalculating the reduction values for a minimum number of nodes. The initial reduction is calculated once per each node (inner and leaf) in the tree.

Comment on topic or styleWhen run on leaf nodes (which contain actual map rows), the reduce function’s third parameter, `rereduce`, is false. The arguments in this case are the keys and values as output by the map function. The function has a single returned reduction value, which is stored on the inner node that working set of leaf nodes has in common, and used as a cache in future reduce calculations.

Comment on topic or styleWhen the reduce function is run on inner nodes, the rereduce flag is true. This allows the function to account for the fact that it will be receiving its own prior output. When rereduce is true, the values passed to the function are intermediate reduction values as cached from previous calculations. When the tree is more than 2 levels deep, the rereduce phase is repeated, consuming chunks of the previous level’s output until the final reduce value is calculated at the root node.

Comment on topic or styleA common mistake new CouchDB users make, is attempting to construct complex aggregate values with a reduce function. Full reductions should result in a scalar value, like `5`, not, for instance, a JSON hash with the set of unique keys, and the count of each. The problem with this approach is that you’ll end up with a very very large final value. The number of unique keys can be nearly as large as the number of total keys, even for a large set. It is fine to combine a few scalar calculations into one reduce function, for instance to find the total, average, and standard deviation of a set of numbers in a single function.

Comment on topic or styleIf you’re interested in pushing the edge of CouchDB’s incremental reduce functionality, have a look at Google’s Sawzall paper, which gives examples of some of the more exotic reductions that can be accomplished in a system with similar constraints.

Validation Functions

Comment on topic or styleIn this chapter we’ll look closely at the individual components of Sofa’s validation function. Sofa has the basic set of validation features you’ll want in your apps, so understanding it’s validation function will give us a good foundation for others we may write in the future.

Comment on topic or styleCouchDB uses the `validate_doc_update` function to prevent invalid or unauthorized document updates from proceeding. We use it in the example application to ensure that blog posts can only be authored by logged in users. CouchDB’s validation functions — like map and reduce functions — can’t have any side-effects; they run in isolation of a request. They have the opportunity to block not only end user document saves, but also replicated documents from other CouchDBs.

Document Validation Functions

Comment on topic or styleTo ensure that users may only save documents which provide these fields, we can validate their input by adding another member to the `\_design/` document, the `validate_doc_update` function. This is the first time you've seen CouchDB's external process in action. CouchDB sends functions and documents to a JavaScript interpreter. This mechanism is what allows us to write our document validation functions in JavaScript. The `validate_doc_update` function gets executed for each document you want to create or update. If the validation function raises an exception the update is denied, when it doesn't, the updates are accepted.

Comment on topic or styleDocument validation is optional. If you don't create a validation function, no checking is done and documents with any content or structure can be written into your CouchDB database. If you have multiple design documents each with a `validate_doc_update` function, all of those functions are called upon each incoming write request. Only if all pass does the write succeed. The order of the validation execution is not defined. Each validation function must act on its own.

Figure: The JavaScript Document Validation Function

Comment on topic or styleValidation functions can cancel document updates by throwing errors. To throw an error in such a way that the user will be asked to authenticate, before retrying the request, use JavaScript code like:

```
throw({unauthorized : message});
```

Comment on topic or styleWhen you're trying to prevent an authorized user from saving invalid data, use this:

```
throw({forbidden : message});
```

Comment on topic or styleThis function throws `forbidden` errors when a post does not contain the necessary fields. In places it uses a `validate()` helper to clean up the JavaScript. We also use simple JavaScript conditionals to ensure that the `doc._id` is set to be the same as `doc.slug` for the sake of pretty urls.

Comment on topic or styleIf no exceptions are thrown, CouchDB expects the incoming document to be valid and will write it to the database. By using JavaScript to validate JSON documents, we can deal with any structure a document might have. Given that you can just make up document structure as you go, being able to validate what you come up with is pretty flexible and powerful. Validation can also be a valuable form of documentation.

Validation's Context

Comment on topic or styleBefore we delve into the details of our validation function, let's talk about the context in which they run, and the effects they can have.

Comment on topic or styleValidation functions are stored in *design documents* under the `validate_doc_update` field. There is only one per design document, but there can be many design documents in a database. In order for a document to be saved, it must pass validations on all design documents in the database (the order in which multiple validations are executed is left undefined). In this chapter we'll assume you are working in a database with only one validation function.

Writing One

Comment on topic or styleThe function declaration is simple, it takes three arguments: the proposed document update, the current version of the document on disk, and an object corresponding to the user initiating the request.

```
function(newDoc, oldDoc, userCtx) {}
```

Comment on topic or styleAbove is the simplest possible validation function, which when deployed, would allow all updates regardless of content or user roles. The converse, which never lets anyone do anything, looks like this.

```
function(newDoc, oldDoc, userCtx) {
  throw({forbidden : 'no way'});
}
```

Comment on topic or styleNote that if you install this function in your database you won't be able to perform any other document operations until you remove it from the design document or delete the design document. Admins can create and delete design documents despite the existence of this extreme validation function.

Comment on topic or styleWe can see from these examples that the return value of the function is ignored. Validations functions prevent document updates by raising errors. When the validation function passes without raising errors, the update is allowed to proceed.

Type

Comment on topic or styleThe most basic use of validation functions is to ensure that documents are properly formed to fit your application's expectations. Without validation, you need to check for the existence of all fields on a document that your map reduce or user-interface code needs to function. With validation you know that any saved documents meet whatever criteria you require.

Comment on topic or styleA common pattern in most languages, frameworks, and databases is using types to distinguish between subsets of your data. For instance, in Sofa we have a few document types, most prominently `post` and `comment`.

Comment on topic or styleCouchDB itself has no notion of types, but they are a convenient shorthand for use in your application code, including map reduce views, display logic, and user interface code. The convention is to use a field called `type` to store document types, but many frameworks use other fields as CouchDB itself doesn't care which field you use. (For instance the CouchRest Ruby client uses `couchrest-type`)

Comment on topic or styleHere's an example validation function that runs only on posts:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.type == "post") {
    // validation logic goes here
  }
}
```

Comment on topic or styleSince CouchDB stores only one validation function per design document, you'll end up validating multiple types in one function, so the overall structure becomes something like:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.type == "post") {
    // validation logic for posts
  }
  if (newDoc.type == "comment") {
    // validation logic for comments
  }
  if (newDoc.type == "unicorn") {
    // validation logic for unicorns
  }
}
```

Comment on topic or styleIt bears repeating that `type` is a completely optional field. We present it here as a helpful technique for managing validations in CouchDB, but there are other ways to write validation functions. Here's an example that uses *duck typing* instead of an explicit type attribute:

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.title && newDoc.body) {
    // validate that the document has an author
  }
}
```

Comment on topic or styleThis validation function ignores the type attribute altogether, and instead makes the somewhat simpler requirement that any document with both a title and a body must have an author. For some applications typeless validations are simpler. For others it can be a pain to keep track of which sets of fields are dependent on one another.

Comment on topic or styleIn practice many applications end up using a mix of typed and untyped validations. For instance Sofa uses document types to track which fields are required on a given document, but also uses duck typing to validate the structure of particular named fields. Eg: We don't care what sort of document we're validating, if the document has a `created_at` field we ensure that the field is a properly formed timestamp. Similarly when we validate the author of a document, we don't care what type of document it is, we just ensure that the author matches the user who saved the document.

Required Fields

Comment on topic or styleThe most fundamental validation is ensuring that particular fields are available on a document. The proper use of required fields can make writing map reduce views much simpler, as you don't have to test for all the properties before using them — you know all documents will be well-formed.

Comment on topic or styleRequired fields also make display logic much simpler. Nothing says amateur like the word `undefined` showing up throughout your application. If you know for certain that all documents will have a field, you can avoid lengthy conditional statements to render the display differently depending on document structure.

Comment on topic or styleSofa requires a different set of fields on post and comments. Here's a subset of the Sofa validation function:

```
function(newDoc, oldDoc, userCtx) {
  function require(field, message) {
    message = message || "Document must have a " + field;
    if (!newDoc[field]) throw({forbidden : message});
  };

  if (newDoc.type == "post") {
    require("title");
    require("created_at");
    require("body");
    require("author");
  }
  if (newDoc.type == "comment") {
    require("name");
    require("created_at");
    require("comment", "You may not leave an empty comment");
  }
}
```

Comment on topic or styleThis is our first look at actual validation logic. You can see that the actual error throwing code has been wrapped in a helper function. Helpers like the `require` function listed above go a long way toward making your code clean and readable. The `require` function is simple, it takes a field name and an optional message, and ensures that the field is not empty or blank.

Comment on topic or styleOnce we've declared our helper function we can simply use it in a type-specific way. Posts require a `title`, a `timestamp`, a `body` and an `author`. Comments require a `name`, a `timestamp`, and the `comment` itself. If we wanted to require that every single document contained a `created_at` field we could move that declaration outside of any type conditional logic.

Timestamps

Comment on topic or styleTimestamps are an interesting problem in validation functions. Because validation functions are run at replication time as well as during normal client access, we can't require that timestamps be set close to the server's system time. We can require two things: that timestamps do not change after they are initially set, and that they are well formed. What it means to be well-formed depends on your application. We'll look at Sofa's particular requirements here, as well as digress a bit about other options for timestamp formats.

Comment on topic or styleFirst, let's look at a validation helper that does not allow fields, once set, to be changed on subsequent updates.

```
function(newDoc, oldDoc, userCtx) {
  function unchanged(field) {
    if (oldDoc && toJSON(oldDoc[field]) != toJSON(newDoc[field]))
      throw({forbidden : "Field can't be changed: " + field});
  }
  unchanged("created_at");
}
```

Comment on topic or styleThe `unchanged` helper is a little more complex than the `require` helper, but not much. The first line of the function prevents it from running on initial updates. The `unchanged` helper doesn't care at all what goes into a field the first time it is saved. However, if there exists an already-saved version of the document, the `unchanged` helper requires that whatever fields it is used on, are the same between the new and the old version of the document.

Comment on topic or styleJavaScript's equality test is not well-suited to working with deeply nested objects. We use CouchDB's JavaScript runtime's built-in `toJSON` function in our equality test, which is better than testing for raw equality. Here’s why:

```
js> [] == []
false
```

Comment on topic or styleJavaScript considers these arrays to be different because it doesn’t look at the contents of the array when making the decision. Since they are distinct objects, JavaScript must consider them not equal. We use the `toJSON` function to convert objects to a string representation, which makes comparisons more likely to succeed in the case where two object have the same contents. This is not guaranteed to work for deeply nested objects, as `toJSON` may serialize objects

The `js` command gets installed when you install CouchDB’s spidermonkey dependency. It is a command line application that lets you parse, evaluate and run JavaScript code. `js` lets you quickly test JavaScript code snippets like the one above. You can also run a syntax check of your JavaScript code using `js file.js`. In case CouchDB’s error messages are not helpful, you can resort to testing your code standalone and get a useful error report.

Authorship

Comment on topic or styleAuthorship is an interesting question in distributed systems. In some environments you can trust the server to ascribe authorship to a document. Currently CouchDB has a simple built-in validation system which manages *node admins*. There are plans to add a database admin role, as well as other roles. The authentication system is pluggable, so you can integrate with existing services to authenticate users to CouchDB using an http layer, LDAP integration, or through other means.

Comment on topic or styleSofa uses the builtin node admin account system and so is best suited for single or small groups of authors. Extending Sofa to store author credentials in CouchDB itself is an exercize left to the reader.

Comment on topic or styleSofa’s validation logic says that documents saved with an author field must be saved by the author listed on that field.

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.author) {
    enforce(newDoc.author == userCtx.name,
      "You may only update documents with author " + userCtx.name);
  }
}
```

Wrapping Up

Comment on topic or styleValidation functions are a powerful tool to ensure only documents you expect end up in your databases. You can test writes to your database by content, by structure and by user who is making the document request. Together, these three angles let you build sophisticated validation routines that will stop anyone from tampering with your database.

Comment on topic or styleOf course validation functions are no substitute for a full security system, although they go a long way and work well with CouchDB’s other security mechanisms. Read more about CouchDB’s security in the Security chapter.

Show Functions

Comment on topic or styleCouchDB’s JSON documents are great for programatic access in most environments. Almost all languages have HTTP and JSON libraries, and in the unlikely event that yours doesn’t, writing them is fairly simple. However, there is one important use case that JSON documents don’t cover: building plain old HTML web pages. Browsers are powerful and it’s exciting that we can build Ajax applications using only CouchDB’s JSON and HTTP APIs, but this approach is not appropriate for most public facing websites.

Comment on topic or styleHTML is the lingua franca of the web, for good reasons. By rendering our JSON documents into HTML pages, we make them available and accessible for a wider variety of uses. With the pure-Ajax approach, visually impaired visitors to our blog stand a chance of not seeing any useful content at all, as popular screen-reading browsers have a hard time making sense of pages when the content is changed on the fly via JavaScript. Another important concern for authors is that their writing be indexed by search engines. Maintaining a high-quality blog doesn’t do much good if readers can’t find it via a web search. Most search engines do not execute JavaScript found within a page, so to them an Ajax blog looks devoid of content. We also musn’t forget that HTML is likely more friendly as an archive format in the long-term, than the platform-specific JavaScript and JSON approach we used in the previous section. Also, by serving plain HTML we make our site snappier, as the browser can render meaningful content with fewer round-trips to the server. These are just a few of the reasons it makes sense to provide web content as HTML.

Comment on topic or styleThe traditional way to accomplish the goal of rendering HTML from database records is by using a middle-tier application server, such as Ruby on Rails or Django, which loads the appropriate records for a user request, runs a template function using them, and returns the resulting HTML to the visitors browser. The basics of this don’t have change in CouchDB’s case; wrapping JSON views and documents with an application server is relatively straightforward. Rather than using browser-side JavaScript load JSON from CouchDB and render dynamic pages, Rails or Django (or your framework of choice) could make those same HTTP requests against CouchDB, render the output to HTML, and return it to the browser. We won’t cover this approach in this book, as it’s specific to particular languages and frameworks, and surveying the different options would take more space than you want to read.

Comment on topic or styleCouchDB includes functionality designed to make it possible to do most of what an application tier would do, without relying on additional software. The appeal of this approach is that CouchDB can serve the whole application without dependencies on a complex environment such as might be maintained on a production web server. Because CouchDB is designed to run on client computers, where the environment is out of control of application developers, having some built in templating capabilities greatly expands the potential uses of these applications. When your application can be served by a standard CouchDB instance you gain deployment ease and flexibility.

The Show Function API

Comment on topic or styleShow functions, as they are called, have a constrained API designed to ensure cacheability and side-effect free operation. This is in stark contrast to other application servers, which give the programmer the freedom to run any operation as the result of any request. Let’s look at a few example show functions.

Comment on topic or styleThe most basic show function looks something like this:

```
function(doc, req) {
  return '<h1>' + doc.title + '</h1>';
}
```

Comment on topic or styleWhen run with a document that has a field called `title` with the content "Hello World", this function will send an HTTP response with the default content-type of `text/html`, the UTF-8 character encoding, and the body `<h1>Hello World</h1>`.

Comment on topic or styleThe simplicity of the request/response cycle of a show function is hard to overstate. The most common question we hear about it is, "how can I load another document so that I can render its content as well?" The short answer is that you can't. The longer answer is that for some applications you might use a list function to render a view result as HTML, which gives you the opportunity to use more than one document as the input of your function.

Comment on topic or styleThe basic function from a document and a request to a response, with no side effects and no alternative inputs, stays the same even as we start using more advanced features. Here's a more complex show function illustrating the ability to set custom headers:

```
function(doc, req) {
  return {
    body : '' + doc.title + '',
    headers : {
      "Content-Type" : "application/xml",
      "X-My-Own-Header": "you can set your own headers"
    }
  }
}
```

Comment on topic or styleIf this function were called with the same document as we used in the previous example, the response would have a content-type of `application/xml`, and the body `<foo>Hello World</foo>`. You should be able to see from this, how you'd be able to use show functions to generate any output you need, from any of your documents.

Comment on topic or stylePopular uses of show functions are for outputting HTML page, CSV files, or XML needed for compatibiity with a particular interface. The CouchDB test suite even illustrates using show functions to output a PNG image. To output binary data, there is the option to return a Base-64 encoded string, like this:

```
function(doc, req) {
  return {
    base64 :
      ["iVBORw0KGgoAAAANSUhEUgAAABAAAAAQCAMAAAAoLQ9TAAAAsV",
        "BMVEUAAAD///////////////////////5ur3rEBn///////////////wDBL/",
        "AADuBAe9EB3IEBz/7+//X1/qBQn2AgP/f3/ilpzsDxfpChDtDhXeCA76AQH/v7",
        "/84eLyWV/uc3bJPEf/Dw/uw8bRWmP1h4zxSlD6YGHuQ0f6g4XyQkXvCA36MDH6",
        "wMH/z8/yAwX64ODeh47BHiv/Ly/20dLQLTj98PDXWmP/Pz//39/wGyJ7Iy9JAA",
        "AADHRSTlMAbw8vf08/bz+Pv19jK/W3AAAAg0lEQVR4Xp3LRQ4DQRBD0QqTm4Y5",
        "zMxw/4OleiJlHeUtv2X6RbNO1Uqj9g0RMCuQO0vBIg4vMFeOpCWIWmDOw82fZx",
        "vaND1c8OG4vrdOqD8YwgpDYDxRgkSm5rwu0nQVBJuMg++pLXZyr5jnc1BaH4GT",
        "LvEliY253nA3pVhQqdPt0f/erJkMGMB8xucAAAAASUVORK5CYII="].join(''),
    headers : {
      "Content-Type" : "image/png"
    }
  };
}
```

Comment on topic or styleThe above function outputs a 16 x 16 pixel version of the CouchDB logo.

The JavaScript code necessary to generate images from document contents would likely be quite complex, but the ability to send Base-64 encoded binary data means that query servers written in other languages like C or PHP have the ability to output any data type.

Side-Effect Free

Comment on topic or styleWe’ve mentioned that a key constraint of show functions is that they are side effect free. This means that you can’t use them to update documents, kick off background processes, or trigger any other function. In the big picture, this is a **good thing**, as it allows CouchDB to give performance and reliability gaurantees that standard web frameworks can’t. Because a show function will always return the same result given the same input, and can’t change anything about the environment in which it runs, it’s output can be cached and intelligently reused. In a high-availability deployment with proper caching, this means that a given show function will only be called once for any particular document, and the CouchDB server may not even be contacted for subsequent requests.

Comment on topic or styleWorking without side-effects can be a little bit disorienting for developers who are used to the anything-goes approach offered by most application servers. It’s considered best practice to ensure that actions run in response to GET requests are side-effect free and cacheable, but rarely do we have the discipline to achieve that goal. CouchDB takes a different tack: because we’re a database, not an application server, we think it’s more important to enforce best practices (and ensure that developers don’t write functions that adversely effect the database server) than offer absolute flexibility. Once you’re used to working within these constraints, they start to make a lot of sense. (There’s a reason they are considered best practices.)

Design Doc

Comment on topic or styleBefore we look into show functions themselves, we’ll quickly review how they are stored on design documents. CouchDB looks for show functions stored in a top-level field called `shows` which is named like this to be parallel with `views`, `lists` and `filters`. Here’s an example design document that defines two show functions:

```
{
  "_id" : "_design/show-function-examples",
  "shows" : {
    "summary" : "function(doc, req){ ... }",
    "detail" : "function(doc, req){ ... }"
  }
}
```

Comment on topic or styleThere’s not much to note here except the fact that design documents can define multiple show functions. Now let’s see how these functions are run.

Querying Show Functions

Comment on topic or styleWe’ve described the show function API, but we haven’t yet seen how these functions are run.

Comment on topic or styleThe show function lives inside a design document, so to invoke it we append the name of the function to the design document itself, and then the id of the document we want to render:

```
GET /mydb/_design/mydesign/_show/myshow/72d43a93eb74b5f2
```

Comment on topic or styleBecause show functions (and the others like list, etc) are available as resources within the design document path, all resources provided by a particular design doc can be found under a common root, which makes custom application proxying simpler. We'll see an example of this in the next section.

Comment on topic or styleIf the document with id `72d43a93eb74b5f2` does not exist, the request will result in an HTTP 404 Not Found error response.

Comment on topic or styleHowever, show functions can also be called without a document id at all, like this:

```
GET /mydb/_design/mydesign/_show/myshow
```

Comment on topic or styleIn this case, the `doc` argument to the function has the value `null`. This option is useful in cases where the show function can make sense without a document. For instance, in the example application we'll explore in the next section, we use the same show function to provide for editing existing blog posts when a docid is given, as well as for composing new blog posts when no docid is given. The alternative would be to maintain an alternate resource (likely a static HTML attachment) with parallel functionality. As programmers we strive not to repeat ourselves, which motivated us to give show functions the ability to run without a document id.

Design Document Resources

Comment on topic or styleIn addition to the ability to run show functions, other resources are available within the design document path. This combination of features within the design document resource means that applications can be deployed without exposing the full CouchDB API to visitors, with only a simple proxy to rewrite the paths. We won't got into full detail here, but the gist of it is that end users would run the above query from a path like this:

```
GET /_show/myshow/72d43a93eb74b5f2
```

Comment on topic or styleUnder the covers, an HTTP proxy can be programmed to prepend the database and design document portion of the path, in this case `/mydb/_design/mydesign`, so that CouchDB sees the standard query. With such a system in place, end users can only access the application via functions defined on the design document, so developers can enforce constraints and prevent access to raw JSON document and view data. While it doesn't provide 100% security, using custom rewrite rules is an effective way to control the access end-users have to a CouchDB application. This technique has been used in production by a few sites at the time of this writing.

Query Parameters

Comment on topic or styleThe request object (including helpfully parsed versions of query parameters) is available to show functions as well. By way of illustration, here's a show function which returns different data based on the URL query parameters:

```
function(req, doc) {
  return "
Comment on topic or styleAye aye, " + req.parrot + "!";
}
```

Comment on topic or styleRequesting this function with a query parameter will result in the query parameter being used in the output:

```
GET /mydb/_design/mydesign/_show/myshow?parrot=Captain
```

Comment on topic or styleIn this case we’ll see the output: `<p>Aye aye, Captain!</p>`

Comment on topic or styleAllowing URL parameters into the function does not effect cacheability, as each unique invocation results in a distinct URL. However, making heavy use of this feature will lower your cache effectiveness. Query parameters like this are most useful to do things like switch the mode or the format of the show function output. It’s recommended that you avoid using them for things like inserting custom content (such as the requesting user’s nickname) into the response, as that will mean that each users’s data must be cached seperately.

Accept Headers

Comment on topic or stylePart of the HTTP spec allows for clients to give hints to the server about which content-types they are capable of accepting. At this time, the JavaScript query server shipped with CouchDB 0.10.0 contains helpers for working with Accept headers. However, web browser support for Accept headers is **very poor**, which has prompted frameworks such as Ruby on Rails to remove thier support for them. CouchDB may or may not follow suite here, but the fact remains that you are discouraged from relying on Accept headers for applications which will be accessed via web browsers.

Comment on topic or styleThere is a suite of helpers for Accept headers present as well, which allows you to specify the format in a query parameter as well. For instance

```
GET /db/_design/app/_show/post
Accept: application/xml
```

Comment on topic or styleis equivalent to a similar URL with mismatched Accept headers. This is because browsers don’t use sensible Accept headers for feed URLs. Browsers 1, Accept headers 0. Yay browsers.

```
GET /db/_design/app/_show/post?format=xml
Accept: x-foo/whatever
```

Comment on topic or styleThe request function allows developers to switch response Content-types based on the client’s request. The next example adds the ablity to return either HTML, XML, or developer-designated content-type: "foo".

Comment on topic or styleCouchDB’s `main.js` library provides the `provides("format", render_function)` function, which makes it easy for developers to handle client requests for multiple Mime Types in one form function.

Comment on topic or styleThis function also shows off the use of `registerType(name, mime_types)`, which adds new types to mapping object used by `respondWith`. The end result is ultimate flexibility for developers, with an easy interface for handling different types of requests. `main.js` uses a JavaScript port of *Mimeparse*, an open source reference implementation, to provide this service.

Etags

Comment on topic or styleWe’ve mentioned that show function requests are side effect free and cacheable, but we haven't discussed the mechanism used to accomplish this. *Etags* are a standard HTTP mechanism for indicating whether a cached copy of an HTTP response is still current. Essentially, when the client makes its first request to a resource, the response is accompanied by an Etag, which is an opaque string token unique to the version of the resource requested. The second time the client makes a request against the same resource, it sends along the original Etag with the request. If the server determines that the Etag still matches the resource, it can avoid sending the full response, instead replying with message that essentially says "you have the latest version already."

Comment on topic or styleWhen implemented properly, the use of Etags can cut down significantly on server load. CouchDB provides an Etag header, so that by using an HTTP proxy-cache like Squid or memcached, you'll instantly remove load from CouchDB.

Functions and Templates

Comment on topic or styleCouchDB's process runner only looks at the functions stored under `show`, but we'll want to keep the template html seperate from the content negotiation logic. The `couchapp` script handles this for us, using the `!code` and `!json` handlers.

Comment on topic or styleLet's follow the show function logic through the files Sofa splits it into. Here's Sofa's `edit` show function:

```
function(doc, req) {
  // !json templates.edit
  // !json blog
  // !code vendor/couchapp/path.js
  // !code vendor/couchapp/template.js

  // we only show html
  return template(templates.edit, {
    doc : doc,
    docid : toJSON((doc && doc._id) || null),
    blog : blog,
    assets : assetPath(),
    index : listPath('index','recent-posts',{descending:true,limit:8})
  });
}
```

Comment on topic or styleIt should look pretty straightforward. First, we have the function's *head* or *signature* that tells us we are dealing with a function that takes two arguments `doc` and `req`.

Comment on topic or styleThe next four lines are comments, as far as JavaScript is concerned. But these are special documents. CouchApp knows how to read these special comments on top of the show function. They include *macros*; a macro starts with a bang `!` and a name. Currently, CouchApp supports the two macros `!json` and `!code`.

The `!json` Macro

Comment on topic or styleThe `!json` macro takes one argument, the path to a file in the CouchApp directory hierarchy in the *dot notation*: Instead of a slash `/` (or backslash `\`), you use a dot `.`. The `!json` macro then reads the contents of the file and puts them into a variable that has the same name as the file's path in dot notation.

Comment on topic or styleFor example, if you use the macro like this:

```
// !json template.edit
```

Comment on topic or styleCouchDB will read the file `template/edit.*` and place its contents into a variable:

```
var template.edit = "contents of edit.*"
```

Comment on topic or styleWhen specifying the path you omit the file's extension. That way you can read `.json` or `.js`, `.html` files, or any other files into variables in your functions. Because the macro matches files with any extensions, you can't have two files with the same name but different extensions.

Comment on topic or styleIn addition, you can specify a directory and CouchApp will load all the files in this directory and any subdirectory.

```
// !json template
```

Comment on topic or styleCreates:

```
var template.edit = "contents of edit.*"
var teplate.post = "contents of post.*"
```

Comment on topic or styleNote that the macro also takes care of creating the top level `template` variable, we just omitted that here for brevity. The `!json` macro will only generate valid JavaScript.

The `!code` Macro

Comment on topic or styleThe `!code` macro is similar to the `!json` macro, but it serves a slightly different purpose. Instead of making the contents of one ore more files available as variables in your functions, it replaces itself with the contents of the file referenced in the argument to the macro.

Comment on topic or styleThis is useful for sharing library functions between CouchDB functions (map/reduce/show/list/validate) without having to maintain their source code in multiple places.

Comment on topic or styleOur example shows this line:

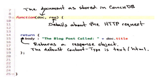
```
// !code vendor/couchapp/path.js
```

Comment on topic or styleIf you look at the CouchApp sources, there is a file in `vendor/couchapp/path.js` that includes a bunch of useful function related to the URL path of a request. In the example above CouchApp will replace the line with the contents of `path.js` making the functions locally available to the show function.

Comment on topic or styleThe `!code` macro can only load a single file at a time.

Learning Shows

Comment on topic or styleBefore we dig into the full complex beast (yeah, I said it) that will render the Post permalink pages, let's look at some *Hello World* form examples. The first one just shows the function arguments, and the simplest possible return value.



```
function(doc, req) {


    return {
     body : "The Blog Post Called: " + doc.title



    };






}
```

The document as stored in CouchDB
Details about the HTTP request
Returns a response object.
The default Content-Type is text/html.

Figure: Basic Form Function

Comment on topic or style*A **form** is a JavaScript function that converts a document, and some details about the HTTP request, into an HTTP response.* Typically it will be used to construct HTML, but it is also capable of returning Atom feeds, images, or even just filtered JSON.

Comment on topic or styleThe document argument is just like the documents passed to Map functions.

Using Templates

Comment on topic or styleThe only thing missing from the show function development experience, is the ability to render HTML without ruining your eyes looking at a whole lot of manual string concatenation, among other unpleasantries. Most programming environments solve this problem with templates, eg: documents which look like HTML but have portions of thier content filled out dynamically.

Comment on topic or styleDynamically combining template strings and data in JavaScript is a solved problem. However it hasn't caught on partly because JavaScript doesn't have very good support for multiline "heredoc" strings. After all, once you get through escaping quotes and leaving out newlines, it's not much fun to edit HTML templates inlined into JavaScript code. We'd much rather keep our templates in seperate files, where we can avoid all the escaping work, and they can be syntax-highlighted by our editor.

Comment on topic or styleThe `couchapp` script has a couple of helpers to make working with templates and library code stored in design documents less painful. In the function below, we use them to load a Blog Post template, as well the JavaScript function responsible for rendering it.

Comment on topic or styleAs you can see, we take the opportunity in the function to strip JavaScript tags from the form post. That regex is not secure, and the blogging application is only meant to be written to by it's owners, so we should probably drop the regex, and simplify the function to avoid transforming the document, instead passing it directly to the template. Or else we should port a known-good santinitization routine from another language and provide it in the templates library.

Writing Templates

Comment on topic or styleWorking with templates, instead of trying to cram all the presentation into one file, makes editing forms a little more relaxing. The templates are stored in their own file, so you don't have to worry about JavaScript or JSON encoding, and your text editor can highlight the template's HTML syntax. CouchDB's JavaScript query server includes the E4X extensions for JavaScript, which can be helpful for XML templates, but does not work well for HTML. We'll explore E4X templates in the Viewing Lists of Blog Posts, when we cover forms for views, which makes providing an ATOM feed of view results easy and memory-efficient.

Figure: The Blog Post Template

Comment on topic or styleTrust us when we say that looking at this HTML page is much more relaxing than trying to understand what a raw-JavaScript one is trying to do. The template library we're using in the example blog is by John Resig, and was chosen for simplicity. It could easily be replaced by one of many other options, such as the Django template language, available in JavaScript.

Comment on topic or styleThis is a good time to note that CouchDB's architecture is designed to make it simple to swap out languages for the query servers. With a query server written in Lisp or Python or

Ruby (or any language that supports JSON and stdio) you could have an even wider variety of templating options. The CouchDB team recommends sticking with JavaScript as it provides the highest level of support and interoperability, but other options are available.

≈= List Functions

Comment on topic or styleJust as show functions convert documents to arbitrary output formats, CouchDB *list functions* allow you to render the output of view queries in any format. The powerful iterator API allows for flexibility to filter and aggregate rows on the fly, as well as outputting raw transformations for an easy way to make Atom feeds, HTML lists, CSV files, config files, or even just modified JSON.

Comment on topic or styleList functions are stored under the `lists` field of a design document. Here's an example design document that contains two list functions:

```
{
  "_id" : "_design/foo",
  "_rev" : "1-67at7bg",
  "lists" : {
    "bar" : "function(head, req) { var row; while (row = getRow()) { ... } }",
    "zoom" : "function() { return 'zoom!' }",
  }
}
```

Arguments to the List Function

Comment on topic or styleThe function is called with two arguments, which can sometimes be ignored, as the row data itself is loaded during function execution. The first argument, `head`, contains information about the view. Here's what you might see looking at a JSON representation of `head`:

```
{total_rows:10, offset:0}
```

Comment on topic or styleThe request itself is a much richer data structure. This is the same request object that is available to shows, update and filter functions. We'll go through it in detail here as a reference. Here's the example `req` object:

```
{
  "info": {
    "db_name": "test_suite_db","doc_count": 11,"doc_del_count": 0,
    "update_seq": 11,"purge_seq": 0,"compact_running": false,"disk_size": 4930,
    "instance_start_time": "1250046852578425","disk_format_version": 4},
```

Comment on topic or styleThe database info, as available in an info request against a db's URL, is included in the request parameters. This allows you to stamp rendered rows with an update sequence, and know the database you are working with.

```
  "verb": "GET",
  "path": ["test_suite_db","_design","lists","_list","basicJSON","basicView"],
```

Comment on topic or styleThe HTTP verb and the path in the client from the client request are useful, especially for rendering links to other resources within the application.

```
  "query": {"foo":"bar"},
```

Comment on topic or styleIf there are parameters in the query string (in this case corresponding to `?foo=bar`) they will be parsed and available as a JSON object at `req.query`.

```
  "headers":
    {"Accept": "text/html,application/xhtml+xml ,application/xml;q=0.9,*/*;q=0.8",
    "Accept-Charset": "ISO-8859-1,utf-8;q=0.7,*;q=0.7","Accept-Encoding":
"gzip,deflate",
    "Accept-Language": "en-us,en;q=0.5","Connection": "keep-alive",
    "Cookie": "_x=95252s.sd25; AuthSession=","Host": "127.0.0.1:5984",
    "Keep-Alive": "300",
    "Referer": "http://127.0.0.1:5984/_utils/couch_tests.html?
script/couch_tests.js",
    "User-Agent": "Mozilla/5.0 Gecko/20090729 Firefox/3.5.2"},
  "cookie": {"_x": "95252s.sd25","AuthSession": ""},
```

Comment on topic or styleHeaders give list and show functions the ability to provide the content type response that the client prefers, as wel as other nifty things, like cookies. Note that cookies are also parsed into a JSON representation. Thanks Mochiweb!

```
  "body": "undefined",
  "form": {},
```

Comment on topic or styleIn the case where the verb is POST, the request body (and a form decoded JSON representation of it if applicable) are available as well.

```
  "userCtx": {"db": "test_suite_db","name": null,"roles": ["_admin"]}
}
```

Comment on topic or styleFinally, the `userCtx` is the same as that sent to the validation function. It provides access to the database the user is authenticated against, the users's name, and the roles they've been granted. In the example above, you see an anonymous user working with a CouchDB node that is in "admin party" mode. Unless an admin is specified, everyone is an admin.

Comment on topic or styleThat's enough about the arguments to list functions, now it's time to look at the mechanics of the function itself.

An Example List Function

Comment on topic or styleSo let's put this knowledge to use. In the chapter intro, we mentioned using lists to generate config files. One fun thing about that is that if you keep your configuration information in CouchDB, and generate it with lists, you don't have to worry about being able to regenerate it again, because you know the config will be generated by a pure function from your database, and not other sources of information. This level of isolation will ensure that your config files can be generated correctly, as long as CouchDB is running. Because you can't fetch data from other system services, files, or network sources, you can't accidentally write a config file generator that fails due to external factors.

Comment on topic or styleChris got excited about the idea of using `_list` functions to generate config files for the sort of services people usually configure using CouchDB, specifically via Chef, an Apache-licensed infrastructure automation tool. The key feature for infrastructure automation is that deployment scripts are idempotent - that is, running your scripts multiple times will have the same intended effect as running them once, something that becomes critical when a script fails half way through. This encourages crash-only design, where your scripts can bomb out multiple times but your data remains consistent, because it takes the guesswork out of provisioning and updating servers in the case of previous failures.

Comment on topic or styleLike map, reduce, and show functions, lists are pure-functions, from a view query and an http request, to an output format. They can't make queries against remote services or otherwise access outside data, so you know they are repeatable. Using a `_list` function to generate an http server configuration file ensures that the configuration is generated repeatably, based only on the state of the database.

Comment on topic or styleImagine you are running a shared hosting platform, with one name-based virtual host per user. You'll need a config file that starts out with some node configuration (which modules to use, etc) and is followed by one config section per user, setting things like the user's http directory, subdomain, forwarded ports, etc.

```
function(head, req) {
  // helper function definitions would be here...
  var row, userConf, configHeader, configFoot;
  configHeader = renderTopOfApacheConf(head, req.query.hostname);
  send(configHeader);
```

Comment on topic or styleIn the first block of the function, we're rendering the top of the config file, using the function `renderTopOfApacheConf(head, req.query.hostname)`. This may include information that's posted into the function, like the internal name of the server which is being configured, or the root directory in which user html files are organized. We won't show the function body, but you can imagine that it would return a long multi-line string that handles all the global configuration for your server, and sets the stage for the per-user configuration which will be based on view data.

Comment on topic or styleThe call to `send(configHeader)` is the heart of your ability to render text using list functions. Put simply, it just sends an HTTP chunk to the client, with the content of the strings pasted to it. There is some batching behind the scenes, as CouchDB speaks with the JavaScript runner with a synchronous protocol, but from the perspective of a programmer, `send()` is how HTTP chunks are born.

Comment on topic or styleNow that we've rendered and sent the file's head, it's time to start rendering the list itself. Each list item will be the result of converting a view row to a virtual hosts configuration element. The first thing we do is call `getRow()` to get a row of the view.

```
  while (row = getRow()) {
    var userConf = renderUserConf(row);
    send(userConf)
  }
```

Comment on topic or styleThe while loop used here will continue to run until `getRow()` returns null, which is how CouchDB signals to the list function that all valid rows (based on the view query parameters) have been exhausted. Before we get ahead of ourselves, let's check out what happens when we do get a row.

Comment on topic or styleIn this case, we simply render a string based on the row, and send it to the client. Once all rows have been rendered, the loop is complete. Now is a good time to note that the function has the option to return early. Perhaps it is programmed to stop iterating when it sees a particular users document, or based on a tally it's been keeping of some resource allocated in the configuration. In those cases, the loop can end early with a `break` statement or other method. There's no requirement for the list function to render every row that is sent to it.

```
  configFoot = renderConfTail();
  return configFoot;
}
```

Comment on topic or styleFinally, we close out the configuration file, and return the final string value to be sent as the last HTTP chunk. The last action of a list function is always to return a string, which will be sent as the final HTTP chunk to the client.

Comment on topic or styleTo use our config file generation function in practice, we might run a command-line script that looks like:

```
curl http://localhost:5984/config_db/_design/files/_list/apache/users?
hostname=foobar > apache.conf
```

Comment on topic or styleThis will render our Apache config based on data in the user's view, and save it to a file. What a simple way to build a reliable configuration generator!

List Theory

Comment on topic or styleNow that we've seen a complete list function, it's worth mentioning some of the helpful properties they have.

Comment on topic or styleThe most obvious thing is the iterator-style API. Because each row is loaded independently by calling `getRow()`, it's easy not to leak memory. The list function API is capable of rendering lists of arbitrary length without error, when used correctly.

Comment on topic or styleOn the other hand, this API gives you the flexibility to bundle a few rows in a single chunk of output, so if you had a view of say, user accounts, followed by subdomains owned by that account, you could use a slightly more complex loop to build up some state in the list function, for rendering more complex chunks. Let's look at an alternate loop section:

```
var subdomainOwnerRow, subdomainRows = [];
while (row = getRow()) {
```

Comment on topic or styleWe've entered a loop which will continue until we have reached the endkey of the view. The view is structured so that user profile row is emitted, followed by all of that user's subdomains. We'll use the profile data and the subdomain infomation to template the configuration for each individual user. This means we can't render any subdomain configuration until we know we've received all the rows for the current user.

```
  if (!subdomainOwnerRow) {
    subdomainOwnerRow = row;
```

Comment on topic or styleThis case is only true for the first user, we're merely setting up the initial conditions.

```
  } else if (row.value.user != subdomainOwnerRow.value.user) {
```

Comment on topic or styleThis is the end case. It will only ever be called after all the subdomain rows for the current user have been exhausted. It is triggered by a row with a mismatched user, indicating that we have all the subdomain rows.

```
    send(renderUserConf(subdomainOwnerRow, subdomainRows));
```

Comment on topic or styleWe know we are ready to render everything for the current user, so we pass the profile row and the subdomain rows to a render function (which nicely hides all the gnarly nginx config details from our fair reader). The result is sent to the http client, which writes it to the config file.

```
    subdomainRows = [];
    subdomainOwnerRow = row;
```

Comment on topic or styleWe've finished with that user, so let's clear the rows and start working on the next user.

```
  } else {
    subdomainRows.push(row);
```

Comment on topic or styleAhh, back to work, collecting rows.

```
  }
}
send(renderUserConf(subdomainOwnerRow, subdomainRows));
```

Comment on topic or styleThis last bit is tricky - after the loop is finished (we've reached the end of the view query) we've still got to render the last user's config. Wouldn't want to forget that!

Comment on topic or styleThe gist of this loop section is that we collect rows that belong to a particular user until we see a row that belongs to another user, at which point we render output for the first user, clear our state, and start working with the new user. Techniques like this show how much flexibility is allowed by the list iterator API.

Comment on topic or styleMore uses along these lines include filtering rows that should be hidden from a particular result set, finding the top N grouped reduce values (eg to sort a tag cloud by popularity), and even writing custom reduce functions (as long as you don't mind that reductions are not stored incrementally.)

Querying Lists

Comment on topic or styleWe haven't looked in detail at the ways list functions are queried. Just like show functions, they are resources available on the design document. The basic path to a list function is as follows:

```
/db/_design/foo/_list/list-name/view-name
```

Comment on topic or styleBecause the list name and the view name are both specified, this means it is possible to render a list against more than one view. So for instance you could have a list function that renders blog comments in the Atom XML format, and then run it against both a global view of recent comments, as well as a view of recent comments by blog post. This would allow you to use the same list function to provide an Atom feed for comments across an entire site, as well as individual comment feeds for each post.

Comment on topic or styleAfter the path to list, come the view query parameter. Just like a regular view, calling a list function without any query parameters results in a list that reflects every row in the view. Most of the time you'll want to call it with query parameters to limit the returned data.

Comment on topic or styleYou're alread familar with the view query options from the Finding Your Data With Views chapter. The same query options apply to the `_list` query. Let's look at URLs side by side.

A JSON View Query

```
GET /db/_design/sofa/_view/recent-posts?descending=true&limit=10
```

Comment on topic or styleThis view query is just asking for the 10 most recent blog posts. Of course this query could include paramters like `startkey` or `skip` – we're leaving them out for simplicity. To run the same query through a list function, we access it via the list resource:

The HTML List Query

```
GET /db/_design/sofa/_list/index/recent-posts?descending=true&limit=10
```

Comment on topic or styleThe `index` list here is a function from JSON to HTML. Just like the above view query, additional query parameters can be applied to paginate through the list. As we'll see in the example application chapter, once you have a working list, adding pagination is trivial.

The Atom List Query

```
GET /db/_design/sofa/_list/index/recent-posts?descending=true&limit=10&format=atom
```

Comment on topic or styleThe list function can also look at the query parameters, and do things like switch which output to render based on parameters. You can even do things like pass the user-name into the list using a query parameter (but it's not recommended, as you'll ruin cache-efficiency.)

Lists, Etags, and Caching

Comment on topic or styleJust like show functions and view queries, lists are sent with proper HTTP Etags, which makes them cacheable by intermediate proxies. This means that if your server is starting to bog down in list-rendering code, it should be possible to relieve load by using a caching reverse proxy like Squid. We won't go into the details of Etags and caching here, as they are covered in the Shows chapter.

Standalone Applications

Comment on topic or styleCouchDB is useful for many areas of application. Because of it's incremental map reduce and replication characteristics, it is especially well suited to online interactive document and data management tasks. These are the sort of workloads experienced by the majority of web applications. This coupled with CouchDB's HTTP interface make it a natural fit for the web.

Comment on topic or styleIn this section we'll tour a document-oriented web application – a basic blog implementation. As a lowest common denominator, we'll be using plain old HTML and JavaScript. The lessons learned should apply to Django/Rails/Java-style middleware applications and even to intensive map reduce data mining tasks. CouchDB's API is the same, regardless of whether you're running a small installation or an industrial cluster.

Comment on topic or styleThere is no right answer about which application development framework you should use with Couch. We've seen successful applications in almost every commonly used language and framework. For this example application we'll use a two layer architecture: CouchDB as the data layer, and the browser for the user interface. We think this is a viable model for many document-oriented applications, but it also makes a great way to teach CouchDB, because we can easily assume that all of you have a browser at hand, without having to ensure you're familiar with a particular server-side scripting language.

Use the Correct Version

Comment on topic or styleThis section is interactive, so be prepared to follow along with your laptop and a running CouchDB database. We've made the full example application and all of the source code examples available online, so you'll start by downloading the current version of the example application and installing it on your CouchDB instance.

Comment on topic or styleA challenge of writing this book and preparing it for production is that CouchDB is evolving at a rapid pace. The basics haven't changed in a long time, and probably won't change much in the future, but things around the edges are moving forward rapidly for CouchDB's 1.0 release.

Comment on topic or styleThis book is going to press as CouchDB version 0.10.0 is about to be released. Most of the code was written against 0.9.1 and the development trunk which is becoming version 0.10. We'll work with two other software packages: CouchApp, which is a set of tools for editing and sharing CouchDB application code; and Sofa, the example blog itself.

Comment on topic or styleAs a reader, it is your responsibility to use the correct versions of these packages. For CouchApp, the correct version is always the latest. The correct version of Sofa depends on which version of CouchDB you are using. To see which version of CouchDB you are using run the following command:

```
curl http://127.0.0.1:5984
```

Comment on topic or styleAnd you should see something like one of the three below:

```
{"couchdb":"Welcome","version":"0.9.1"}
{"couchdb":"Welcome","version":"0.10.0"}
{"couchdb":"Welcome","version":"0.11.0a858744"}
```

Comment on topic or styleThese three correspond to versions 0.9.1, 0.10.0, and *trunk*. If the version of CouchDB you have installed is 0.9.1 or earlier, you should upgrade to at least 0.10.0, as Sofa makes use of feature not present until 0.10.0. There is an older version of Sofa that will work, but this book covers features and APIs that are part of the 0.10 release of CouchDB. It's conceivable that there will be a 0.9.2, 0.10.1 and even a 0.10.2 release by the time you read this. Please use the latest release of whichever version you prefer.

Comment on topic or style*Trunk* refers to the latest development version of CouchDB available in the Apache Subversion respository. We recommend that you use a released version of CouchDB, but as developers, we often use trunk. Sofa's master branch will tend to work on trunk, so if you want to stay on the cutting edge, that's they way to do it.

Portable JavaScript

Comment on topic or styleIf you're not familiar with JavaScript, we hope the source examples are given with enough context and explanation so that you can keep up. If you are familiar with JavaScript, you’re probably already excited that CouchDB supports view and template rendering JavaScript functions.

Comment on topic or styleOne of the advantages to building applications that can be hosted on any standard CouchDB installation, is that they are portable via replication. This means that your application, if you develop it to be served directly from CouchDB, gets offline mode "for free". Local data makes a big difference for users, in a number of ways which we won’t get into here. We call applications that can be hosted from a standard CouchDB, *CouchApps*.

Comment on topic or styleCouchApps are a great vehicle for teaching CouchDB because we don’t need to worry about picking a language or framework, we’ll just work directly with CouchDB, so that readers get a quick overview of a familiar application pattern. Once you’ve worked through the example app you’ll have seen enough to know how to apply CouchDB to your problem domain. If you don’t know much about Ajax development you’ll learn a little about jQuery as well, we hope you find the experience relaxing.

Applications are Documents

Comment on topic or styleApplications are stored as design documents. You can replicate design documents just like everything else in CouchDB. Because design documents can be replicated, whole CouchApps are replicated. CouchApps can be updated via replication, but they are also easily "forked" by the users, who can alter the source code at will.

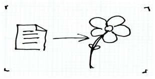


Illustration: CouchDB executes application code stored in Design Documents

Comment on topic or styleBecause applications are just a special kind of document, they are easy to edit and share.

Comment on topic or styleThinking of peer-based application replication takes me back to my first year of high school, when my friends and I would share little programs between the TI-85 graphing calculators we were required to own. Two calculators could be connected via a small cable and we’d share physics cheat sheets, Hangman, some multi-player text-based adventures, and at the height of our powers, I believe there may have been a Doom clone running.

Comment on topic or styleThe TI-85 programs were in Basic, so everyone was always hacking each others hacks. Perhaps the most ridiculous program was a version of Spy Hunter that you controlled with your mind. The idea was that you could influence the pseudo random number generator by concentrating hard enough, and thereby control the game. Didn’t work. Anyway, the point is that when you give people access to the source code, there’s no telling what might happen.

Comment on topic or styleIf one person doesn’t like the æsthetics of your application, she can tweak the CSS. If she doesn’t agree with your interface choices, she can improve the HTML. If she wants to modify the functionality, she can edit the JavaScript. Taken to the extreme, she may want to completely fork your application for her own purposes. When she shows the modified version to her friends and co-workers, and hopefully you, there is a chance that more people may want to make improvements.

Comment on topic or styleAs the original developer, you have the control over your version and you can accept or reject changes as you see fit. If someone messes about with the source code for a local application and breaks things beyond repair they can replicate the original copy from your server.

Comment on topic or styleOf course, this may not be your cup of tea. Don't worry, you can be as restrictive as you like with CouchDB. You can restrict access to data however you wish, but beware of the opportunities you might be missing. There is a middle ground between open collaboration and restricted access controls.

Comment on topic or styleOnce you've finished the installation procedure, you'll be able to see the full application code for Sofa, both in your text editor, and as a design document in Futon.

Standalone

Comment on topic or styleWhat happens if you add a HTML file as a document attachment? Exactly the same thing. We can serve Web pages directly with CouchDB. Of course, we might also need images, stylesheets, or scripts. No problem, just add these resources as document attachments and link to them using relative URIs.

Comment on topic or styleLet's take a step back. What do we have so far? A way to serve HTML documents and other static files on the Web. That means that we can build and serve traditional Web sites using CouchDB. Fantastic! Right? Isn't this a little like reinventing the wheel? Well, a very important difference is that we also have a document database sitting in the background. We can talk to this database using the JavaScript served up with our Web pages. Now we're really cooking with gas!

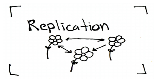
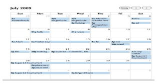


Illustration: Replicating application changes to a group of friends

Comment on topic or styleCouchDB's features are a foundation for building standalone Web application backed by a powerful database. As a proof of concept, look no further than CouchDB's built-in administrative interface. Futon is a fully functional database management application built using HTML, CSS, and JavaScript. Nothing else. CouchDB and Web applications go hand in hand.

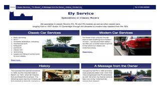
In the Wild

Comment on topic or styleThere are plenty of examples of CouchApps in the wild. Here are just a few screenshots of sites and applications that use a standalone CouchDB architecture.



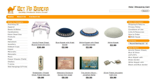
Screenshot: Group Calendar

Comment on topic or styleDamien decided to see how long it takes to implement a shared calendar with realtime updates as events are changed on the server. It took about an afternoon, thanks to some amazing open-source jQuery plugins. The calendar demo is still running on Chris's server.

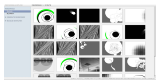


Screenshot: Ely Service

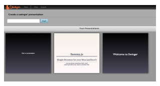
Comment on topic or styleJason Davies swapped the backend of the Ely Service website out with CouchDB, without changing anything visible to the user. The technical details are covered on his blog.


Screenshot: Bet Ha Bracha

Comment on topic or styleJason also converted his mom’s ecommerce website to a CouchApp. It uses the `_update` handler to hook into different transaction gateways.


Screenshot: Processing JS Studio

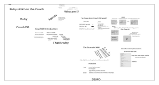
Comment on topic or styleProcessing JS is a toolkit for building animated art that runs in the browser. Processing JS Studio is a gallery for Processing sketches.


Screenshot: Swinger

Comment on topic or styleSwinger is a CouchApp for building and sharing presentations. It uses the Sammy JavaScript application framework. "Sammy + CouchDB == brothers from another mother."

Screenshot: Nymphormation

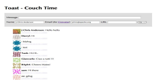
Comment on topic or styleNymphormation is a link sharing and tagging site by Benoît Chesneau. It uses CouchDB’s cookie authentication as well as making it possible to share links using replication.


Screenshot: Boom Amazing

Comment on topic or styleBoom Amazing is a CouchApp by Alexander Lang that allows you to zoom, rotate, and pan around an SVG file, record the different positions and then replay those for a presentation… or something else. (from the Boom Amazing Readme)

Screenshot: Twitter Client

Comment on topic or styleThe CouchDB Twitter Client was one of the first standalone CouchApps to be released. It’s documented in Chris’s blog post, My Couch or Yours, Shareable Apps are the Future The screenshot shows the word-cloud generated from a Map Reduce view of CouchDB’s archived tweets. The cloud is normalized against the global view, so universally common words don’t dominate the chart.


Screenshot: Toast

Comment on topic or styleToast is a chat application which allows users to create channels and then invite others to real-time chat there. It was initially a demo of the `_changes` event loop but started to take off as a way to chat.

Screenshot: Sofa

Comment on topic or styleSofa is the example app for this section, and it has been deployed by a few different authors around the web. This screenshot is from Jan’s tumblelog.

Comment on topic or styleTo see Sofa in action, visit Chris’s site which has been running Sofa since late 2008.

Wrapping Up

Comment on topic or styleChris decided to port his blog from Ruby on Rails to CouchDB. He started by exporting Rails ActiveRecord objects as JSON documents, paring away some features and adding others, as he converted to HTML and JavaScript.

Comment on topic or styleThe resulting blog engine features access controlled posting, open comments with the possibility of moderation, Atom feeds, Markdown formatting, and a few other little goodies. This book is not about *jQuery*, so while we use this JavaScript library, we’ll refrain from dwelling on it. Readers familiar with using asynchronous *XMLHttpRequest (XHR)* should feel right at home with the code. Keep in mind that the figures and code samples in this section elide many of the bookkeeping details.

Comment on topic or styleWe will be studying this application and learning how it exercises all the core features of CouchDB. The skills learnt in this section should be broadly applicable to any CouchDB application domain, whether you intend to build a self hosted CouchApp or not.

Managing Design Documents

Comment on topic or styleApplications that live in CouchDB — nice. You just attach a bunch of HTML and JavaScript files to a design document and you are good to go. Spice that up with view-powered queries, and show functions that render any media type from your JSON documents and you have all it takes to write self-contained CouchDB applications.

Working with the Example Application

Comment on topic or styleIf you want to install and hack on your own version of Sofa while you read the following chapters, we’ll be using CouchApp to upload the source code as we explore it.

Comment on topic or styleWe’re particularly excited by the prospect of deploying applications to CouchDB, because depending on a least-common denominator environment, that encourages users to control not just the data but also the source code, will let more people build personal web apps. And when the web app you’ve hacked together in your spare time hits the big-time, the ability of CouchDB to scale to larger infrastructure sure doesn’t hurt.

Comment on topic or styleIn a CouchDB design doc there are a mix of development languages (HTML, JS, CSS) that go into different places like attachments and design document attributes. Ideally you want your development environment to help you as much as possible. More importantly, you're used to: Proper syntax highlighting, validation, integrated documentation, macros, helpers and whatnot. Editing HTML and JavaScript code as the string-attributes of a JSON object is not exactly modern computing.

Comment on topic or styleLucky for you, we've been working on a solution: Enter *CouchApp*. CouchApp lets you develop CouchDB applications in a convenient directory hierarchy: Views and shows are separate `.js`-files neatly organized, your static assets (CSS, images) have their place and with the simplicity of a `couchapp push` you save your app to a design doc in CouchDB. Make a change? `couchapp push` and off you go.

Comment on topic or styleThis chapter guides you through the installation and moving parts of *CouchApp*. You will learn what other neat helpers it has in store to make your life easier (Gosh, aren't we awfully nice?). Once we have CouchApp, we'll use it to install and deploy Sofa to a CouchDB database.

Installing CouchApp

Comment on topic or styleThe CouchApp Python script and JavaScript framework we'll be using grew out of the work designing this example application. It's now in use for a variety of applications, and has a mailing list, wiki, and a community of hackers. Just search the internet for "couchapp" to find the latest information. Many thanks to Benoît Chesneau for building and maintaining the library (and contributing to CouchDB's Erlang codebase and many of the Python libraries.)

Comment on topic or styleCouchApp is easiest to install using the Python `easy_install` script, which is part of the `setuptools` package. If you are on the Mac, `easy_install` should already be available. If `easy_install` is not installed and you are on a Debian variant, such as Ubuntu, you can use the following command to install it:

```
sudo apt-get install python-setuptools
```

Comment on topic or styleIf all goes well, you should see output like the following:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  python-setuptools
0 upgraded, 1 newly installed, 0 to remove and 53 not upgraded.
Need to get 195kB of archives.
After this operation, 909kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com karmic/main python-setuptools 0.6c9-0ubuntu4
[195kB]
Fetched 195kB in 1s (108kB/s)
Selecting previously deselected package python-setuptools.
(Reading database ... 117857 files and directories currently installed.)
Unpacking python-setuptools (from .../python-setuptools_0.6c9-
0ubuntu4_all.deb) ...
Setting up python-setuptools (0.6c9-0ubuntu4) ...
```

Comment on topic or styleOnce you have `easy_install`, installing CouchApp should be as easy as:

```
sudo easy_install -U couchapp
```

Comment on topic or styleYou should see output like this:

```
Searching for couchapp
Reading http://pypi.python.org/simple/couchapp/
Reading http://github.com/couchapp/couchapp/tree/master
Best match: Couchapp 0.3.31
Downloading http://pypi.python.org/packages/source/C/Couchapp/Couchapp-
0.3.31.tar.gz#md5=a346459155995942dea462e183f104f1
Processing Couchapp-0.3.31.tar.gz
Running Couchapp-0.3.31/setup.py -q bdist_egg --dist-dir /tmp/easy_install-
Ey7eZR/Couchapp-0.3.31/egg-dist-tmp-QObSXT
Adding Couchapp 0.3.31 to easy-install.pth file
Installing couchapp script to /usr/local/bin

Installed /usr/local/lib/python2.6/dist-packages/Couchapp-0.3.31-py2.6.egg
Processing dependencies for couchapp
Finished processing dependencies for couchapp
```

Comment on topic or styleHopefull this worked and you are ready to start using CouchApp. If not, read on…

Comment on topic or styleThe most common problem people experience installing CouchApp is with old versions of dependencies, especially `easy_install` itself. If you experienced an installation error, the best next step is to attempt to upgrade setuptools and then upgrade CouchApp, like this:

```
sudo easy_install -U setuptools
sudo easy_install -U couchapp
```

Comment on topic or styleIf you have other problems installing CouchApp, have a look at setuptools for Python’s easy install troubleshooting, or visit the CouchApp mailing list.

Using CouchApp

Comment on topic or styleInstalling CouchApp via `easy_install` should, as they say, be easy. Assuming all goes according to plan, it take care of any dependencies and puts the `couchapp` utility into your system’s `PATH` so you can immediately begin by running the help command:

```
couchapp --help
```

Comment on topic or styleYou should see output like this:

CouchApp help listing

```
$ couchapp --help
Usage: couchapp [options] cmd

Options:
  --version          show program's version number and exit
  -h, --help         show this help message and exit
  -v                 print message to stdout
  -q                 don't print any message

  Generate a new CouchApp! (start here):
    couchapp generate <appname> [appdir]

  Pushes a CouchApp to CouchDB:
    couchapp push [options] [appdir] [appname] [dburl]
    --atomic         store atomically the couchapp.
    --export         Export the generated design doc to your console. If
                     --output is specified, write to the file.
    --output=OUTPUT  Combined with --export it allow you to save the generated
                     design doc to the file.

  Clones/Pulls a CouchApp from a url (like http://host/db/_design/CA_name):
    couchapp clone/pull <dburl> [dir]

  Initialize CouchApp .couchapprc:
    couchapp init [options] <appdir>

    --db=DB          full url of default database

  Install a vendor:
    couchapp vendor install vendor_url [option][appdir]

    --scm=SCM        scm used to install the vendor, by default git
```

Comment on topic or styleWe'll be using the `clone` and `push` commands. Clone pulls an application from a running instance in the cloud, saving it as a directory structure on your filesystem. Push deploys a standalone CouchDB application from your filesystem to any CouchDB over which you have administrative control.

Download the Sofa source code

Comment on topic or styleThere are three ways to get the Sofa source code - each of them are equally valid, it's just a matter of personal preference, and how you plan to use the code once you have it. The easiest way is to use CouchApp to clone it from a running instance. If you didn't install CouchApp in the previous section, you can read the source code (but not install and run it) by downloading and extracting zip or tar file. If you are interested in hacking on Sofa, and would like to join the development community, the best way to get the source code is from the official Git repository. We'll cover these three methods in turn.

Bird Break: A happy bird to ease any install-induced frustration

CouchApp Clone

Comment on topic or styleOne of the easiest ways to get the Sofa source code is by cloning directly from Chris's blog using CouchApp's `clone` command to download Sofa's design document to a collection of files on your local harddrive. The clone command operates on a design document URL, which can be hosted in any CouchDB database accessible via HTTP. To clone Sofa from the version running Chris's blog, run the following command:

```
couchapp clone http://jchrisa.net/drl/_design/sofa
```

Comment on topic or styleYou should see this output:

```
[INFO] Cloning sofa to sofa...
```

Comment on topic or styleNow that you’ve got Sofa on your local filesystem, you can skip to the bottom of the chapter to make a small local change and push it to your own CouchDB.

Zip and Tar files

Comment on topic or styleIf you merely want to peruse the source code while reading along with the book, it is available as standard zip or tar downloads. To get the zip version access the following URL from your browser, which will redirect to the latest zip file of Sofa. If you prefer, a tar file is available as well.

Join the Sofa Development Community on Github

Comment on topic or styleThe most up-to-date version of Sofa will always be available at its public code repository. If you are interested in staying up to date with development efforts, and contributing patches back to the source, the best way to do it is via Git and Github.

Comment on topic or styleGit is a form of distributed version control - it allows groups of developers to track and share changes to software. If you are familiar with Git, you’ll have no trouble using it to work on Sofa. If you’ve never used Git before, it has a bit of a learning curve, so depending on your tolerance for new software, you might want to save learning Git for another day - or you might want to dive in head first! For more information about Git, and how to install it, see the official Git home page. For other hints and help using Git, see the Github guides.

Comment on topic or styleTo get Sofa (including all development history) using Git, run the following command.

```
git clone git://github.com/jchris/sofa.git
```

Comment on topic or styleIf all goes well you should see output like this:

```
Initialized empty Git repository in /Users/me/sofa/.git/
remote: Counting objects: 1103, done.
remote: Compressing objects: 100% (663/663), done.
remote: Total 1103 (delta 598), reused 658 (delta 360)
Receiving objects: 100% (1103/1103), 142.48 KiB, done.
Resolving deltas: 100% (598/598), done.
```

Comment on topic or styleNow that you’ve got the source, lets take a quick tour.

The Sofa Source Tree

Comment on topic or styleOnce you’ve succeeded with any of these methods, you’ll have a copy of Sofa on your local disk. The following text is generated by running the `tree` command on the Sofa directory, to reveal the full set of files it contains. It is annotated inline to make it clear how various files and directories correspond to the Sofa design document.

```
sofa/
|-- README.md
|-- THANKS.txt
```

Comment on topic or styleThe source tree contains some files which aren’t necessary for the application - the README and THANKS files are among those.

```
|-- _attachments
```

```
|   |-- LICENSE.txt
|   |-- account.html
|   |-- blog.js
|   |-- jquery.scrollTo.js
|   |-- md5.js
|   |-- screen.css
|   |-- showdown-licenese.txt
|   |-- showdown.js
|   |-- tests.js
|   `-- textile.js
```

Comment on topic or styleThe `_attachments` directory contains files which are saved to the Sofa design document as binary attachments. CouchDB serves attachments directly (instead of including them in a JSON wrapper) so this is where we store JavaScript, CSS, and HTML files that the browser will access directly.

```
|-- blog.json
```

Making your first edit to the Sofa source code will show you how easy it is to modify the application.

Comment on topic or styleThe `blog.json` file contains JSON used to configure individual installations of Sofa. Currently it sets one value, the title of the blog. You should open this file now and personalize the title field - you probably don't want to name your blog "Daytime Running Lights", now's your chance to come up with something more fun!

Comment on topic or styleYou could add other blog configuration to this file, maybe things like how many posts to show per page, and a URL to an about page for the author. Working changes like these into the application will be easy once you've walked through the following chapters.

```
|-- couchapp.json
```

Comment on topic or styleWe'll see later that `couchapp` outputs a link to Sofa's home page when `couchapp push` is run. The way this works is pretty simple - CouchApp looks for an JSON field on the design document, at the address `design_doc.couchapp.index`, if it finds it, it appends the value to the location of the design doc itself to build the URL. If there is no CouchApp index specified, but the design document has an attachment called `index.html`, then it is considered the index page. In Sofa's case we use the index value to point to a list of the most recent posts.

```
|-- helpers
|   `-- md5.js
```

Comment on topic or styleThe `helpers` directory here is just an arbitrary choice - CouchApp will push any files and folders to the design document - in this case the source code to md5.js is JSON encoded and stored on the `design_document.helpers.md5` element.

```
|-- lists
|   `-- index.js
```

Comment on topic or styleThe `lists` directory contains a JavaScript function that will be executed by CouchDB to render view rows as Sofa's HTML and Atom indexes. You could add new list functions by creating new files within this directory. Lists are covered in depth in the Viewing Lists of Blog Posts Chapter.

```
|-- shows
|   |-- edit.js
|   `-- post.js
```

Comment on topic or styleThe `shows` directory holds the functions CouchDB uses to generate HTML views of blog posts. There are two views, one for reading posts, and the other for editing. We'll look at these functions in the next few chapters.

```
|-- templates
|   |-- edit.html
|   |-- index
|   |   |-- head.html
|   |   |-- row.html
|   |   `-- tail.html
|   `-- post.html
```

Comment on topic or styleThe `templates` directory is like the `helpers` directory, and unlike the `lists`, `shows`, or `views` directory, in that the code stored this is not directly executed on CouchDB's server side. Instead, the templates are included into the body of the `list` and `show` functions using macros run by CouchApp when pushing code to the server. These CouchApp macros are covered later in this chapter. The key point is that the `templates` name could be anything - it is not a special member of the design document, just a convenient place to store and edit our template files.

```
|-- validate_doc_update.js
```

Comment on topic or styleThis file corresponds to the JavaScript validation function used by Sofa to ensure that only the blog owner can create new posts, as well as ensuring the comments are well formed. Sofa's validation function is covered in detail in Storing Documents.

```
|-- vendor
|   `-- couchapp
|       |-- README.md
|       |-- _attachments
|       |   `-- jquery.couchapp.js
|       |-- couchapp.js
|       |-- date.js
|       |-- path.js
|       `-- template.js
```

Comment on topic or styleThe `vendor` directory holds code that is managed independently of the Sofa application itself. In Sofa's case the only vendor package used is `couchapp`, which contains JavaScript code that knows how to do things like link between *list* and *show* URLs and render templates.

Comment on topic or styleDuring `couchapp push` files within a `vendor/**/_attachments/*` path are pushed as design document attachments. In this case `jquery.couchapp.js` will be pushed to an attachment called `couchapp/jquery.couchapp.js` (so that multiple vendor packages can have the same attachment names without worry of collisions.)

```
`-- views
    |-- comments
    |   |-- map.js
    |   `-- reduce.js
    |-- recent-posts
    |   `-- map.js
    `-- tags
        |-- map.js
        `-- reduce.js
```

Comment on topic or styleThe `views` directory holds Map Reduce view definitions, with each view represented as a directory, holding files corresponding to map and reduce functions.

Deploying Sofa

Comment on topic or styleThe source code is safely on your hard drive, and you've even been able to make minor edits to the `blog.json` file. Now it's time to deploy the blog to a local CouchDB. The `push` command is very simple, and should work the first time, but there are two other steps involved in setting up an admin account on your CouchDB, and for your CouchApp deployments. By the end of this chapter you'll have your own running copy of Sofa.

Pushing Sofa to Your CouchDB

Comment on topic or styleAnytime you make edits to the on-disk version of Sofa, and want to see them in your browser, run the following command:

```
couchapp push . sofa
```

Comment on topic or styleThis deploys the Sofa source code into CouchDB. You should see output like this:

```
[INFO] Pushing CouchApp in /Users/jchris/sofa to design doc:
http://127.0.0.1:5984/sofa/_design/sofa
[INFO] Visit your CouchApp here:
http://127.0.0.1:5984/sofa/_design/sofa/_list/index/recent-posts?
descending=true&limit=5
```

Comment on topic or styleIf you get an error, make sure your target CouchDB instance is running by making a simple HTTP request to it:

```
curl http://127.0.0.1:5984
```

Comment on topic or styleThe response should look a lot like:

```
{"couchdb":"Welcome","version":"0.10.0"}
```

Comment on topic or styleIf CouchDB is not running yet, go back to the Getting Started chapter and follow the hello world instructions there.

Visit the Application

Comment on topic or styleIf CouchDB was running, then `couchapp push` should have directed you to visit the application's index URL. Visting the URL should show you something like this:

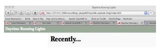


Figure: Empty index page

Comment on topic or styleWe're not done yet - there are a couple of steps remaining before you've got a fully functional Sofa instance.

Setup Your Admin Account

Comment on topic or styleSofa is a single-user application. You, the author, are the administrator and the only one who can add and edit posts. To make sure no one else goes in and messes with your writing, you must create an administrator account in CouchDB. This is a straightforward task. Find your `local.ini` file and open it in your text editor. (By default it's stored at `/usr/local/etc/couchdb/local.ini`) If you haven't already, uncomment the `[admins]` section at the end of the file. Next, add a line right below the `[admins]` section with your preferred username and password.

```
[admins]
jchris = secretpass
```

Comment on topic or styleNow that you've edited your `local.ini` configuration file, you need to restart CouchDB for changes to take effect. Depending on how you started CouchDB, there are different methods of restarting it. If you started in a console, then hitting control-c, and rerunning the same command you used to start it is the simplest way.

Comment on topic or styleIf you don't like your passwords lying around in plain-text files, don't worry. When CouchDB starts up and reads this file, it takes your password and changes it to a secure hash, like this:

```
[admins]
jchris = -hashed-207b1b4f8434dc604206c2c0c2aa3aae61568d6c,964 \
  06178007181395cb72cb4e8f2e66e
```

Comment on topic or styleCouchDB will now ask you for your credentials when you try to create databases or change documents; exactly the things you want to keep to yourself.

Deploying to a Secure CouchDB

Comment on topic or styleNow that we've set up admin credentials, we'll need to supply them on the command line when running `couchapp push`. Let's try it:

```
couchapp push . http://jchris:secretpass@localhost:5984/sofa
```

Comment on topic or styleMake sure to replace `jchris` and `secretpass` with your actual values or you will get a permission denied error. If all works according to plan everything is set up in CouchDB and you should be able to start using your blog.

Comment on topic or styleAt this point we are technically ready to move on, but you'll be much happier if you make use of the `.couchapprc` file as documented in the next section.

Configuring CouchApp with `.couchapprc`

Comment on topic or styleIf you don't want to have to put the full URL (potentially including authentication parameters) to your database onto the command line each time you push, you can use the `.couchapprc` file to store deployment settings. The contents of this file are not pushed along with the rest of the app, so it can be a safe place to keep credentials for uploading your app to secure servers.

Comment on topic or styleThe `.couchapprc` file lives in the source directory of your application, so you should look to see if it is at `/path/to/the/directory/of/sofa/.couchapprc` (or create it there if it is missing). Dotfiles (files with names that start with a period) are left out of most directory listings. Use whatever tricks your OS has to "show hidden files" - the simplest one in a standard command shell is to list the directory using `ls -a` which will show all hidden files as well as normal files.

An Example `.couchapprc`

```
{
  "env": {
    "default": {
      "db": "http://jchris:secretpass@localhost:5984/sofa"
    },
    "staging": {
      "db": "http://jchris:secretpass@jchrisa.net:5984/sofa-staging"
    },
    "drl": {
      "db": "http://jchris:secretpass@jchrisa.net/drl"
    }
  }
}
```

Comment on topic or styleWith this file set up, you can push your CouchApp with the command `couchapp push`, which will push the application to the "default" database. CouchApp also supports alternate environments. To push your application to a development database, you could use `couchapp push dev`. In our experience, taking the time to setup a good `.couchapprc` is always worth it. Another benefit is that it keeps your passwords off the screen when you are working.

Storing Documents

Comment on topic or style*Documents* are CouchDB’s central data structure. To best understand and use CouchDB, you need to *think in documents*. This chapter walks you though the lifecycle of designing and saving a document. We’ll follow up by reading documents and aggregating and querying them with views. In the next section, you’ll see how CouchDB can also transform documents into other formats.

Comment on topic or styleDocuments are self-contained units of data. You might have heard the term *record* to describe something similar. Your data is usually made up of small native types such as integers and strings. Documents are the first level of abstraction over these native types. They provide some structure and logically group the primitive data. The height of a person might be encoded as an integer (`176`), but this integer is usually part of a larger structure that contains a label (`"height": 176`) and related data (`{"name":"Chris", "height": 176}`).

Comment on topic or styleHow many data items you put into your documents depends on your application and a bit on how you want to use views (later), but generally, a document roughly corresponds to an object instance in your programming language. Are you running an online shop? You will have *items* and *sales* and *comments* for your items. They all make good candidates for objects and subsequently documents.

Comment on topic or styleDocuments differ subtly from garden-variety objects, in that they usually have authors, and CRUD operations. Document-based software (like the word-processors and spreadsheets of yore) builds its storage model around saving documents, so that authors get back what they created. Similarly, in a CouchDB application, you may find yourself giving greater leeway to the presentation layer. If instead of adding timestamps to your data in a controller, you allow the user to control them, you get draft status and the ability to publish articles in the future for free. (By viewing published docs using an endkey of *now*.)

Comment on topic or styleDocument integrity… Validation functions are available so that you don't have to worry about bad data causing errors in your system. Often in document-based software, the client application edits and manipulates the data, saving it back. As long as you give the user the document they asked you to save, they'll be happy.

Comment on topic or styleSay your users can comment on the item ("lovely book"); you have the option to store the comments as an array, on the item-document. This makes it trivial to find the item's comments, but, as they say, "it doesn't scale". A popular item could have tens of comments, or even hundreds, or more.

Comment on topic or styleInstead of storing a list on the item-document, in this case it may be better to model comments into a collection of documents. There are patterns for accessing collections, which CouchDB makes easy. You likely only want to show ten or twenty at a time, and provide "previous" and "next" links. By handling comments as individual entities, you can group them with views. A group could be the entire collection or slices of ten or twenty, sorted by the item they apply to, so it's easy to grab the set you need.

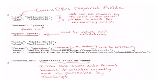
Comment on topic or styleA rule of thumb: Break up into documents everything that you will be handling separately in your application. Items are single, and comments are single, but you don't need to break them into smaller pieces. Views are a convenient way to group your documents in meaningful ways.

Comment on topic or styleLet's go through building our example application to show you in practice how to work with documents.

JSON Document Format

Comment on topic or styleThe first step in designing any application (once you know what the program is *for* and have the user-interaction nailed down) is deciding on the format it will use to represent and store data. Our example blog is written in JavaScript. A few lines back we said documents roughly represent your data objects, in this case there is a an exact correspondence. CouchDB borrowed the JSON data format from JavaScript; this allows us to directly use documents as native objects when programming. This is really convenient and leads to fewer problems down the road (if you ever worked with an ORM system, you might know what we are hinting at).

Comment on topic or styleLet's draft a JSON format for blog posts. We know we'll need each post to have an author, a title, and a body. We know we'd like to use document ids to find documents, so URLs are search-engine friendly, and that we'd also like to list them by creation date.



```
{
  "_id": "hello_world",
  "_rev": "1766895818",

  "author": "jchris",

  "type": "post",

  "title": "Hello World",
  "slug": "hello_world",

  "format": "markdown",
  "body": "Hello from [CouchDB](http://couchdb.org).",
  "html": "<p>Hello from <a href=\"http://couchdb.org\">CouchDB</a></p>",

  "created_at": "2008/11/23 17:52:50 +0000"
}
```

CouchDB's required fields
id can be chosen by the client or the server.
rev is used for concurrency control.
thats me!
used by views and validations
Compose in Markdown, convert to HTML
I like this JSON date format because it collates correctly and is parseable by JavaScript

The JSON Post Format
Comment on topic or styleIt should be pretty straightforward to see how JSON works. Curly braces (`{}`) wrap objects and objects are key-value lists. Keys are strings that are wrapped in double quotes (`""`) Finally, a value is a string, an integer, an object, or an array (`[]`). Keys and values are separated by a colon (`:`) and multiple keys and values by comma (`,`). That's it. For a complete description of the

JSON format see the JSON Primer appendix.

Comment on topic or styleFigure 12-1 shows a document that meets our requirements. The cool thing is: We just made it up on the spot. We didn't go and define a schema, we didn't prescribe how things should look like. We just created a document with whatever we just need. Now, requirements for objects change all the time during the development of an application. Coming up with a different document that meets new, evolved needs is just as easy.

Comment on topic or styleDo I really look like a guy with a plan? You know what I am? I'm a dog chasing cars. I wouldn't know what to do with one if I caught it. You know, I just… do things. The mob has plans, the cops have plans, Gordon's got plans. You know, they're schemers. Schemers trying to control their little worlds. I'm not a schemer. I try to show the schemers how pathetic their attempts to control things really are.

— *The Joker, The Dark Knight*

Comment on topic or styleLet's examine the document in a little more detail. The first two members (`_id` and `_rev`) are for CouchDB's housekeeping and act as identification for a particular *instance* of a document. `_id` is easy: If I store something in CouchDB, it creates the `_id` and returns it to me. I can use the `_id` to build the URL where I can get my something back.

Comment on topic or styleYour document's `_id` defines the URL the document can be found under. Say you have a database `movies`. All documents can be found somewhere under the URL `/movies`, but where exactly?

Comment on topic or styleIf you store a document with the `_id` `Jabberwocky` (`{"_id":"Jabberwocky"}`) into your `movies` database, it will be available under the URL `/movies/Jabberwocky`. So if you send a GET request to `/movies/Jabberwocky`, you will get back the JSON that makes up your document (`{"_id":"Jabberwocky"}`).

Comment on topic or styleThe `_rev` (or *revision id*) describes a version of a document. Each change creates a new document version (that again is self-contained), and updates the `_rev`. This becomes useful because when saving a document, you must provide an up to date `_rev`, so that CouchDB knows you've been working against the latest document version.

Comment on topic or styleWe touched on this in the Eventual Consistency chapter. The revision id acts as a gatekeeper for writes to a document in CouchDB's MVCC system. A document is a shared resource, many clients can read and write them at the same time. To make sure two writing clients don't step on each others feet, each client must provide what it believes is the latest revision id of a document along with the proposed changes. If the on-disk revision id matches the provided `_rev`, CouchDB will accept the change. If it doesn't, the update will be rejected. The client should read the latest version, integrate his changes and try saving again.

Comment on topic or styleThis mechanism ensures two things: A client can only overwrite a version it knows, and it can't trip over changes made by other clients. This works without CouchDB having to manage explicit *locks* on any document. This ensures that no client has to wait for another client to complete any work. Updates are serialized, so CouchDB will never attempt to write documents faster than your disk can spin, and it also means that two mutually conflicting writes can't be written at the same time.

Beyond Id and Rev: Your Document Data

Comment on topic or styleNow that you thoroughly understand the role of `_id` and `_rev` on a document, let's look at everything else we're storing.

```
{
  "_id":"Hello-Sofa",
  "_rev":"2-2143609722",
  "type":"post",
```

Comment on topic or styleThe first thing is the type of the document. Note that this is an application-level parameter, not anything particular to CouchDB. The type is just an arbitrarily named key-value pair as far as Couch is concerned. For us, as we're adding blog posts to Sofa, it has a little deeper meaning. Sofa uses the `type` field to determine which validations to apply. It can then rely on documents of that type being valid in the views and the user interface. This removes the need to check for every field and nested JSON value before using it. This is purely by convention and you can make up your own, or you can infer the type of a document by its structure ("has an array with three elements" — a.k.a. *ducktyping*), we just thought this is easy to follow and we hope you agree.

```
  "author":"jchris",
  "title":"Hello Sofa",
```

Comment on topic or styleThe author and title fields are set when the post is created. The title field can be changed, but the author field is locked by the validation function for security. Only the author may edit the post.

```
  "tags":["example","blog post","json"],
```

Comment on topic or styleSofa's tag system just stores them as an array on the document. This kind of denormalization is a particularly good fit for CouchDB.

```
  "format":"markdown",
  "body":"some markdown text",
  "html":"
Comment on topic or stylethe html text",
```

Comment on topic or styleBlog posts are composed in the Markdown HTML format to make them easy to author. The Markdown format as typed by the user is stored in the `body` field. Before the blog post is saved, Sofa converts it to HTML in the client's browser. There is an interface for previewing the Markdown conversion, so you can be sure it will display as you like.

```
  "created_at":"2009/05/25 06:10:40 +0000"
}
```

Comment on topic or styleThe `created_at` field is used to order blog post in the Atom feed and on the HTML index page.

The Edit Page

Comment on topic or styleThe first page we need to build, in order to get one of these blog entries into our post, is the interface for creating and editing posts.

Comment on topic or styleEditing is more complex than just rendering posts for visitors to read, but that means once you've read this chapter, you'll have seen most of the techniques we touch in the other chapters.

Comment on topic or styleThe first thing to look at is the *show function* used to render the HTML page. If you haven't already, read the Show Functions Chapter to learn about the details of the API. We'll just look at this code in the context of Sofa, so you can see how it all fits together.

```
function(doc, req) {
  // !json templates.edit
  // !json blog
  // !code vendor/couchapp/path.js
  // !code vendor/couchapp/template.js
```

Comment on topic or styleSofa's edit page show function is very straightforward. In the above section, we're just important the templates and libraries we'll use. The important line is the `!json` macro which loads the `edit.html` template from the templates directory. These macros are run by CouchApp, as Sofa is being deployed to CouchDB. For more information about the macros see the Shows chatper.

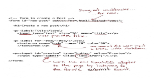
```
  // we only show html
  return template(templates.edit, {
    doc : doc,
    docid : toJSON((doc && doc._id) || null),
    blog : blog,
    assets : assetPath(),
    index : listPath('index','recent-posts',{descending:true,limit:8})
  });
}
```

Comment on topic or styleThe rest of the function is simple. We're just rendering the HTML template with data culled from the document. In the case where the document does not yet exist, we make sure to set the docid to null. This allows us to use the same template both for creating new blog posts as well as editing existing ones.

The HTML Scaffold

Comment on topic or styleThe only missing piece of this puzzle is the HTML that it takes to save a document like this.

Comment on topic or styleIn your browser, visit `http://127.0.0.1:5984/blog/_design/sofa/_show/edit` and using your text editor, open the source file `templates/edit.html` (or view source in your browser). Everything is ready to go, all we have to do is wire up CouchDB using in-page JavaScript.



```
<!-- form to create a Post -->
<form id="new-post" action="new.html" method="post">
  <h1>Create a new post</h1>
  <p><label>Title</label>
    <input type="text" size="50" name="title"></p>
  <p><label for="body">Body</label>
    <textarea name="body" rows="28" cols="80">
    </textarea></p>
  <p><input id="preview" type="button" value="Preview"/>
    <input type="submit" value="Save &rarr;"/></p>
</form>
```

Sorry, not unobtrusive... ..for now.
user provides title
we convert the user input to HTML with Markdown.
We'll tie our CouchDB adapter to the page by listening to the form's submit event.

Figure: HTML listing for edit.html

Comment on topic or styleJust like any web application, the important part of the HTML is the form for accepting edits. The edit form captures a few basic data items: the post title, the body (in Markdown format), and any tags the author would like to apply.

Create a new post

Comment on topic or styleTitle

Comment on topic or styleBody

Comment on topic or stylePreview

Save →

Comment on topic or styleWe start with just a raw HTML document, containing a normal HTML form. We use JavaScript to convert user input into a JSON document and save it to CouchDB. In the spirit of focusing on CouchDB, we won’t dwell on the JavaScript here. It’s a combination of Sofa-specific application code, CouchApp’s JavaScript helpers, and jQuery for interface elements. The basic story is that it watches for the user to click "Save", and then applies some callbacks to the document before sending it to CouchDB.

Saving a Document

Comment on topic or styleThe JavaScript that drives blog post creation and editing centers around the

HTML form from the previous figure. The CouchApp jQuery plugin provides some abstraction, so we don’t have to concern ourselves with the details of how the form is converted to a JSON document when the user hits the submit button. `$.CouchApp` also ensures that the user is logged in, and makes their information available to the application.

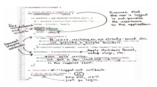


Figure: JavaScript callbacks for edit.html

```
$.CouchApp(function(app) {
  app.loggedInNow(function(login) {
```

Comment on topic or styleThe first thing we do is ask the CouchApp library to make sure the user is logged in. Assuming the answer is yes, we’ll proceed to set up the page as an editor. This means we apply JavaScript event handler to the form, and specify callbacks we’d like to run on the document, both when it is loaded and when it saved.

```
    // w00t, we're logged in (according to the cookie)
    $("#header").prepend(''+login+'');
    // setup CouchApp document/form system, adding app-specific callbacks
    var B = new Blog(app);
```

Comment on topic or styleNow that we know the user is logged in, we can render their user name at the top of the page. The variable `B` is just a shortcut to some of the Sofa-specific blog rendering code. It contains methods for converting blog post bodies from Markdown to HTML, as well as a few other odds and ends. We pulled these functions into `blog.js` so we could keep them out of the way of main code.

```
    var postForm = app.docForm("form#new-post", {
      id : <%= docid %>,
      fields : ["title", "body", "tags"],
      template : {
        type : "post",
        format : "markdown",
        author : login
      },
```

Comment on topic or styleCouchApp’s `app.docForm()` helper is a function to setup and maintain a correspondence between a CouchDB document and an HTML form. Let’s look at the first three arguments passed to it by Sofa. The `id` argument tells `docForm()` where to save the document. This can be null in the case of a new document. We set `fields` to an array of form elements which will correspond directly to JSON fields in the CouchDB document. Finally, the `template` argument is given a JavaScript object which will be used as the starting point, in the case of a new document. In this case we ensure that the document has a type equal to "post", and that the default format is Markdown. We also set the author to be the login name of the current user.

```
      onLoad : function(doc) {
        if (doc._id) {
          B.editing(doc._id);
          $('h1').html('Editing '+doc._id+'');
          $('#preview').before(' ');
          $("#delete").click(function() {
            postForm.deleteDoc({
              success: function(resp) {
                $("h1").text("Deleted "+resp.id);
                $('form#new-post input').attr('disabled', true);
              }
            });
            return false;
          });
        }
        $('label[for=body]').append(' with '+(doc.format||'html')+'');
```

Comment on topic or styleThe `onLoad` callback is run when the document is loaded from CouchDB. It is useful for decorating the document before passing it to the form, or for setting up other user interface elements. In this case we check to see if the document already has an id. If it does, that means it’s been saved, so we create a button for deleting it, and setup the callback to the delete function. It may look like a lot of code, but it’s pretty standard for Ajax applications. If there is one criticism to make of this section, it’s that the logic for creating the delete button could be moved to the `blog.js` file, so we can keep more user-interface details out of the main flow.

```
    },
    beforeSave : function(doc) {
      doc.html = B.formatBody(doc.body, doc.format);
      if (!doc.created_at) {
        doc.created_at = new Date();
      }
      if (!doc.slug) {
        doc.slug = app.slugifyString(doc.title);
        doc._id = doc.slug;
      }
      if(doc.tags) {
        doc.tags = doc.tags.split(",");
        for(var idx in doc.tags) {
          doc.tags[idx] = $.trim(doc.tags[idx]);
        }
      }
    },
```

Comment on topic or styleThe `beforeSave()` callback to `docForm` is run after the user clicks the submit button. In Sofa’s case it manages setting the blog post’s timestamp, tranforming the title into an acceptable document id (for prettier URLs), and processing the documents tags from a string into an array. It also runs the Markdown to HTML conversion in the browser, so that once the document is saved, the rest of the application has direct access to the HTML.

```
    success : function(resp) {
      $("#saved").text("Saved _rev: "+resp.rev).fadeIn(500).fadeOut(3000);
      B.editing(resp.id);
    }
  });
```

Comment on topic or styleThe last callback we use in Sofa is the `success` callback. It is fired when the document is successfully saved. In our case we use it to flash a message to the user letting them know they’ve succeeded, as well as to add a link to the blog post, so that when you create a blog post for the first time you can click through to see its permalink page.

Comment on topic or styleThat’s it for the `docForm()` callbacks.

```
  $("#preview").click(function() {
    var doc = postForm.localDoc();
    var html = B.formatBody(doc.body, doc.format);
```

```
      $('#show-preview').html(html);
      // scroll down
      $('body').scrollTo('#show-preview', {duration: 500});
    });
```

Comment on topic or styleSofa has a function to preview blog posts before saving them. Since this doesn’t effect how the document is saved, the code that watches for events from the "preview" button is not applied within the `docForm()` callbacks.

```
  }, function() {
    app.go('<%= assets %>/account.html#'+document.location);
  });
});
```

Comment on topic or styleThe last bit of code here is triggered when the user is not logged in. All it does is redirect them to the account page, so they can log in and be sent back to try editing again.

Validation

Comment on topic or styleHopefully you can see how the above code will send a JSON document to CouchDB when the user clicks save. That’s great for creating a user interface, but it does nothing to protect the database from unwanted updates. This is where validation functions come into play. With a proper validation function, even a determined hacker cannot get unwanted documents into your database. Let’s look at how Sofa’s works. For more depth on validation functions, see Validation Functions.

```
function (newDoc, oldDoc, userCtx) {
  // !code lib/validate.js
```

Comment on topic or styleThis line imports a library from Sofa which makes the rest of the function much more readable. It is just a wrapper around the basic ability to mark requests as either `forbidden` or `unauthorized`. In this chapter we’ll concentrate on the business logic of the validation function, just be aware that unless you use Sofa’s `validate.js` you’ll need to work with the more primitive logic that the library abstracts.

```
  unchanged("type");
  unchanged("author");
  unchanged("created_at");
```

Comment on topic or styleThese lines do just what they say. If the document’s `type`, `author`, or `created_at` field is changed, they throw an error saying the update is forbidden. Note that they make no assumptions about the content of these fields. They merely state that updates must not change the content from one revision of the document to the next.

```
  if (newDoc.created_at) dateFormat("created_at");
```

Comment on topic or styleThe `dateFormat` helper makes sure that the date (if one is provided) is in the format that Sofa’s views expect.

```
  // docs with authors can only be saved by their author
  // admin can author anything...
  if (!isAdmin(userCtx) && newDoc.author && newDoc.author != userCtx.name) {
      unauthorized("Only "+newDoc.author+" may edit this document.");
  }
```

Comment on topic or styleIf the person saving the document is an admin, let the edit proceed. Otherwise, make certain that the author and the person saving the document are the same. This ensure that authors may only edit their own posts.

```
  // authors and admins can always delete
  if (newDoc._deleted) return true;
```

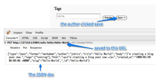
Comment on topic or styleThe next block of code will check the validity of various types of documents. However, deletions will normally not be valid according to those specifications, because their content is just `_deleted: true`, so we short-circut the validation function here.

```
  if (newDoc.type == 'post') {
    require("created_at", "author", "body", "html", "format", "title", "slug");
    assert(newDoc.slug == newDoc._id, "Post slugs must be used as the _id.")
  }
}
```

Comment on topic or styleFinally, we have the validation for the actual Post document itself. Here we require the fields that are particular to the post document. Because we've validated that they are present, we can count on them in views and user interface code.

Save your first post

Comment on topic or styleLet's see how this all works together! Fill out the form with some practice data, and hit "Save" to see a success response.



```
PUT http://127.0.0.1:5984/sofa-hello/Hello-World 201 Created 64ms
```

```
{"type":"post","format":"markdown","author":"jchris","title":"Hello World!","body":"I'm creating a blog post now.","tags":["testing"],"html":"<p>I'm creating a blog post now.</p>","created_at":"2009/01/20 06:09:06 +0000","slug":"Hello-World","_id":"Hello-World"}
```

Screenshot: JSON over HTTP to save the blog post

Comment on topic or styleThe figure shows how JavaScript has used HTTP to *PUT* the document to a URL, constructed of the database name plus the document id. It also shows how the document is just sent as a JSON string in the body of the PUT request. If you were to *GET* the document URL, you'd see the same set of JSON data, with the addition of the `_rev` parameter as applied by CouchDB.

Comment on topic or styleTo see the JSON version of the document you've saved, you can also browse to it in Futon. Visit `http://127.0.0.1:5984/_utils/database.html?blog/_all_docs` and you should see a document with an id corresponding to the one you just saved. Click it to see what Sofa is sending to CouchDB.

Wrapping Up

Comment on topic or styleWe've covered how to design JSON formats for your application, how to enforce those designs with validation functions, the basics of how documents are saved, and maybe more than you wanted to know about the B-tree internals. In the next chapter we'll show how to load documents from CouchDB and display them in the browser.

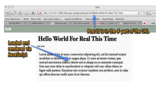
Showing Documents in Custom Formats

Comment on topic or styleCouchDB's show functions are a RESTful API inspired by a similar feature in Lotus Notes. In a nutshell, they allow you to serve documents to clients, in any format you choose.

Comment on topic or styleA show function builds an HTTP response with any Content-Type, based on a stored JSON document. For Sofa, we'll use them to show the blog post permalink pages. This will ensure that these pages are indexable by search engines, as well as make the pages more accessible.

Comment on topic or styleSofa's show function displays each blog post as an HTML page, with links to stylesheets and other assets, which are stored as attachments to Sofa's design doc.

Comment on topic or styleHey, this is great, we've rendered a blog post!


Sofa
http://127.0.0.1:5984/sofa-ajax/_design/sofa/post.html#Hello-World-For-Real-This-Time
Post id is in the # part of the URL
Loaded and rendered via JavaScript

Hello World For Real This Time

just now

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

When the document is successfully loaded and rendered, it will look something like this screenshot.A Rendered Post

Comment on topic or styleThe complete show function and template will render a static, cacheable resource, that does not depend on details about the current user, or anything else aside from the requested document and Content-Type. Generating HTML from a show function can not cause any side effects in the database, which has positive implications for building simple scalable applications.

Rendering Documents With Show Functions

Comment on topic or styleLet's look at the source code. The first thing we'll see is the JavaScript function body, which is very simple - it simply runs a template function to generate the HTML page. Let's break it down:

```
function(doc, req) {
  // !json templates.post
  // !json blog
  // !code vendor/couchapp/template.js
  // !code vendor/couchapp/path.js
```

Comment on topic or styleWe're familiar with the `!code` and `!json` macros from the Managing Design Documents chapter. In this case, we're using them to import a template and some metadata about the blog (as JSON data), as well as to include link and template rendering functions as inline code.

Comment on topic or styleNext we render the template:

```
  return template(templates.post, {
    title : doc.title,
    blogName : blog.title,
    post : doc.html,
    date : doc.created_at,
    author : doc.author,
```

Comment on topic or styleThe blog post title, html body, author, and date are taken from the document, with the blog's title included from it's JSON value. The next three calls all use the `path.js` library to generate links based on the request path. This ensures that links within the application are correct.

```
    assets : assetPath(),
    editPostPath : showPath('edit', doc._id),
    index : listPath('index','recent-posts',{descending:true, limit:5})
  });
}
```

Comment on topic or styleSo we’ve seen that that function body itself just calculates some values (based on the document, the request, and some deployment specifics, like the name of the database) to send to the template for rendering. The real action is in the HTML template. Let’s take a look.

The Post Page Template

Comment on topic or styleThe template defines the output HTML, with the exception of a few tags which are replaced with dynamic content. In Sofa’s case, the dynamic tags look like `<%= replace_me %>`, which is a common templating tag delimitor.

Comment on topic or styleThe tempate engine used by Sofa is adapted from John Resig’s blog post JavaScript Micro-Templating. It was chosen as the simplest one that worked in the server-side context without modification. Using a different template engine would be a simple exercise.

Comment on topic or styleLet’s look at the template string. Remember that it is included in the JavaScript using the CouchApp `!json` macro, so that CouchApp can handle escaping it and including it to be used by the templating engine.

Comment on topic or styleThis is the first time we’ve seen a template tag in action - the blog post title, as well as the name of the blog as defined in `blog.json` are both used to craft the HTML `<title>` tag.

Comment on topic or styleBecause show functions are served from within the design document path, we can link to attachments on the design document using relative URIs. Here we’re linking to `screen.css`, a file stored in the `_attachments` folder of the Sofa source directory.

`Edit this post`

`<%= blogName %>`

Comment on topic or styleAgain we’re seeing template tags used to replace content. In this case we link to the edit page for this post, as well as linking to the index page of the blog.

<%= title %>

<%= date %>

Comment on topic or styleThe post title is used for the `<h1>` tag, and the date is rendered in a special tag with a class of `date`. See the last section of this chapter, Dynamic Dates, for an explanation of why we output static dates in the html, instead of rendering a user friendly string like "3 days ago" to describe the date.

```
<%= post %>
```

Comment on topic or styleIn the close of the template, we render the post HTML (as converted from Markdown and saved from the author's browser.)

Dynamic Dates

Comment on topic or styleWhen running CouchDB behind a caching proxy, this means each show function should only have to be rendered once per updated document. However, it also explains why the timestamp looks like `2008/12/25 23:27:17 +0000` instead of "9 days ago".

Comment on topic or styleIt also means that for presentation items that depends on the current time, or the identity of the browsing user, we'll need to use client-side JavaScript to make dynamic changes to the final HTML.

Dynamic Dates

```
    $('.date').each(function() {
      $(this).text(app.prettyDate(this.innerHTML));
    });
```

Comment on topic or styleWe include this detail about the browser-side JavaScript implementation, not to teach you about Ajax, but because it epitomizes the kind of thinking that makes sense when you are presenting documents to client applications. CouchDB should provide the most useful format for the document, as request by the client, but when it comes time to integrate information from other queries, or bring the display up-to-date with other web services, by asking the client's application to do the lifting, you move computing cycles and memory costs from CouchDB to the client. Since there are typically many more clients than CouchDBs, pushing the load back to the clients means each CouchDB can serve more users.

Viewing Lists of Blog Posts

Comment on topic or styleThe last few chapters dealt with getting data into and out of CouchDB. You learned how to model your data into documents, and retrieve it via the HTTP API. In this chapter we'll look at the views used to power Sofa's index page, and the list function which renders those views as HTML or XML depending on the client's request.

Comment on topic or styleNow that we've successfully created a blog post and rendered it as HTML, we'll be building the front page where visitors will land when they've found your blog. This page will have a list of the ten most recent blog posts, with titles and short summaries. The first step here is to write the map reduce query which constructs the index used by CouchDB at query time, to find blog posts based on when they were written.

Comment on topic or styleIn the Finding Your Data with Views chapter we noted that reduce isn't needed for many common queries. For the index page we're only interested in an ordering of the posts by date, so we don't need to use a reduce function, as the map function alone is enough to order the posts by date.

Map of Recent Blog Posts

Comment on topic or styleYou're now ready to write the map function that builds a list of all blog posts. The goals for this view are simple: sort all blog posts by date.

Comment on topic or styleHere is the source code for the view function. I'll call out the important bits as we encounter them.

```
function(doc) {
  if (doc.type == "post") {
```

Comment on topic or styleThe first thing we do is ensure that the document we're dealing with is a post. We don't want comments or anything other than blog posts getting on the front page. The expression `doc.type == "post"` evaluates to true for posts but no other kind of document. In the Validation Functions chapter we saw that the validation function gives us certain guarantees about posts, designed to make us comfortable about putting them on the front page of our blog.

```
    var summary = (doc.html.replace(/<(.|\n)*?>/g, '').substring(0,350) + '...');
```

Comment on topic or styleThis line shortens the blog post's HTML (generated from Markdown before saving) and strips out most tags an images, at least well enough to keep them from showing up on the index page, for brevity.

Comment on topic or styleThe next section is the crux of the view. We're emitting for each document, a key (`doc.created_at`) and a value. The key is used for sorting, so that we can pull out all the posts in a particular date-range efficiently.

```
    emit(doc.created_at, {
      html : doc.html,
      summary : summary,
      title : doc.title,
      author : doc.author
    });
```

Comment on topic or styleThe value we've emitted is a JavaScript object, which copies some fields from the document (but not all), and the summary string we've just generated. It's preferable to avoid emitting entire documents. As a general rule, you want to keep your views as lean as possible. Only emit data you plan to use in your application. In this case we emit the summary (for the index page) the html (for the atom feed), the blog post title, and its author.

```
  }
};
```

Comment on topic or styleYou should be able to follow the definition of the above map function just fine by now. The `emit()` call creates an entry for each blog post document in our view's result set. We'll call the view `recent-posts`. Our design document looks like this now

```
{
  "_design/sofa",
  "views": {
    "recent-posts": {
      "map": "function(doc) { if (doc.type == "post") { ... }"
    }
  }
  "_attachments": {
    ...
  }
}
```

Comment on topic or styleCouchApp manages aggregating the filesystem files into our JSON design document, so we can edit our view in a file called `views/recent-posts/map.js` where we'll see this code:

Comment on topic or styleOnce the map function is stored on the design document, our view is ready to be queried for the latest ten posts.

Comment on topic or styleLet's query the view:

Comment on topic or styleAgain, this looks very similar to displaying a single post, the only real difference now is that we get an array of JSON objects back instead of just a single JSON object.

Comment on topic or styleGET request to the URI `/blog/_design/sofa/_view/recent-posts`.

Comment on topic or styleA view defined in the document `/database/_design/designdocname` in the `views` field, end up being callable under `/database/_design/designdocname/_view/viewname`.

Comment on topic or styleYou can pass in HTTP query arguments to customize your view query. In this case, we pass in

```
descending: true,
limit: 5
```

Comment on topic or styleto get the latest post first and only the first 5 post at all. The actual view request URL then is

```
/blog/_design/sofa/_view/recent-posts?descending=true&limit=5
```

Rendering The View as HTML Using a List Function

Comment on topic or styleThe `_list` API is covered in detail in The Design Documents section. In our example application, we'll use a JavaScript list function to render a view of recent blog posts as both XML and HTML formats. CouchDB's JavaScript view server also ships with the ability to respond appropriately to HTTP content negotiation and Accept headers.

Comment on topic or styleThe essence of the `_list` API is a function which is fed one row at a time and sends the response back one chunk at a time.

Sofa's List Function

Comment on topic or styleLet's take a look at Sofa's list function. This is a rather long listing, and introduces a few new concepts, so we'll take it slow and be sure to cover everything of interest.

```
function(head, req) {
  // !json templates.index
  // !json blog
  // !code vendor/couchapp/path.js
  // !code vendor/couchapp/date.js
  // !code vendor/couchapp/template.js
  // !code lib/atom.js
```

Comment on topic or styleThe top of the function declares the arguments `head` and `req`. Our function does not use `head`, just `req`, which contains information about the request such as the headers sent by the client and a representation of the query string as sent by the client. The first lines of the function are CouchApp macros which pull in code and data from elsewhere in the design document. As we've described in more detail in the Managing Design Documents chapter, these macros allow us to work with short, readable functions which pull in library code from elsewhere in the design document. Our list function uses the CouchApp JavaScript helpers for generating urls (path.js), for working with Date object (date.js) and the template function we're using to render HTML.

```
  var indexPath = listPath('index','recent-posts',{descending:true, limit:5});
  var feedPath = listPath('index','recent-posts',{descending:true, limit:5,
format:"atom"});
```

Comment on topic or styleThe next two lines of the function generate URLs used to link to the index page itself, as well as the XML Atom feed version of it. The `listPath` function is defined in path.js - the upshot is that it knows how to link to lists generated by the same design document it is run from.

Comment on topic or styleThe next section of the function is responsible for rendering the HTML output of the blog. Refer to the Show Functions chapter for details about the API we use here. In short, clients can describe the format(s) they prefer in the HTTP Accept header, or in a `format` query parameter. On the server, we declare which formats we provide, as well as assigning each format a priority. In cases where the client accepts multiple format, the first declared format is returned. It is not uncommon for browsers to accept a wide range of formats, so take care to put HTML at the top of the list, or else you can end up with browsers recieving alternate formats when they expect HTML.

```
  provides("html", function() {
```

Comment on topic or styleThe `provides` function takes two arguments, the name of the format (which is keyed to a list of default mime-types), and a function to execute when rendering that format. Note that when using `provides`, all `send` and `getRow` calls must happen within the render function. Now let's look at how the HTML is actually generated.

```
    send(template(templates.index.head, {
      title : blog.title,
      feedPath : feedPath,
      newPostPath : showPath("edit"),
      index : indexPath,
      assets : assetPath()
    }));
```

Comment on topic or styleThe first thing that we see is a template being run with an object that contains the blog title and a few relative URLs. The template function used by Sofa is fairly simple, it just replaces some parts of the template string with passed in values. In this case, the template string is stored in the variable `templates.index.head` which was imported using a CouchApp macro at the top of the function. The second argument to the template function are the values which will be inserted into the template, in this case `title`, `feedPath`, `newPostPath`, `index`, and `assets`. We'll look at the template itself later in this chapter. For now it's sufficient to know that the template stored in `templates.index.head` renders the topmost portion of the HTML page, which does not change regardless of the contents of our recent posts view.

Comment on topic or styleNow that we've rendered the top of the page, it's time to loop over the blog posts, rendering them one at a time. The first thing we do is declare our variables and our loop.

```
    var row, key;
    while (row = getRow()) {
      var post = row.value;
      key = row.key;
```

Comment on topic or styleThe `row` variable is used to store each JSON view row as it is sent to our function. The `key` variable plays a different role. Because we don't know ahead of time which of our rows will be the last row to be processed, we keep the key available in its own variable, to be used after all rows are rendered, to generate the link to the next page of results.

```
send(template(templates.index.row, {
    title : post.title,
    summary : post.summary,
    date : post.created_at,
    link : showPath('post', row.id)
  }));
}
```

Comment on topic or styleNow that have the row and its key safely stored, we use the template engine again for rendering. This time we use the template stored in `templates.index.row`, with a data item that includes the blog post title, a URL for its page, the summary of the blog post we generated in our map view, and the date the post was created.

Comment on topic or styleOnce all the blog posts included in the view result have been listed, we're ready to close the list and finish rendering the page. The last string does not need to be sent to the client using `send()`, but can be returned from the HTML function. Aside from that minor detail, rendering the tail template should be familiar by now.

```
    return template(templates.index.tail, {
      assets : assetPath(),
      older : olderPath(key)
    });
  });
```

Comment on topic or styleOnce the tail has been returned, we close the HTML generating function. If we didn't care to offer an Atom feed of our blog, we'd be done here. But we know most readers are going to be accessing the blog through a feed reader or some kind of syndication, so an Atom feed is crucial.

```
  provides("atom", function() {
```

Comment on topic or styleThe Atom generation function is defined in just the same way as the HTML generation function - by being passed to `provides()` with a label describing the format it outputs. The general pattern of the Atom function is the same as the HTML function: output the first section of the feed, then output the feed entries, and finally close the feed.

```
    // we load the first row to find the most recent change date
    var row = getRow();
```

Comment on topic or styleOne difference is that for the Atom feed, we need to know when it was last changed. This will normally be the time at which the first item in the feed was changed, so we load the first row, before outputting any data to the client (other than HTTP headers, which are set when the provides function picks the format). Now that we have the first row, we can use the date from it to set the Atom feed's last-updated field.

```
    // generate the feed header
    var feedHeader = Atom.header({
      updated : (row ? new Date(row.value.created_at) : new Date()),
      title : blog.title,
      feed_id : makeAbsolute(req, indexPath),
      feed_link : makeAbsolute(req, feedPath),
    });
```

Comment on topic or styleThe `Atom.header` function is defined in `lib/atom.js` which was imported by CouchApp at the top of our function. This library uses JavaScript's E4X extension to generate feed XML.

```
    // send the header to the client
    send(feedHeader);
```

Comment on topic or styleOnce the feed header has been generated, sending it to the client uses the familiar `send()` call. Now that we're done with the header, we'll generate each Atom entry, based on a row in the view. We use a slightly different loop format in this case than in the HTML case, as we've already loaded the first row in order to use it's timestamp in the feed header.

```
    // loop over all rows
    if (row) {
      do {
```

Comment on topic or styleThe JavaScript `do / while` loop is similar to the `while` loop used in the HTML function, except that it's guaranteed to run at least once, as it evaluates the conditional statement after each iteration. This means we can output an entry for the row we've already loaded, before calling `getRow()` to load the next entry.

```
        // generate the entry for this row
        var feedEntry = Atom.entry({
          entry_id : makeAbsolute(req, '/' +
            encodeURIComponent(req.info.db_name) +
            '/' + encodeURIComponent(row.id)),
          title : row.value.title,
          content : row.value.html,
          updated : new Date(row.value.created_at),
          author : row.value.author,
          alternate : makeAbsolute(req, showPath('post', row.id))
        });
        // send the entry to client
        send(feedEntry);
```

Comment on topic or styleRendering the entries also uses the Atom library in `atom.js`. The big difference between the Atom entries and the list items in HTML, is that for our HTML screen we only output the summary of the entry text, but for the Atom entries we output the entire entry. By changing the value of `content` from `row.value.html` to `row.value.summary` you could change the Atom feed to only include shortened post summaries, forcing subscribers to click through to the actual post to read it.

```
      } while (row = getRow());
    }
```

Comment on topic or styleAs we mentioned above, this loop construct puts the loop condition at the end of the loop, so here is where we load the next row of the loop.

```
    // close the loop after all rows are rendered
    return "";
  });
};
```

Comment on topic or styleOnce all rows have been looped over, we end the feed by returning the closing XML tag to client as the last chunk of data.

Scaling Basics

Comment on topic or styleScaling is an overloaded term. Finding a discrete definition is tricky. Everyone and her grandmother have their own idea what scaling means. Most definitions are valid, but they can be contradicting. To make things even worse, there are a lot of misconceptions about scaling. To find out what it is, one needs a scalpel to find out the important bits.

Comment on topic or styleFirst, scaling doesn't refer to a specific technique or technology, scaling, or *scalable*, is an attribute of a specific architecture. What is being *scaled* varies for nearly each project.

Comment on topic or styleScaling is specialization.

— *Joe Stump, Lead Architect of digg.com*

Comment on topic or styleJoe's quote is the one that we find is the most accurate description of scaling. It is also wishy-washy; but that is the nature of scaling. An example: A website like `facebook.com` with a whole lot of users and data associated with these users and with more and more users coming in everyday might want to scale over user-data that typically lives in a database. In contrast `flickr.com` is at it's core like Facebook with users and data for users, but in flickr's case, the data that grows fastet is images uploaded by users. These images do not necessarily live in a database so scaling image storage is flickr's path to growth. [fact check and/or find better example].

Comment on topic or styleIt is common to think of scaling as *scaling out*. This is shortsighted. Scaling can also mean *scaling in* - that is, being able to use fewer computers when demand declines. More on that later.

Comment on topic or styleThese are just two services. There are a lot more and every one has different things they want to scale. CouchDB is a database; we are not going to cover every aspect of scaling any system. We concentrate on the bits that are interesting to you, the CouchDB user. We identified three general properties that you can scale with CouchDB:

- Comment on topic or style Read requests
- Comment on topic or style Write requests
- Comment on topic or style Data

Scaling Read Requests

Comment on topic or styleA read request is retrieves a piece of information from the database. It passes the following stations within CouchDB: The HTTP server module needs to accept the request. For that, it opens a socket to send data over. The next station is the HTTP request handle module that analyzes the request and directs it to the appropriate sub-module in CouchDB. For single documents, the request then gets passed to the database module where the data for the document is looked up on the filesystem and returned all the way up again.

Comment on topic or styleAll this takes processing time and enough sockets (or file descriptors) must be available. The storage backend of the server must be able to fulfill all read requests. There are a few more things that can limit a system to accept more read requests; the basic point here is that a single server can only process so many concurrent requests. If your applications generates more requests you need to set up a second server that your application can read from.

Comment on topic or styleThe nice thing about read requests is that they can be cached. Often-used items can be held in memory and can be returned at a much higher level than the one that is your bottleneck. Requests that can use this cache, don't ever hit your database and are thus virtually toll-free. The Load Balancing chapter explains this scenario.

Scaling Write Requests

Comment on topic or styleA write requests is like a read request, only a little worse. It not only reads a piece of data from disk, it writes it back after modifying it. Remember the nice thing about reads being cacheable. Writes: Not so much. A cache must be notified when a write changes data or clients must be told to not use the cache. If you have multiple servers for scaling reads, a write must occur on all servers. In any case, you need to work harder with a write. The Clustering chaptercovers methods for scaling write requests across servers.

Scaling Data

Comment on topic or styleThe third way of scaling is scaling data. Todays hard drives are cheap and have a lot of capacity, and it only gets better in the future, but there is only so much data a single server can make sensible use of. It must maintain one more indexes to the data which uses disk space again. Creating backups will take longer and other maintenance tasks become a pain.

Comment on topic or styleThe solution is to chop the data into manageable chunks and put each chunk on a separate server. All servers with a chunk now form a *cluster* that holds all your data. The Clustering chapter takes a look at creating and using these clusters.

Comment on topic or styleWhile we are taking separate looks at scaling of reads, writes, and data, these rarely occur isolated. Decisions to scale one will affect the others. We will describe individual as well as combined solutions in the following chapters.

Basics First

Comment on topic or styleReplication is the basis for all of the three scaling methods. Before we go scaling, the Replication chapter will get you familiar with CouchDB's excellent replication feature.

Replication

Comment on topic or styleThis chapter introduces CouchDB's world class replication system. Replication synchronizes two copies of the same database, allowing users to have low latency access data no matter where they are. These databases can live on the same server or on two different servers, CouchDB doesn't make a distinction. If you change one copy of the database, replication will send these changes to the other copy.

Comment on topic or styleReplication is a one-off operation: You send an HTTP request to CouchDB that includes a *source* and a *target* database and CouchDB will send the changes from the source to the target. That is all. Granted, calling something world class and then only needing one sentence to explain it does seem odd. But part of the reason why CouchDB's replication is so powerful lies in its simplicity. Let's see what replication looks like:

```
POST /_replicate HTTP/1.1
{"source":"database","target":"http://example.org/database"}
```

Comment on topic or styleThis call sends all the documents in the local database `database` to the remote database `http://example.org/database`. A database is considered "local" when it is on the same CouchDB instance you send the `POST /_replicate` HTTP request to. All other instances of CouchDB are "remote".

Comment on topic or styleIf you want to send changes from the target to the source database, you just make the same HTTP requests, only with source and target database swapped. That is all.

```
POST /_replicate HTTP/1.1
{"source":"http://example.org/database","target":"database"}
```

Comment on topic or styleA remote database is identified by the same URL you use to talk to it. CouchDB replication works over HTTP using the same mechanisms that are available to you. This example shows that replication is a *unidirectional* process. Documents are copied from one database to another and not automatically vice versa. If you want *bidirectional* replication, you need to trigger two replications with *source* and *target* swapped.

The Magic

Comment on topic or styleWhen you ask CouchDB to replicate a database to another one, it will go and compare the two databases to find out which documents on the source differ from the target and then submit a batch of the changed documents to the target until all changes are transferred. Changes include new documents, changed documents and deleted documents. Documents that already exist on the target in the same revision are not transferred, only newer revisions.

Comment on topic or styleDatabases in CouchDB have a *sequence number* that gets incremented every time the database is changed. CouchDB remembers what changes came with which sequence number. That way, CouchDB can answer questions like "What changed in database A between sequence number 212 and now" by returning a list of new and changed documents. Finding the differences between databases with this is an efficient operation. It also adds to the robustness of replication.

Comment on topic or styleCouchDB views use the same mechanism when determining when a view needs updating and which documents to replication. You can use this to build your own solutions as well.

Comment on topic or styleYou can use replication on a single CouchDB instance to create snapshots of your databases to be able to test code changes without risking data loss or to be able to refer back to older states of your database. But replication gets really fun if you use two or more different computers, potentially geographically spread out.

Comment on topic or styleWith different servers, potentially hundreds or thousands of miles apart. problems are bound to happen. Servers crash, network connections break off, things go wrong. When a replication process is interrupted it leaves two replicating CouchDB's in an intermediate state. Then when the problems are gone, and you trigger replication again it continues where it left off.

Simple Replication with the Admin Interface

Comment on topic or styleYou can run replication from your Web browser using Futon, CouchDB's built-in administration interface. Start CouchDB and open your browser at `http://127.0.0.1:5984/_utils/`. On the right hand side you will see a list of things to visit in Futon, click on "replication".

Comment on topic or styleFuton will show you an interface to start replication. You can specify a source and a target by either picking a database from the list of local databases or you can fill in the URL of a remote database.

Comment on topic or styleClick on the *Replicate* button, wait a bit and have a look at the lower half of the screen where CouchDB gives you some statistics about the replication run or, if an error occurred, an explanatory message.

Comment on topic or styleCongratulations, you ran your first replication.

Replication in Detail

Comment on topic or styleSo far we've skipped over the result from a replication request. Now is a good time to look at it in detail. Here is an example, prettified.

```
{
  "ok": true,
  "source_last_seq": 10,
  "session_id": "c7a2bbbf9e4af774de3049eb86eaa447",
  "history": [
    {
      "session_id": "c7a2bbbf9e4af774de3049eb86eaa447",
      "start_time": "Mon, 24 Aug 2009 09:36:46 GMT",
      "end_time": "Mon, 24 Aug 2009 09:36:47 GMT",
      "start_last_seq": 0,
      "end_last_seq": 1,
      "recorded_seq": 1,
      "missing_checked": 0,
      "missing_found": 1,
      "docs_read": 1,
      "docs_written": 1,
      "doc_write_failures": 0,
    }
  ]
}
```

Comment on topic or style`"ok": true`, similar to other responses tells us everything went well. `source_last_seq` includes the source's `update_seq` value that was considered by this replication. Each replication request is assigned a `session_id` which is just a UUID; you can also talk about a *replication session* identified by this id.

Comment on topic or styleThe next bit is the replication *history*. CouchDB maintains a list of history sessions for future reference. The history array is currently capped at 50 entries. Each unique replication trigger object (the JSON string that includes the source and target databases as well as potential options) gets its own history. Let's see what a history entry is all about:

Comment on topic or styleThe `session_id` is recorded here again for convenience. The start- and end-time for the replication session are recorded. the `_last_seq` denote the `update_seq`s that were valid at the beginning and the end of the session. `recorded_seq` is the `update_seq` of the target again. Its different from `end_last_seq` if a replication process dies in the middle and is restarted. `missing_checked` is the number of docs on the target that are already there and don't need to be replicated. `missing_found` is the number of missing documents on the source.

Comment on topic or styleThe last three `docs_read`, `docs_written` and `doc_write_failures` show how many docs we read from the source, written to the target and how many failed. If all is well `_read` and `_written` are identical and `doc_write_failures` is 0. If not, you know something went wrong during replication. Possible failures are a server crash on either side, a lost network connection or a `validate_doc_update` function rejecting a document write.

Comment on topic or styleOne common scenario is triggering replication on nodes that have admin accounts enabled. Creating design documents is restricted to admins and if the replication is triggered without admin credentials, writing the design documents during replication will fail and be recorded as `doc_write_failures`. If you have admins, be sure to include the credentials in the replication request:

```
> curl -X POST http://127.0.0.1:5984/_replicate \
  -d '{"source":"http://example.org/database", \
       "target":"http://admin:password@e127.0.0.1:5984/database"}'
```

Continuous Replication

Comment on topic or styleNow that you know how replication works under the hood, we share a neat little trick. When you add `"continuous":true` to the replication trigger object, CouchDB will not stop after replicate all missing documents from the source to the target. It will listen on CouchDB's `_changes` API (see the Change Notifications chapter) and automatically replicate over any new docs as the come into the source to the target. In fact, they are not replicated right away, there's a complex algorithm determining the ideal moment to replicate for maximum performance. The algorithm is complex and is fine-tuned every once in a while and documenting it here wouldn't make much sense.

```
> curl -X POST http://127.0.0.1:5984/_replicate \
  -d '{"source":"db", "target":"db-replica", "continuous":true}'
```

Comment on topic or styleAt the time of writing, CouchDB doesn't remember continuous replications over a server restart. For the time being, you are required to trigger them again, when you restart CouchDB. In the future, CouchDB will allow you to define permanent continuous replications that survive a server restart without you having to do anything.

That's it?

Comment on topic or styleReplication is the foundation on which the following chapters build on. Make sure you understood this chapter. If you don't feel comfortable yet, just read it again and play around with the replication interface in Futon.

Comment on topic or styleWe haven't yet told you everything about replication. The next chapters show your how to manage replication conflicts (see the Conflict Management chapter), how to use a set of synchronized CouchDB instances for load balancing (see the Load Balancing chapter) and how to build a cluster of CouchDB's that can handle more data or write requests than a single node (see the Clustering chapter).

Conflict Management

Comment on topic or styleSuppose you are sitting in a coffee shop working on your book. Chris comes over and tells you about his new phone. The new phone came with a new number and you have Chris dictate it while you change it using your laptop's address book application.

Comment on topic or styleLuckily, your address book is built on CouchDB; so when you come home, all you need to do to get your home computer up to date with Chris's number is replicate your address book from your laptop. Neat, eh? What's more, CouchDB has a mechanism to maintain continuous replication, so you can keep a whole set of computers in sync with the same data, whenever a network connection is available.

Comment on topic or styleLet's change the scenario a little bit. Chris didn't anticipate meeting you at the coffee shop and sent you a mail with the new number. You weren't using the WiFi so you could concentrate on your work. You didn't read his email until getting back home and meanwhile, it was a long day, you have long forgotten that you changed the number in the address book on your laptop. You read the email, however, when getting back home and you simply copy & paste the number into your address book on your home computer. Now, and here is the twist, you copied the number wrong on your laptop's address book.

Comment on topic or styleYou now have a document in each of the databases that has different information. This situation is called a *conflict*. Conflicts occur in distributed systems. They are a natural state of your data. How does CouchDB's replication system deal with conflicts? [fix story-to-textbook-lingo-switch]

Comment on topic or styleWhen you replicate two databases in CouchDB and you have conflicting changes, CouchDB will detect that and flag the affected document with the special attribute `"\_conflicts":true`. Next, CouchDB determines which of the changes will be stored as the latest revision (remember, documents in CouchDB are versioned). The version that gets picked to be the latest revision is the *winning revision*. The *losing revision* gets stored as the previous revision.

Comment on topic or styleCouchDB does not attempt to merge the conflicting revision. Your application dictates how the merging should be done. The choice of picking the winning revision is arbitrary. In the case of the phone number, there is no way for a computer to decide on the *right* revision. This is not specific to CouchDB, no other software can this (ever had your contacts sync tool for your phone ask you which contact from which source to take?).

Comment on topic or styleReplication guarantees that conflicts are detected and that each instance of CouchDB makes the same choice regarding winners and losers, independent of all the other instances. There is no group-decision made, instead, a deterministic algorithm determines the order of the conflicting revision. After replication, all instances taking part have the same data. The data set is said to be in a *consistent state*. If you ask any instance for a document, you will get the same answer regardless of which one you ask.

Comment on topic or styleWhether or not, CouchDB picked the version that your application needs, you need to go and resolve the conflict, just as you need to resolve a conflict in a version control system like Subversion. Simply create a version that you want to be the latest by either picking the lastest, or the previous, or both (by merging them) and save it as the now latest revision. Done.

Replicate again and your resolution will populate over to all other instances of CouchDB. Your conflict resolving on one node could lead to further conflicts all of which will need to be addressed, but eventually, you will end up with a conflict free database on all nodes.

The Split Brain

Comment on topic or styleThis is an interesting conflicts scenario in that we helped a British broadcasting company build a solution for that is now in production. The basic set up is this: to guarantee that the company's website are online 24/7 even in the event of the loss of a data center they have multiple data centers backing the website. The "loss" of a data center is a rare occasion, but it can be as simple as network outages where the data center is still alive and well, but can't be reached by anyone.

Comment on topic or styleThe split brain scenario is one where the two (for simplicity's sake we'll stick to two) data centers are up and well connected to end-users, but the connection between the data centers, which is most likely not the same connection that end-users use to talk to the computers in the data center, fails.

Comment on topic or styleThe inter-data-center connection is used to keep both centers *in sync* so either can take over for the other in case of a failure. When that link goes down we end up with two halves of a system that act independently; the split brain.

Comment on topic or styleAs long as all end users can get to their data, the split brain is not scary. Resolving the split brain situation and bringing up the connection that links the data centers and staring synchronization again is where it gets hairy. Arbitrary conflict resolution, like CouchDB does by default can lead to unwanted effects on the user's side. Data could revert to an earlier stage and leave the impression that changes weren't reliably saved while in fact they were.

Conflict Resolution by Example

Comment on topic or styleLet's go through an illustrated example of how conflicts emerge an how to solve them in super slow-motion. Figure 4-1 illustrates the basic setup: we have two CouchDB databases and we are replicating from database A to database B. To keep this simple we assume triggered replication and not continuous replication and we don't replicate back from database B to A. All other replication scenarios can be reduced to this setup, so this explains everything we need to know.

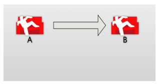


Figure 4-1: Conflict Management by Example
Comment on topic or styleWe start out by creating a document in database A. Note the clever use of imagery to identify a specific revision of a document. Since we are not using continuous replication database B won't know about the new document for now.

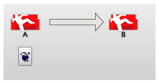


Figure 4-2: Conflict Management by Example
Comment on topic or styleWe now trigger replication and tell it to use database A as the source and database B as the target. Our document gets copied over to database B. To be precise, the latest revision of our document gets copied over.

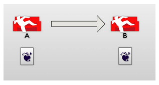


Figure 4-3: Conflict Management by Example

Comment on topic or styleNow we go to database B and update the document. We change some values and upon change, CouchDB generates a new revision for us. Note that this revision got a new image. Node A is ignorant of any activity.

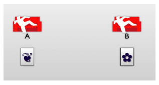


Figure 4-4: Conflict Management by Example

Comment on topic or styleNow we make a change to our document in database A and we change some other values. See how it makes a different image for us to see the difference. It is important to not that this is still the same document. It's just that there are two different revisions of that same document in each database.

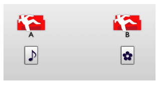


Figure 4-5: Conflict Management by Example

Comment on topic or styleNow we trigger replication again from database A to database B as before. By the way, it doesn't make a difference if the two databases live in the same CouchDB server or on different servers connected over a network.

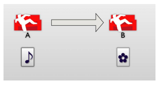


Figure 4-6: Conflict Management by Example

Comment on topic or styleWhen replicating CouchDB detects that there are two different revisions for the the same document and it creates a conflict. A document conflict means that there are now two latest revisions for this document.

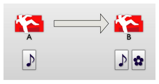


Figure 4-7: Conflict Management by Example

Comment on topic or styleFinally, we tell CouchDB which version we like to be latest revision by resolving the conflict. Now both databases have the same data.

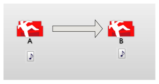


Figure 4-8: Conflict Management by Example

Comment on topic or styleOther possible outcomes include choosing the other revision and replicating that decision back to database A or creating yet another revision in database B that includes parts of both conflicting revisions (a *merge*) and replicate that back to database A.

Working with Conflicts

Comment on topic or styleNow that we walked through replication with pretty pictures, let's get our hands dirty and see what the API calls and responses for this and other scenarios look like. We'll be continuing The Core API chapter by using *curl* on the command line to make raw API requests.

Comment on topic or styleFirst, we create two databases that we can use for replication. These live on the same CouchDB instance, but they might as well live on a remote instance, CouchDB doesn't care.

Comment on topic or styleTo save us some typing we create a shell variable for our CouchDB base URL that we want to talk to. We then proceed and create two databases `db` and `db-replica`:

```
HOST="http://127.0.0.1:5984"

> curl -X PUT $HOST/db
{"ok":true}

> curl -X PUT $HOST/db-replica
{"ok":true}
```

Comment on topic or styleIn the next step we create a simple document `{"count":1}` in `db`. And trigger replication to `db-replica`:

```
curl -X PUT $HOST/db/foo -d '{"count":1}'
{"ok":true,"id":"foo","rev":"1-74620ecf527d29daaab9c2b465fbce66"}

curl -X POST $HOST/_replicate -d
'{"source":"db","target":"http://127.0.0.1:5984/db-replica"}'
{"ok":true,...,"docs_written":1,"doc_write_failures":0}]}
```

Comment on topic or styleWe skip a bit of the output of the replication session, see the Replication chapter for details. If you see `"docs_written":1` and `"doc_write_failures":0` our document made it over to `db-replica`.

Comment on topic or styleWe now update the document to `{"count":2}` in `db-replica`. Note that we now need to include the correct `_rev` property.

```
> curl -X PUT $HOST/db-replica/foo -d '{"count":2,"_rev":"1-
74620ecf527d29daaab9c2b465fbce66"}'
{"ok":true,"id":"foo","rev":"2-de0ea16f8621cbac506d23a0fbbde08a"}
```

Comment on topic or styleNext, we create the conflict! We change our document on `db` to `{"count":3}`. Our document is now logically in conflict, but CouchDB doesn't know about it until we replicate again:

```
> curl -X PUT $HOST/db/foo -d '{"count":3,"_rev":"1-
74620ecf527d29daaab9c2b465fbce66"}'
{"ok":true,"id":"foo","rev":"2-7c971bb974251ae8541b8fe045964219"}

> curl -X POST $HOST/_replicate -d
'{"source":"db","target":"http://127.0.0.1:5984/db-replica"}'
{"ok":true,..."docs_written":1,"doc_write_failures":0}]}
```

Comment on topic or styleTo see that we have a conflict, we create a simple view in `db-replica`. The map function looks like this:

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

Comment on topic or styleWhen we query this view, we get this result:

```
{"total_rows":1,"offset":0,"rows":[
{"id":"foo","key":["2-7c971bb974251ae8541b8fe045964219"],"value":null}
]}
```

Comment on topic or styleThe `key` here corresponds to the `doc._conflicts` property of our document in `db-replica`. It is an array listing all *conflicting revisions*. We see that the revision we wrote on `db` (`{"count":3}`) is in conflict. CouchDB’s automatic promotion of one revision to be the winning revision chose our first change (`{"count":2}`). To verify that, we just request that document from `db-replica`:

```
> curl -X GET $HOST/db-replica/foo
{"_id":"foo","_rev":"2-de0ea16f8621cbac506d23a0fbbde08a","count":2}
```

How does CouchDB Decide Which Revision to Use?

Comment on topic or styleCouchDB guarantees that each instance that sees the same conflict comes up with the same winning and losing revisions. It does so by running a deterministic algorithm to pick the winner. Application should not rely on the details of this algorithm and must always resolve conflicts. We’ll tell you how it works anyway.

Comment on topic or styleEach revision includes a list of previous revisions. The revision with the longest revision history list becomes the winning revision. If they are the same, the `_rev` values are compared in ASCII sort order, the highest wins.

Comment on topic or styleIn our example `2-de0ea16f8621cbac506d23a0fbbde08a` is higher than `2-7c971bb974251ae8541b8fe045964219`.

Comment on topic or styleOne advantage of this algorithm is that CouchDB nodes do not have to talk to each other to agree on winning revisions. We already learned that the network is prone to errors and avoiding it for conflict resolution makes CouchDB very robust.

Comment on topic or styleTo resolve the conflict we need to determine which one we want to keep. Let’s say we want to keep the highest value. This means we don’t agree with CouchDB’s automatic choice. To do this, we first overwrite the target document with our value and then simply delete the revision we don’t like:

```
curl -X DELETE $HOST/db-replica/foo?rev=2-de0ea16f8621cbac506d23a0fbbde08a
{"ok":true,"id":"foo","rev":"3-bfe83a296b0445c4d526ef35ef62ac14"}

curl -X PUT $HOST/db-replica/foo -d "{\"count\":3,\"_rev\":\"2-
7c971bb974251ae8541b8fe045964219\"}"
{"ok":true,"id":"foo","rev":"3-5d0319b075a21b095719bc561def7122"}
```

Comment on topic or styleCouchDB creates yet another revision that reflects our decision. Note that the `3-` didn’t get incremented this time. We didn’t create a new version of the document body, we just deleted a conflicting revision. To see all is well, we check if our revision ended up in the document.

```
curl GET $HOST/db-replica/foo
{"_id":"foo","_rev":"3-5d0319b075a21b095719bc561def7122","count":3}
```

Comment on topic or styleWe also verify that our document is no longer in conflict by querying our conflicts view again and we see that there are no more conflicts:

```
{"total_rows":0,"offset":0,"rows":[]}
```

Comment on topic or styleFinally, we replicate from `db-replica` back to `db` by simply swapping `source` and `target` in our request to `_replicate`:

```
curl -X POST $HOST/_replicate -d
'{"target":"db","source":"http://127.0.0.1:5984/db-replica"}'
```

Comment on topic or styleAnd see that that our revision ends up in `db`, too:

```
curl GET $HOST/db/foo
{"_id":"foo","_rev":"3-5d0319b075a21b095719bc561def7122","count":3}
```

Comment on topic or styleAnd we're done.

Deterministic Revision Ids

Comment on topic or styleLet's have a look at this revision id: `3-5d0319b075a21b095719bc561def7122`. Parts of the format might look familiar. The first part is an integer followed by a dash (`3-`). The integer increments for each new revision the document receives. Updates to the same document on multiple instances create their own, independent increments. When replicating, CouchDB knows that there are two different revisions `2-` (like in our example above) by looking at the second part.

Comment on topic or styleThe second part is an md5-hash over a set of document properties: The JSON body, the attachments and the `_deleted` flag. This allows CouchDB to save on replication time in case you make the same change to the same document on two instances. Earlier versions (0.9 and back) used random integers to specify revisions and making the same change on two instances would result in two different revision ids, creating a conflict, where it was not really necessary. CouchDB 0.10 and above uses deterministic revision ids using the md5 hash.

Comment on topic or styleFor example, let's create two documents `a` and `b` with the same contents to illustrate the deterministic revision ids.

```
curl -X PUT $HOST/db/a -d '{"a":1}'
{"ok":true,"id":"a","rev":"1-23202479633c2b380f79507a776743d5"}

> curl -X PUT $HOST/db/b -d '{"a":1}'
{"ok":true,"id":"b","rev":"1-23202479633c2b380f79507a776743d5"}
```

Wrapping Up

Comment on topic or styleThis concludes our tour of the conflict management system. You should now be able to create distributed setups that deal with conflicts in a proper way.

Load Balancing

Comment on topic or styleJill wakes up at 4:30 am looking dazzled at her mobile phone. She receives text message after text message one every minute. Finally, Joe calls. Joe is furious and Jill has trouble understanding what Joe is saying. In fact, Jill has a hard time remembering why Joe would call her in the middle of the night. Then she remembers: “Joe is running this online shop selling sports gear on one of your servers and he is furious because the server went down and Joe’s customers in New Zeeland [Sanity check time zones] are angry because they can’t get to the online shop.”

Comment on topic or styleThis is a typical scenario and you probably have seen a lot of variations of it being either in the role of Jill, or Joe, or both. If you are Jill you want to sleep at night and if you are Joe you want your customers to buy from you whenever it pleases them.

Having a Backup

Comment on topic or styleThe problems persists, computers fail and there are a lot of ways they can fail. There are hardware problems, power outages, bugs in the operating system or application software. Only CouchDB doesn’t have any bugs. Wait, that is of course not true, there can even be problems in CouchDB, no piece of software is free of bugs (except maybe Donald Knuth’s TeX system).

Comment on topic or styleWhatever the cause is, you want to make sure that the service you are providing (in this case the database for an online store) is resilient against failure. The road to resilience is a road of finding and removing single points of failure: A server’s power supply can fail. To avoid the server turning off on such an event, most come with at least two power supplies. To take this further, you could get a server where everything exists twice or more often, but that would be a highly specialized (and expensive) piece of hardware. It is much cheaper to get two similar servers where the one can take over when the other has a problem. You need to make sure both servers have the same set of data in order to switch them without a user noticing.

Comment on topic or styleRemoving all single points of failure will give you a highly available or fault tolerant system. The order of tolerance is only restrained by your budget. If you can’t afford to lose a customer’s shopping cart in any event, you need to store it on at least two servers in at least to far apart geographical locations.

Comment on topic or styleAmazon does that for example for their `amazon.com` Web site. If one datacenter is victim of an earthquake, a user will still be able to shop.

Comment on topic or styleIt is likely though that Amazon’s problems are not your problems and that you have a whole set of new problems when your data center goes away. But you still want to be able to live through a server failure.

Comment on topic or styleBefore we dive into setting up a highly available CouchDB system, we look at another situation:

Comment on topic or styleJoe calls Jill during regular business hours and relays his customer’s complaints that loading the online shop takes “forever”. Jill takes a quick look at the server and concludes that this is a lucky problem to have, leaving Joe puzzled. Jill explains that Joe’s shop is suddenly attracting a lot more users that buy things. Joe chimes in “I got this great review on that blog”, that’s where they must come from and a quick referrer check reveals that indeed a lot of the new customers are coming from a single site. The blog post already includes comments of unhappy customers voicing their frustration with the slow site. Joe wants to make his customers happy and asks Jill what to do. Jill advises to set up a second server that can take half of the load of the current server, making sure all requests get answered in a reasonable amount of time. Joe agrees and Jill sets out to set things up.

Comment on topic or styleThe solution to the outlined problem looks a lot like the one for providing a fault tolerant setup: Install a second server, synchronize all data. The difference is that with fault tolerance, the second server just sits there and waits for the first one to fail. The the second case, a second server helps to answer all incoming requests. The second case is not fault tolerant. If one server crashes, the other would get all the requests and is likely to break down or at least provide very slow service, both of which is not acceptable. Keep in mind that while the solutions look similar, high availability and fault tolerance are not the same. We get back to the second scenario in a bit, but first we will have a look at how to set up a fault tolerant CouchDB system.

Comment on topic or styleWe already gave it away in the previous chapters: The solution to synchronizing servers is replication.

Clustering

Comment on topic or styleOk, you've made it this far, I'm assuming you more or less understand what CouchDB is and how the application API works. Maybe you've deployed an application or two, and now you've dealing with enough traffic that you need to think about scaling. "Scaling" is an imprecise word, in this chapter we'll be dealing with the aspect of putting together a partitioned or sharded cluster, that will have to grow at an increasing rate over time from day one.

Comment on topic or styleIn this chapter we'll look at request and response dispatch in a CouchDB cluster with stable nodes. Then we'll cover how to add redundant hot-failover twin nodes, so you don't have to worry about losing machines. In a large cluster you should plan for 5-10% of your machines experiencing some sort of failure or reduced performance, so cluster design must prevent node failures from impacting reliability. Finally we'll look at adjusting cluster layout dynamically by splitting or merging nodes using replication.

Introducing CouchDB-Lounge

Comment on topic or styleCouchDB Lounge is a proxy-based partitioning and clustering application, originally developed for Meebo, a Web-based instant messaging service. Lounge comes with two major components, one that handles simple GET and PUT requests for documents, and the other which distribtues view requests.

Comment on topic or styleThe *dumbproxy* handles simple requests for anything which isn't a CouchDB view. This comes as a module for nginx, a high-performance reverse HTTP proxy. Due to the way reverse HTTP proxies work, this automatically allows configurable security, encryption, load distribution, compression, and of course, aggressive caching of your database resources.

Comment on topic or styleThe *smartproxy* only handles CouchDB view requests, and dispatches them to all the other nodes in the cluster, so as to distribute the work, making view performance a function of the cluster's cumulative processing power. This comes as a daemon for Twisted, a popular and high-performance event-driven network programing framework for Python.

Consistent Hashing

Comment on topic or styleCouchDB's storage model uses unique IDs to save and retrieve documents. Siting at the core of Lounge is a simple method of hashing your document IDs. Lounge then uses the first few characters of this hash to determine which shard to dispatch the request to. You can configure this behaviour by writing a *shard map* for Lounge, which is just a simple text configuration file.

Comment on topic or styleBecause Lounge allocates a portion of the hash (known as a keyspace) to each node, you can add as many nodes as you like. Because the hash function produces hexidecimal strings that bare no apparent relation to your docids, and because we dispatch requests based on the first few characters, we ensure that all nodes see roughly equal load. And bcause the hash function is consitent, Lounge will take any arbitrary docid from an HTTP request URI and point it to the same node each time.

Comment on topic or styleThis idea of splitting a colection of shards based on a keyspace is commonly illustrated as a ring, with the hash wrapped around the outside. Each tic mark designates the boundaries in keyspace between two partitions. The hash function maps from document IDs to positions on the ring. The ring is continuous so you can always add more nodes by splitting a single partition into pieces. With 4 physical servers you allocate the keyspace into 16 independent partitions by distributing them across the servers like so:

```
A: 0,1,2,3
B: 4,5,6,7
C: 8,9,a,b
D: c,d,e,f
```

Comment on topic or styleSo if the hash of your docid starts with "0", it would be dispatched to shard A. Similarly for "1", "2", or "3". Whereas, if the hash started with "c", "d", "e", or "f", it would be dispatched to shard D. As a full example, the hash "71db329b58378c8fa8876f0ec04c72e5" is mapped to the node B, database 7 in the table above. This could map to `http://B.couches.local/db-7/` on your backend cluster. In this way, the hash table is just a mapping from hashes to backend database URIs. But don't worry if this all sounds very complex, all you have to do is provide a mapping of shards to nodes and Lounge will build the hash ring appropriately - so no need to get your hands dirty if you don't want to.

Comment on topic or styleTo frame the same concept with Web architecture, because CouchDB uses HTTP, the proxy can partition documents according to the request URL, without inspecting the body. This is a core principal behind REST, and is one of the many benifits using HTTP affords us. In practice this is accomplished by running the hash function against the request URI, and comparing the result to find the the portion of the keyspace allocated. Lounge then looks up the associated shard for the hash in a configuration table, forwarding the HTTP request to the back-end CouchDB server.

Comment on topic or styleConsistent hashing is a simple way to ensure that you can always find the documents you saved, while balancing storage load evenly across partitions. Because the hash function is simple (it is based on CRC32) you are free to implement your own HTTP intermediaries or clients that can similarly resolve requests to the correct physical location of your data.

Redundant Storage

Comment on topic or styleConsistent hashing solves the problem of how to break a single logical database up evenly across a set of partitions, which can the be distributed across multiple servers. It does not address the problem of how to ensure that data you've stored is safe from loss due to hardware or software failure. If you are serious about your data, you can't consider it saved until you have at least two copies of it, preferably in different geographical locations.

Comment on topic or styleCouchDB replication makes maintaining hot-failover redundant slaves or load-balanced multi-master databases relatively painless. The specifics of how to manage replication are covered in the Replication chapter. What is important in this context is to understand that maintaining redundant copies is orthogonal to the harder task of ensuring that the cluster consistently chooses the same partition for a particular document id.

Comment on topic or styleFor data safety you'll want to have at least two or three copies of everything. However, if you encapsulate redundancy the higher layers of the cluster can treat each partition as a single unit, and let the logical partitions themselves manage redundancy and failover.

Redundant Proxies

Comment on topic or styleJust as we can't accept the possibility of hardware failure leading to data loss, we'll need to run multiple instances of the proxy nodes to avoid the chance that a proxy node crash could leave portions of the cluster unavailable. By running redundant proxy instances, and load balancing across them, we can both increase cluster throughput as well as reliability

View Merging

Comment on topic or styleConsistent hashing leaves documents on the proper node, but documents can still `emit()` any key. The point of incremental map reduce is to bring the function to the data, so we shoudn't redistribute the emitted keys, instead we send the queries to the CouchDB nodes via HTTP proxy, and merge the results using the Twisted Python Smartproxy.

Comment on topic or styleSmartproxy sends each view request to every node, so it needs to merge the responses before returning them to the client. Thankfully this operation is not resource intensive, as merging can be done in constant memory space, no matter how many rows are returned. The Smartproxy recieves the first row from each cluster node, and compares them. We sort the nodes according to their row key, using CouchDB's collation rules. Smartproxy pops the top row from first sorted node, and returns it to the client.

Comment on topic or styleThis process can be repeated as long as the clients continue to send rows, but if a limit is imposed by the client, Smartproxy must end the response early, discarding any extra rows sent by the nodes.

Comment on topic or styleThis layout is simple and loosely coupled. It has the advantage that it is easy to reason about, which helps in understanding topology and diagnosing failures. There is work underway to move the behavior to Erlang, which ought to make managing dynamic clusters possible, as well as let us integrate cluster control into the CouchDB runtime.

Growing the Cluster

Comment on topic or styleUsing CouchDB at web scale likely requires CouchDB clusters that can be scaled dynamically. Growing sites must continuously add more storage capacity, so we need a strategy to increase the size of our cluster without taking it down. Some workloads can result in temporary growth in data size, in which case we'll also need a process for shrinking the cluster without an interruption in service.

Comment on topic or styleIn this section we'll see how we can use CouchDB's replication filters to split one database into several partitions, and how to use that techinque to grow the cluster without downtime. There are simple steps you can take to avoid partitioning databases while growing the cluster.

Comment on topic or style*Oversharding* is a technique where you partition the cluster so that there are multiple shards on each physical machine. Moving a partition from one machine to another is simpler than spitting it into smaller partitions, as the configuration map of the cluster used by the proxy only needs to change to point to shards at thier new homes, rather than adding new logical shards. It's also less resource intensive to move a partition than to split it into many.

Comment on topic or styleOne question we need to answer is "how much should we overshard?" The answer depends on your application and deployment, but there are some forces that push us in one direction over another. If we get the number of shards right, we'll end up with a cluster that can grow optimally.

Comment on topic or styleIn the earlier section on view merging we discussed how merges can be accomplished in constant space, no matter the number of rows returned. The memory space and network resources required to merge views, as well as to map from document ids to partitioned, does however grow linearly with the number of partitions under a given proxy. For this reason we'll want to limit the number of partitions for each proxy. However, we can't accept an upper limit on cluster size. The solution is to use a tree of proxies, where the root proxy partitions to some number of intermediate proxies, which then proxy to database nodes.

Comment on topic or styleThe factors that come into play when deciding how many partitions each proxy should manage are: the storage available to each individual server node, the projected growth rate of the data, the network and memory resources available to proxies, and the acceptable latency for requests against the cluster.

Comment on topic or styleAssuming a conservative 64 shards per proxy, and 1TB of data storage per node (including room for compaction these nodes will need roughly 2TB of drive space), we can see that with a single proxy in front of CouchDB data nodes, we'll be able to store at maximum 64TB of data (on 128 or perhaps 192 server nodes, depending on the level of reducancy required by the system) before we have to increase the number of partitions.

Comment on topic or styleBy replacing database nodes with another proxy, and repartitioning each of the 64 partitions into another 64 partitions, we end up with 4096 partitions and a tree depth of two. Just as the initial system can hold 64 partitions on just a few nodes, we can transition to the two layer tree without needing thousands of machines. If we assume each proxy must be run on it's own node, and that at first database nodes can hold 16 partitions, we'll see that we need 65 proxies, and 256 database machines (not including redundancy factors, which should typically multiply the cluster size by two or three times.) To get started with a cluster that can grow smoothly from 64 terabytes to 4 petabytes, we can begin with roughly 600 to 1000 server nodes, adding new ones as data size grows and we move partitions to other machines.

Comment on topic or styleWe've seen that even a cluster with depth of 2 can hold a vast amount of data. Basic arithmetic shows us the by applying the same process to create a cluster with 3 layers of proxies, we can manage 262 petabytes on thousands of machines. Conservive estimates for the latency introduced by each layer is about 100 ms, so even without performance tuning we should see overall response times of 300ms even with a tree of depth 3, and should be able to manage queries over exabyte datasets in under a second.

Comment on topic or styleBy using oversharding and iteratively replacing full shards (database nodes which host only one partition) with proxy nodes that point to another set of oversharded partitions, we can grow the cluster to very large sizes while incurring a minimum of latency.

Comment on topic or styleNow we need to look at the mechanics of the two processes that allow the cluster to grow: moving a partition from an overcrowded node to an empty node, and splitting a large partition into many sub-partitions. Moving partitions is simpler, which is why it makes sense to use it when possible, running the more resource intensive repartition process only when partitions get large enough that only one or two can fit on each database server.

Moving Partitions

Comment on topic or styleAs we mentioned earlier, each partition is made up of N redundant CouchDB databases, each stored on different physical servers. To keep things easy to reason about, any operations should be applied to all redundant copies automatically. For the sake of discussion we'll just talk about the abstract partition, but be aware that the redundant nodes will all be the same size, and so should require the same operations during cluster growth.

Comment on topic or styleThe simplest way to move a partition from one node to another, is to create an empty database on the target node, and use CouchDB replication to fill the new node with data from the old node. When the new copy of the partition is up to date with the original, the proxy node can be reconfigured to point to the new machine. Once the proxy points to the new partition location, one final round of replication will bring it up to date, and the old partition can be retired, freeing space on the original machine.

Comment on topic or styleAnother method for moving partition databases is to rsync the files on disk from the old node to the new one. Depending on how recently the partition was compacted, this should result in efficient, low-CPU intialization of a new node. Replication can then be used to bring the rsynced file up to date. See more about rsync and replication in the Replication chapter.

Splitting Partitions

Comment on topic or styleThe last major thing we need to run a CouchDB cluster is the capability to split an oversized partition into smaller pieces. In the Replication chapter we discussed how to do filtered replication. Splitting partitions is accomplished by creating the target partitions, and configuring them with the range of hash keys they are interested in. They then apply filtered replication to the source partition database, requesting only documents that meet their hash criteria. The result is multiple partial copies of the source database, so that each new partition has an equal share of the data. In total, they have a complete copy of the original data. Once the replication is complete, and the new partitions have also brought their redundant backups up to date, a proxy for the new set of partitions is brought online, and the top-level proxy pointed at it instead of the old partition. Just like with moving a partition, we should do one final round of replication after the old partition is no longer reachable by the cluster, so that any last second updates are not lost. Once that is done we can retire the old partition so that its hardware can be reused elsewhere in the cluster.

Change Notifications

Comment on topic or styleSay you are building a message service with CouchDB. Each user has an *inbox* database and other users send messages to that users by dropping a message into the inbox database. When the user wants to read all messages he received, he can just open his inbox databases and see all messages.

Comment on topic or styleSo far so simple, but now you've got your users hitting the *refresh* button all the time once they are looking at all messages to see if there are new messages. This is commonly referred to as *polling*. Lots of users are generating a lot of requests that, for most of the time, don't show anything new, just the list of all the messages they already know about.

Comment on topic or styleWouldn't it be nice to ask CouchDB to give you notice when a new message arrived? The `_changes` database API does just that.

Comment on topic or styleThe above scenario can be seen as the *cache invalidation problem*, i.e. when do I know when what I am displaying right now is no longer an apt representation of the underlying data store. Any sort of cache invalidation, not only backend-frontend related can be built using `_changes`.

Comment on topic or style`_changes` is also designed and suited to extract an activity stream from a database. Whether for simple display or, equally important, to act on a new document (or a document change) when it occurs.

Comment on topic or styleThe beauty of systems that use the `_changes` API is that they are *decoupled*. A program that is only interested in latest updates doesn't need to know about programs that create new documents and vice versa.

Comment on topic or styleHere's what a changes item looks like:

```
{"seq":12,"id":"foo","changes":[{"rev":"1-23202479633c2b380f79507a776743d5"}]}
```

Comment on topic or styleThere are three fields, `seq` is the `update_seq` of the database that was created when the document with the `id` got created or changed. Finally, `changes` is an array of fields, by default it includes the document's revision id, but there can be more information about document conflicts among other things.

Comment on topic or styleThe `_changes` API is available for each database. You can get changes that happen in a single database per request. But you can easily send multiple requests to multiple database's `_changes` API if you need that.

Comment on topic or styleThere are three ways to request notifications: *polling* (the default), *long polling* and *continuous*. Each is useful in a different scenario, we'll discuss all of them in detail here.

Comment on topic or styleLet's create a database that we can use in example later in this chapter:

```
> HOST="http://127.0.0.1:5984"
> curl -X PUT $HOST/db
{"ok":true}
```

Polling for Changes

Comment on topic or styleIn the example above we tried to avoid the polling method, but it is very simple and in some cases the only one suitable for a problem. Because it is the simplest case, it is the default for the changes API.

Comment on topic or styleLet's see what the changes for our test database look like. First, the request (we're using `curl` again):

```
curl -X GET $HOST/db/_changes
```

Comment on topic or styleThe result is simple:

```
{"results":[

],
"last_seq":0}
```

Comment on topic or styleThere's nothing there because we didn't put anything in yet, no surprise. But you can guess where we'd see results, when they start to come in. Let's create a document:

```
curl -X PUT $HOST/db/test -d '{"name":"Anna"}'
```

Comment on topic or styleCouchDB replies:

```
{"ok":true,"id":"test","rev":"1-aaa8e2a031bca334f50b48b6682fb486"}
```

Comment on topic or styleNow let's run the changes request again:

```
{"results":[
{"seq":1,"id":"test","changes":[{"rev":"1-aaa8e2a031bca334f50b48b6682fb486"}]}
],
"last_seq":1}
```

Comment on topic or styleYay, we get a notification about our new document. This is pretty neat! But wait, when we created the document, already got information like the revision id, why would we want to make a request to the changes API to get it again? Remember that the purpose of the changes API is to allow you to build decoupled systems. The program that creates the document is very likely not the same program that requests changes for the db since it already know what it put in there (although this is blurry, the same program could be interested in changes made by others).

Comment on topic or styleBehind your back, we created another document, let's see what the changes for the database look now:

```
{"results":[
{"seq":1,"id":"test","changes":[{"rev":"1-aaa8e2a031bca334f50b48b6682fb486"}]},
{"seq":2,"id":"test2","changes":[{"rev":"1-e18422e6a82d0f2157d74b5dcf457997"}]}
],
"last_seq":2}
```

Comment on topic or styleSee how we get a new line in the result that represents the new document? In addition, the first document we put in there got listed again. The default result for the changes API is the history of all changes that the database has seen.

Comment on topic or styleWe've already seen the change for `"seq":1` and we're no longer interested in it really. We can tell the changes API about that by using the `since=1` query parameter.

```
curl -X GET $HOST/db/_changes?since=1
```

Comment on topic or styleIt'll return all changes *after* the `seq` specified by `since`:

```
{"results":[
{"seq":2,"id":"test2","changes":[{"rev":"1-e18422e6a82d0f2157d74b5dcf457997"}]}
],
"last_seq":2}
```

Comment on topic or styleWhile we're discussing options, use `style=all_docs` to get more revision and conflict information in the `changes` array for each result row. If you want to specify the default explicitly, the value is `main_only`.

Comment on topic or styleNetworks are a tricky beast and sometimes you don't know whether there are no changes coming or your network connection went stale. If you add another query parameter `heartbeat=N`, where N is a number CouchDB will send you a newline character each N milliseconds. As long as you are receiving new line characters you know there are no new change notifications, but CouchDB is still there ready to send you the next one when it occurs.

Long Polling

Comment on topic or styleThe technique of long polling was invented for web browsers to remove one of the problems with the regular polling approach: do not run any requests if nothing changed. Long polling works like this: When making a request to the long polling API, you open a HTTP connection to CouchDB until a new row appears in the changes result, both you and CouchDB keep the HTTP connection open. As soon as a result appears, the connection is closed.

Comment on topic or styleThis works well for low frequency updates. If a lot of changes occur for a client, you find yourself opening a lot of new requests and the usefulness of this approach over regular polling declines. Another general consequence of this technique is that for each client requesting a long polling change notification, CouchDB will have to keep a HTTP connection open. CouchDB is well capable of doing so as it is designed to handle many concurrent requests. But you need to make sure your operating system allows CouchDB to use at least as many sockets as you have long polling clients (and a few spare for regular requests, of course).

Comment on topic or styleTo make a long polling request, add the `feed=longpoll` query parameter. For this listing, we added time stamps to show you when things happen.

```
00:00: > curl -X GET "$HOST/db/_changes?feed=longpoll&since=2"
00:00: {"results":[
00:10: {"seq":3,"id":"test3","changes":[{"rev":"1-
02c6b758b08360abefc383d74ed5973d"}]}
00:10: ],
00:10: "last_seq":3}
```

Comment on topic or styleAt 00:10, we create another document behind your back again and CouchDB promptly sends us the change. Note that we used `since=2` to not get any of the previous notifications. Also note that we had to use double quotes for the `curl` command since we are using an ampersand which is a special character for our shell.

Comment on topic or styleThe `style` option works for long polling request just like for regular polling requests.

Continuous Changes

Comment on topic or styleLong polling is great, but you still end up opening an HTTP request for each change notification. For web browsers, this is the only way to avoid the problems of regular polling. But web browsers are not the only client software that can be used to talk to CouchDB. If you are using Python, Ruby, Java or any other language really, you have yet another option:

Comment on topic or styleThe continuous changes API allows you to receive change notifications as they are coming in using a single HTTP connection. You make a request to the continuous changes API and both you and CouchDB will hold the connection open "forever" and CouchDB will send you new lines for notifications when the occur and — opposed to long polling — keeps the HTTP connection open, waiting to send the next notification.

Comment on topic or styleThis is great for both infrequent and frequent notifications and it has the same consequence as long polling: you're going to have a lot of long-living HTTP connections. But again, CouchDB easily supports these.

Comment on topic or styleUse the `feed=continuous` parameter to make a continuous changes API request. Here is a the result, again with time stamps. At `00:10` and `00:15` we'll create a new document each:

```
00:00: > curl -X GET "$HOST/db/_changes?feed=continuous&since=3"
00:10: {"seq":4,"id":"test4","changes":[{"rev":"1-
02c6b758b08360abefc383d74ed5973d"}]}
00:15: {"seq":5,"id":"test5","changes":[{"rev":"1-
02c6b758b08360abefc383d74ed5973d"}]}
```

Comment on topic or styleNote that the continuous changes API result doesn't include a wrapping JSON object with a 'results` member with the individual notification results as array items; it only includes a raw line per notification. Also note that the lines are no longer separated by a comma. Whereas the regular- and long polling APIs' result is a full valid JSON object when the HTTP request returns, the continuous changes API sends individual rows as valid JSON objects. The difference makes it easier for clients to parse the respective results.

Comment on topic or styleThe `style` and `heartbeat` parameters work as expected with the continuous changes API.

Filters

Comment on topic or styleThe change notification API and its three modes of operation already give you a lot of options requesting and processing changes in CouchDB. Filters for changes give you an additional level of flexibility. Let's say the messages from our first scenario have priorities and a user is only interested in notifications about messages with a `high` priority.

Comment on topic or styleEnter filters. Similar to view functions, a filter is a JavaScript function that gets stored in a design document and is later executed by CouchDB. They live in a special member `filters` and in there under a name of your choice; here is an example:

```
{
  "_id": "_design/app",
  "_rev": "1-b20db05077a51944afd11dcb3a6f18f1",
  "filters": {
    "important": "function(doc, req) { if(doc.priority == 'high') { return true; }
else { return false; }}"
  }
}
```

Comment on topic or styleTo query the changes API with this filter, use the `filter=designdocname/filtername` query parameter:

```
curl "$HOST/db/_changes?filter=app/important"
```

Comment on topic or styleThe result now only includes rows for document updates for which the filter function returns true. In our case where the `priority` property of our document has the value `high`. This is pretty neat, but CouchDB tops it up another notch:

Comment on topic or styleLet's take the initial example application where users can send messages to each other. Instead of having a database per user acting as the inbox we now use a single database as the inbox for all users. How can users register for changes that represent a new message being put in his inbox?

Comment on topic or styleWe can make the filter function using a request parameter:

```
function(doc, req)
```

```
{
  if(doc.name == req.query.name) {
    return true;
  }

  return false;
}
```

Comment on topic or styleIf you now run a request adding a `?name=Steve` parameter, the filter function will only return result rows for documents that have the `name` field set to "Steve". If you are running a request for a different user, just change the request parameter (`name=Joe`).

Comment on topic or styleNow, adding a query parameter to a filtered changes request is easy and what would Steve hinter to pass in `name=Joe` as the parameter and see Joe's inbox? Not much, but can CouchDB help with this? We wouldn't bring this up if it couldn't, would we?

Comment on topic or styleThe `req` parameter of the filter function includes a member `userCtx`, the *user context*. It includes information about the user that already has been authenticated over HTTP earlier in the phase of the request. Specifically `req.userCtx.name` includes the user name of the user that make the filtered changes request. We can be sure that the user is who he says he is because he as ben authenticated against one of the authenticating schemes in CouchDB. With this, we don't even need the dynamic filter parameter (although it can still be useful in other situations).

Comment on topic or styleIf you have configured CouchDB to use authentication for requests, a user will have to make an authenticated request and the result is available in our filter function:

```
function(doc, req)
{
  if(doc.name) {
    if(doc.name == req.userCtx.name) {
      return true;
    }
  }

  return false;
}
```

Wrapping Up

Comment on topic or styleThe changes API lets you build sophisticated notification schemes useful in many scenarios with components working isolated and asynchronous yet to the same beat. In combination with replication, this API is the foundation to build distributed, highly available and high-performance CouchDB clusters.

View Cookbook for SQL Jockeys

Comment on topic or styleThis is a collection of some common SQL queries and how to get the same result in CouchDB. The key to remember here is that CouchDB does not work like an SQL database at all and that best practices from the SQL world do not translate well or at all to CouchDB. This cookbook assumes that you are familiar with the CouchDB basics like creating and updating databases and documents.

Using Views (CREATE / ALTER TABLE)

Comment on topic or styleUsing views is a two step process. First you *define* a view, then you *query* it. This is analogous to defining a table structure (with indexes) using `CREATE TABLE` or `ALTER TABLE` and querying it using an SQL query.

Defining a View

Comment on topic or styleDefining a view is done by creating a special document in a CouchDB database. The only actual speciality is the `_id` of the document: it starts with `_design/`, for example `_design/application`. Other than that, it is just a regular CouchDB document. To make sure CouchDB understands that you are defining a view, you need to prepare the contents of that design document in a special format. Here is an example:

```
{
  "_id": "_design/application",
  "_rev": "1-C1687D17",
  "views": {
    "viewname": {
      "map": "function(doc) { ... }",
      "reduce": "function(keys, values) { ... }"
    }
  }
}
```

Comment on topic or styleWe are defining a view `viewname`. The definition of the view consists of two functions. The *map function* and the *reduce function*. Specifying a reduce function is optional. We'll look at the nature of the functions later. Note that `viewname` can be whatever you like; `users`, `by-name`, or `by date` are just some examples.

Comment on topic or styleA single design document can also include multiple view definitions, each identified by a unique name:

```
{
  "_id": "_design/application",
  "_rev": "1-C1687D17",
  "views": {
    "viewname": {
      "map": "function(doc) { ... }",
      "reduce": "function(keys, values) { ... }"
    },
    "anotherview": {
      "map": "function(doc) { ... }",
      "reduce": "function(keys, values) { ... }"
    }
  }
}
```

Querying a View

Comment on topic or styleThe name of the design document and the name of the view are significant for querying the view. To query the view `viewname` you perform a HTTP `GET` request to the following URI:

```
/database/_design/application/_view/viewname
```

Comment on topic or style`database` is the name of the database you created your design document in. Next up is the design document name and then the view name prefixed with `_view/`. To query `anotherview` replace `viewname` in that URI with `anotherview`. If you want to query a view in a different design document adjust the design document name.

Map & Reduce Functions

Comment on topic or styleMap/Reduce is a concept that solves problems by applying a two-step process; aptly named the *map* phase and the *reduce* phase. The map phase looks at all documents in CouchDB separately one after the other and creates a *map result*. The map result is an ordered list of key-value pairs. Both key and value can be specified by the user writing the map function. A map function may call the built-in `emit(key, value)` function 0 to N times per document, creating a row in the map result per invocation.

Comment on topic or styleCouchDB is smart enough to only run a map function once for every document, even on subsequent queries on a view. Only changes to documents, or new documents need to be processed anew.

Map Functions

Comment on topic or styleMap functions run in isolation for every document. They can't modify the document and they can't talk to the outside world; they can't have *side-effects*. This is required so CouchDB can guarantee correct results without having to recalculate a complete result when only one document gets changed.

Comment on topic or styleThe map result looks like this:

```
{"total_rows":3,"offset":0,"rows":[
  {"id":"fc2636bf50556346f1ce46b4bc01fe30","key":"Lena","value":5},
  {"id":"1fb2449f9b9d4e466dbfa47ebe675063","key":"Lisa","value":4},
  {"id":"8ede09f6f6aeb35d948485624b28f149","key":"Sarah","value":6}
}
```

Comment on topic or styleIt is a list of rows sorted by the value of `key`. The `id` is added automatically and refers back to the document that created this row. The `value` is the data you're looking for. For example purposes, it's the girl's age.

Comment on topic or styleThe map function that produces this result is:

```
function(doc) {
  if(doc.name && doc.age) {
    emit(doc.name, doc.age);
  }
}
```

Comment on topic or styleIt includes the `if` statement as a sanity check to see we're operating on the right fields and calls the emit function with the name and age as key and value.

Reduce Functions

Comment on topic or styleReduce functions are explained in the section about Aggregate Functions.

Lookup by Key (SELECT field FROM table WHERE value="searchterm")

Comment on topic or styleUse case: Get a *result* (that can be a record or set of records) associated with a *key* (`"searchterm"`).

Comment on topic or styleTo look something up quickly, regardless of the storage mechanism, an index is needed. An index is a data structure optimised for quick search and retrieval. CouchDB's map result is stored in such an index, which happens to be a b+-tree.

Comment on topic or styleTo look up a value by `"searchterm"` we need to put all values into the key of a view. All we need is a simple map function:

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

Comment on topic or styleThis creates a list of documents that have a `value` field sorted by the data in the `value` field. To find all the records that match `"searchterm"`, we query the view and specify the search term as a query parameter:

```
/database/_design/application/_view/viewname?key="searchterm"
```

Comment on topic or styleConsider the documents from the previous section and say we're indexing on the `age` field of the documents to find all the five year olds:

```
function(doc) {
  if(doc.age && doc.name) {
    emit(doc.age, doc.name);
  }
}
```

Comment on topic or styleQuery:

```
/ladies/_design/ladies/_view/age?key=5
```

Comment on topic or styleResult:

```
{"total_rows":3,"offset":1,"rows":[
{"id":"fc2636bf50556346f1ce46b4bc01fe30","key":5,"value":"Lena"}
]}
```

Comment on topic or styleEasy.

Comment on topic or styleNote that you have to emit a value. The view result includes the associated document id in every row. We can use it to look up more data from the document itself. We can also use the `?include_docs=true` parameter to have CouchDB fetch the documents individually for us.

Lookup by Prefix (SELECT field FROM table WHERE value LIKE "searchterm%")

Comment on topic or styleUse case: Find all documents that have a field value that starts with `searchterm`. For example, you stored a mime-type (like `text/html` or `image/jpg` for each document and now you want to find all documents that are images according to the mime type.

Comment on topic or styleThe solution is very similar to the previous example: all we need is a map function that is a little more clever than the first one. But first an example document:

```
{
  "_id": "Hugh Laurie",
  "_rev": "1-9fded7deef52ac373119d05435581edf",
  "mime-type": "image/jpg",
  "description": "some dude"
}
```

Comment on topic or styleThe clue lies in extracting the prefix we want to search for from our document and put it into our view index. We use a regular expression to match our prefix:

```
function(doc) {
  if(doc["mime-type"]) {
    // from the start (^) match everything that is not a slash ([^\/]+) until
    // we find a slash (\/). Slashes needs to be escaped with a backslash (\/)
    var prefix = doc["mime-type"].match(/^[^\/]+\//);
    if(prefix) {
      emit(prefix, null);
    }
  }
}
```

Comment on topic or styleYou can now query this view with your desired mime-type prefix and not only find all images, but also text, and video and all other formats:

```
/files/_design/finder/_view/by-mime-type?key="image/"
```

Aggregate Functions (SELECT COUNT(field) FROM table)

Comment on topic or styleUse case: Calculate a derived value from your data.

Comment on topic or styleWe haven’t explained reduce functions yet. Reduce functions are similar to aggregate functions in SQL. They compute a value over multiple documents.

Comment on topic or styleTo explain the mechanics of reduce functions, we’ll create one that doesn’t make a whole lot of sense. But this example is easy to understand. We’ll explore more useful reduces later.

Comment on topic or styleReduce functions operate on the output of the map function (also called the *map result* or *intermediate result*). The reduce function’s job, unsurprisingly, is to reduce the list that the map function produces.

Comment on topic or styleHere’s what our summing reduce function looks like:

```
function(keys, values) {
  var sum = 0;
  for(var idx in values) {
    sum = sum + values[idx];
  }
  return sum;
}
```

Comment on topic or styleAn alternate, more idiomatic JavaScript version:

```
function(keys, values) {
  var sum = 0;
  values.forEach(function(element) {
    sum = sum + element;
  });
  return sum;
}
```

Comment on topic or styleThis reduce function takes two arguments, a list of `keys` and a list of `values`. For our summing purposes we can ignore the `keys`-list and only consider the `value` list. We're looping over the list and add each item to a running total that we're returning at the end of the function.

Comment on topic or styleYou see one difference between the map and the reduce function. The map function uses `emit()` to create its result, the reduce function `return`s a value.

Comment on topic or styleFor example, from a list of integer values that specify the age calculate the sum of all years of life for the news headline "786 life-years present at event". A little contrived, but very simple and thus good for demonstration purposes. Consider the documents and the map-view we used earlier in this chapter.

Comment on topic or styleThe reduce function to calculate the total age of all girls is:

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

Comment on topic or styleNote that, instead of the two earlier versions, we used CouchDB's predefined `sum()` function. It does the same as the other two, but it is such a common piece of code, that CouchDB has it included.

Comment on topic or styleThe result for our reduce view now looks like this:

```
{"rows":[
{"key":null,"value":15}
]}
```

Comment on topic or styleThe total sum of all `age` fields in all our documents is `15`. Just what we wanted. The `key` member of the result object is `null` as we can't know anymore which documents took part in the creation of the reduced result. We'll cover more advanced reduce cases further down.

Comment on topic or styleAs a rule of thumb, the reduce function should reduce a single scalar value. That is an integer, a string, a small, fixed-size list or object, that includes an aggregated value (or values) from the `values` argument. It should never just return `values` or similar. CouchDB will give you a warning, if you're trying to use reduce "the wrong way":

```
{"error":"reduce_overflow_error","message":"Reduce output must shrink more
rapidly: Current output: ..."}
```

Get Unique Values (SELECT DISTINCT field FROM table)

Comment on topic or styleGetting unique values is not as easy as adding a keyword. But a reduce view and a special query parameter give us the same result. Let’s say you want a list of tags that your users have tagged themselves with and no duplicates.

Comment on topic or styleFirst, lets look at the source documents. We punt on `_id` and `_rev` attributes here:

```
{
  "name":"Chris",
  "tags":["mustache", "music", "couchdb"]
}

{
  "name":"Noah",
  "tags":["hypertext", "philosophy", "couchdb"]
}

{
  "name":"Jan",
  "tags":["drums", "bike", "couchdb"]
}
```

Comment on topic or styleFirst we need a list of all tags. A map function will do the trick:

```
function(dude) {
  if(dude.name && dude.tags) {
    dude.tags.forEach(function(tag) {
      emit(tag, null);
    });
  }
}
```

Comment on topic or styleThe result will look like this:

```
{"total_rows":9,"offset":0,"rows":[
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"bike","value":null},
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"couchdb","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"couchdb","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"couchdb","value":null},
{"id":"3525ab874bc4965fa3cda7c549e92d30","key":"drums","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"hypertext","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"music","value":null},
{"id":"da5ea89448a4506925823f4d985aabbd","key":"mustache","value":null},
{"id":"53f82b1f0ff49a08ac79a9dff41d7860","key":"philosophy","value":null}
]}
```

Comment on topic or styleAs promised all the tags, including duplicates. Since each document gets run through the map function in isolation, it cannot know if the same key has been emitted already. At this stage, we need to live with that. To achieve uniqueness we need a reduce:

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

Comment on topic or styleThis reduce doesn’t do anything, but it allows us to specify a special query parameter when querying the view:

```
/dudes/_design/dude-data/_view/tags?group=true
```

Comment on topic or styleCouchDB replies:

```
{"rows":[
{"key":"bike","value":true},
{"key":"couchdb","value":true},
{"key":"drums","value":true},
{"key":"hypertext","value":true},
{"key":"music","value":true},
{"key":"mustache","value":true},
{"key":"philosophy","value":true}
]}
```

Comment on topic or styleIn this case, we can ignore the value part, since it is always true, but they result includes a list of all our tags and no duplicates!

Comment on topic or styleWith a small change we can put the reduce to good use too, let's see how many of the non-unique tags are there for each tag. To calculate the tag-frequency, we just use the summing up we already learned about. In the map function, we emit a `1` instead of `null`:

```
function(dude) {
  if(dude.name && dude.tags) {
    dude.tags.forEach(function(tag) {
      emit(tag, 1);
    });
  }
}
```

Comment on topic or styleIn the reduce function, we return the sum of all values:

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

Comment on topic or styleNow, if we query the view with the `?group=true` parameter, we're getting back the count for each tag:

```
{"rows":[
{"key":"bike","value":1},
{"key":"couchdb","value":3},
{"key":"drums","value":1},
{"key":"hypertext","value":1},
{"key":"music","value":1},
{"key":"mustache","value":1},
{"key":"philosophy","value":1}
]}
```

How do I Enforce Uniqueness? (UNIQUE KEY(column))

Comment on topic or styleUse case: Your applications require that a certain value exists only once in a database.

Comment on topic or styleThis is an easy one: Within a CouchDB database, each document must have a unique `_id` field. If you require unique values in a database, just assign them to a document's `_id` field and CouchDB will enforce uniqueness for you.

Comment on topic or styleThere's one caveat though: In the distributed case, when you are running more than one CouchDB node that accepts write requests, uniqueness can only guaranteed per node or outside of CouchDB. CouchDB will allow two identical ids to be written to two different nodes. On replication, CouchDB will detect a conflict and flag the document accordingly.

Security

Comment on topic or styleWe mentioned earlier that CouchDB is still in development and that some thing might have been added since the publication of this book. This is especially tue for the security mechanisms in CouchDB. There is rudimentary support in currently released versions (0.10.0), but as we're writing these lines, additions are being discussed.

Comment on topic or styleIn this chapter we'll look at the basic security mechanisms in CouchDB: The *Admin Party*, *Basic Authentication*, *Cookie Authentication* and *OAuth*.

The Admin Party

Comment on topic or styleWhen you start out fresh, CouchDB allows any request to be made by anyone. Create a database? No problem, here you go. Delete some documents, same deal. CouchDB calls this the *Admin Party*. Everybody has privileges to do anything. Neat.

Comment on topic or styleWhile it is incredibly easy to get started with CouchDB that way, it should be obvious that putting a default installation into the wild is adventurous. Any rogue client could come along and delete a database.

Comment on topic or styleA note of relief: by default CouchDB will only listen on your loopback network interface (`127.0.0.1` or `localhost`) and thus only you will be able to make requests to CouchDB and nobody else. But when you start to open up your CouchDB to the public (e.g. by telling it to bind to your machine's public IP address), you will want to think about restricting access so the next bad guy doesn't ruin your admin party.

Comment on topic or styleIn the above discussions we already dropped some keywords on how things without the admin party work. First there's *Admin* itself which implies some sort of super user. Then there is *privileges*. Let's explore these a little more.

Comment on topic or styleCouchDB has the idea of an *admin user*, an administrator, a super user, root, that is allowed to do anything to a CouchDB installation. By default, everybody is an admin. If you don't like that, you can create specific admin users with a username and password as their credentials.

Comment on topic or styleCouchDB also defines a set of requests that only admin users are allowed to do. If you have defined one or more specific admin users CouchDB will ask for identification for certain requests:

- Comment on topic or style Creating a database (`PUT /database`)
- Comment on topic or style Deleting a database (`DELETE /database`)
- Comment on topic or style Creating a design document (`PUT /database/_design/app`)
- Comment on topic or style Updating a design document (`PUT /database/_design/app?rev=1-4E2`)
- Comment on topic or style Deleting a design document (`DELETE /database/_design/app?rev=1-6A7`)

- Comment on topic or style Triggering compaction (`POST /_compact`)
- Comment on topic or style Reading the task status list (`GET /_active_tasks`)
- Comment on topic or style Restart the server (`POST /_restart`)
- Comment on topic or style Read the active configuration (`GET /_config`)
- Comment on topic or style Update the active configuration (`PUT /_config`)

Creating New Admin Users

Comment on topic or styleLet's do another walk through the API using `curl` to see how CouchDB behaves when you add admin users.

```
> HOST="http://127.0.0.1:5984"
> curl -X PUT $HOST/database
{"ok":true}
```

Comment on topic or styleWhen starting out fresh, we can add a database. Nothing unexpected. Now let's create an admin user, we call her `anna`, her password is `secret`. Note the double quotes, they are needed to denote a string value for the configuration API (as we've learned earlier).

```
curl -X PUT $HOST/_config/admins/anna -d '"secret"'
""
```

Comment on topic or styleAs per the `_config` API's behaviour, we're getting the previous value for the config item we just wrote back. Since our admin user didn't exist, we get an empty string.

Comment on topic or styleWhen we now sneak over to the CouchDB logfile, we find these two entries:

```
[debug] [<0.43.0>] saving to file \
'/Users/jan/Work/couchdb-git/etc/couchdb/local_dev.ini', \
Config: '{{"admins","anna"},"secret"}'

[debug] [<0.43.0>] saving to file \
'/Users/jan/Work/couchdb-git/etc/couchdb/local_dev.ini', Config:\
'{{"admins","anna"}, \
"-hashed-
6a1cc3760b4d09c150d44edf302ff40606221526,a69a9e4f0047be899ebfe09a40b2f52c"}'
```

Comment on topic or styleThe first is our initial request, you see that our admin user gets written to the CouchDB configuration files. We set our CouchDB log level to `debug` too see exactly what is going on. First we see the request coming in with a plaintext password and then again with a hashed password.

Hashing Passwords

Comment on topic or styleSeeing the plain text password above is scary, isn't it? No worries, in normal operation when the log level is not set to `debug`, the plaintext password doesn't show up anywhere. It gets hashed right away. The hash is that big ugly long string that starts out with `-hashed-`. How does that work?

1. Comment on topic or style Create a new 128 bit UUID. This is our *salt*.
2. Comment on topic or style Create a sha1 hash of the concatenation of the bytes of the plain text

password and the salt (`sha1(password + salt)`).

3. Comment on topic or style Prefix the result with `-hashed-` and append `,salt`.

Comment on topic or styleTo compare a plain text password during authentication with the stored hash, the same procedure is run and the resulting hash is compared to the stored hash. The probability of two identical hashes for different passwords is too insignificant to mention (c.f. Bruce Schneier). Should the stored hash fall into the hands of an attacker, it is, by current standards, way too inconvenient (it'd take a lot of money and time) to find the plain text password from the hash.

Comment on topic or styleBut what's with the `-hashed-` prefix? — Remember how the configuration API works? When CouchDB starts up, it reads a set of `.ini` files with config settings. It loads these settings into an internal data store (not a database). The config API lets you read the current configuration as well as change it and create new entries. CouchDB is writing any changes back to the `.ini` files.

Comment on topic or styleThe `.ini` files can also edited by hand when CouchDB is not running. Instead of creating the admin user as we showed above, you could have stopped CouchDB, opened your `local.ini` and added `anna = secret` to the `[admins]` section and restarted CouchDB. Upon reading the new line from `local.ini`, CouchDB would run the hashing algorithm and write back the hash to `local.ini`, replacing the plain text password. In order to make sure CouchDB only hashes plain text passwords and not hash an existing hash a second time, it prefixes the hash with `-hashed-`, to distinguish between plain text passwords and hashed passwords. This means your plain text password can't start with the characters `-hashed-`, but thats pretty unlikely to begin with.

Basic Authentication

Comment on topic or styleNow that we have defined one, CouchDB will not allow us to create new databases unless we give the correct admin user credentials. Let's verify:

```
> curl -X PUT $HOST/somedatabase
{"error":"unauthorized","reason":"You are not a server admin."}
```

Comment on topic or styleThat looks about right. Now we try again with the correct credentials:

```
> HOST="http://anna:secret@127.0.0.1:5984"
> curl -X PUT $HOST/somedatabase
{"ok":true}
```

Comment on topic or styleIf you ever accessed a website or FTP server that was password protected, the `username:password@` URL variant should look familiar.

Comment on topic or styleIf you are security sensitive the missing `s` in `http://` will make you nervous. We're sending our password to CouchDB in plain text. This is a bad thing, right? Yes, but consider our scenario: CouchDB listens on `127.0.0.1` on our development box that we're the sole user of. Who could possibly sniff our password?

Comment on topic or styleIf you are in a production environment, you need to reconsider, though. Will your CouchDB instance communicate over a public network. Even a LAN shared with other co-location customers is public. There are multiple ways to secure communication between you or your application and CouchDB that exceeds the scope of this book. We suggest you read up on VPNs and setting up CouchDB behind an HTTP proxy (like Apache httpd's mod_proxy, nginx or varnish) that will handle SSL for you. CouchDB does not support exposing its API via SSL at the moment. It can however do replication with other CouchDB instances that are behind an SSL proxy.

Update Validations Again

Comment on topic or styleDo you remember Chapter 07: Validation Functions? We had an update validation function that allowed us to verify that the claimed author of document does match the authenticated user name.

```
function(newDoc, oldDoc, userCtx) {
  if (newDoc.author) {
    if(newDoc.author != userCtx.name) {
      throw("forbidden": "You may only update documents with author " +
        userCtx.name});
    }
  }
}
```

Comment on topic or styleWhat is this `userCtX` exactly? It is an object filled with information about the current request's authentication data. Let's have a look what's in there. We'll show you a simple trick how to introspect what's going on in all the JavaScript that you are writing.

```
> curl -X PUT $HOST/somedatabase/_design/log \
  -d '{"validate_doc_update":"function(newDoc, oldDoc, userCtx)
{ log(userCtx); }"}'
{"ok":true,"id":"_design/log","rev":"1-498bd568e17e93d247ca48439a368718"}
```

Comment on topic or styleLet's show the `validate_doc_update` function in pretty:

```
function(newDoc, oldDoc, userCtx) {
  log(userCtx);
}
```

Comment on topic or styleThis gets called for every future document update and it does nothing but print a log entry into CouchDB's log file. If we now create a new document…

```
> curl -X POST $HOST/somedatabase/ -d '{"a":1}'
{"ok":true,"id":"36174efe5d455bd45fa1d51efbcff986","rev":"1-
23202479633c2b380f79507a776743d5"}
```

Comment on topic or style…we should see this in our `couch.log` file:

```
[info] [<0.9973.0>] OS Process :: {"db": "somedatabase","name": "anna","roles":
["_admin"]}
```

Comment on topic or styleLet's prettify this again:

```
{
  "db": "somedatabase",
  "name": "anna",
  "roles": ["_admin"]
}
```

Comment on topic or styleWe see the current database, the name of the authenticated user and an array with `roles` with one role `"_admin"`. We can conclude that admin users in CouchDB are really just `regular users` with the `admin role` attached to them.

Comment on topic or styleBy separating users and roles from each other, the authentication system allows for flexible extension. For now, we just look at admin users.

Cookie Authentication

Comment on topic or styleBasic authentication which uses plain text passwords is nice and convenient, but not very secure if no extra measures are taken. It is also a very poor user experience. If you use basic authentication to identify admins, your application's users need to deal with an ugly, unstylable browser modal dialog that says *non-professional at work* more than anything else.

Comment on topic or styleTo remedy some of these concerns, CouchDB supports *cookie authentication*. With cookie authentication your application doesn't have to include the ugly login dialog the user's browsers comes with. You can use a regular HTML form to submit login to CouchDB. Upon receipt, CouchDB will generate a one-time token that the client can use in its next request to CouchDB. When CouchDB sees the token in a subsequent request, it will authenticate the user based on the token without the need to see the password again. By default, a token is valid for ten minutes.

Comment on topic or styleTo obtain the first token and thus to authenticate a user for the first time, the username and password must be sent to the `_session` API. The API is smart enough to decode HTML form submissions, so you don't have to resort to any smarts in your application.

Comment on topic or styleIf you are not using HTML forms to log in, you need to send an HTTP request that looks as if an HTML form generated it. Luckily, this is super simple:

```
> HOST="http://127.0.0.1:5984"
> curl -vX POST $HOST/_session \
  -H 'application/x-www-form-urlencoded' \
  -d 'username=anna&password=secret'
```

Comment on topic or styleCouchDB replies, and we'll give you some more detail:

```
< HTTP/1.1 200 OK
< Set-Cookie: AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-53hw1sJepVzcVxnriEw;
Version=1; Path=/; HttpOnly
> ...
<
{"ok":true}
```

Comment on topic or styleA `200` response code to tell us all is well, a `Set-Cookie` header including the token we can use for the next request and the standard JSON response to tell use again the request was successful.

Comment on topic or styleNow we can use this token to make another request as the same user without passing in the username and password again:

```
> curl -vX PUT $HOST/mydatabase \
  --cookie AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-53hw1sJepVzcVxnriEw \
  -H "X-CouchDB-WWW-Authenticate: Cookie" \
  -H "Content-Type: application/x-www-form-urlencoded"
{"ok":true}
```

Comment on topic or styleYou can keep using this token for ten minutes by default. After ten minutes you need ao authenticate your user again. The token lifetime can be configured with the `timeout` (in seconds) setting in the `couch_httpd_auth` configuration section.

Secrets

Comment on topic or stylePlease note that for cookie authentication to work, you need to enable the `cookie_authentication_handler` in your `local.ini`:

```
[httpd]
authentication_handlers = {couch_httpd_auth, cookie_authentication_handler}, \
  {couch_httpd_oauth, oauth_authentication_handler}, \
  {couch_httpd_auth, default_authentication_handler}
```

Comment on topic or styleIn addition, you need to define a *server secret*:

```
[couch_httpd_auth]
secret = yours3cr37pr4s3
```

Network Server Security

Comment on topic or styleCouchDB is a networked server and there are best practices around securing these that are beyond the scope of this book. The installation chapter includes some of these best practices. Make sure to understand the implications.

High Performance

Comment on topic or styleThis chapter will teach you the fastest ways to insert and query data with CouchDB. It will also explain why there is a wide range of performance across various techniques.

Comment on topic or styleThe take-home message: Bulk operations result in lower overhead, higher througput and more space efficiency. If you can’t work in bulk in your application, we’ll also describe other options to get throughput and space benefits. Finally, we describe interfacing directly with CouchDB from Erlang, which can be a useful technique if you want to integrate CouchDB storage with a server for non-HTTP protocols, like SMTP (email) or XMPP (chat).

Good Benchmarks are Non-Trivial

Comment on topic or styleEach application is different. Performance requirements are not always obvious. Different use cases need to tune different parameters. A classic trade-off is latency vs throughput. Concurrency is another factor. Many database platforms behave very differently with 100 clients, than they do with 1000 or more concurrent clients. Some data profiles require serialized operations, which increase total time (latency) for the client, and load on the server. We think simpler data and access patterns can make a big difference in the cacheability and scalability of your app, but we’ll get to that later.

Comment on topic or styleThe upshot: **real benchmarks require real-world load.** Simulating load is hard. Erlang tends to perform better under load (especially on multiple cores), so we’ve often seen test rigs that can’t drive CouchDB hard enough to see where it falls over.

Comment on topic or styleLets take a look what a typical web app looks like. This is not exactly how Craigslist works (because I don’t know how Craigslist works), but it is a close enough approximation to illustrate problems with benchmarking.

Comment on topic or styleYou have web server, some middleware, a database. A user request comes in, the web server takes care of the networking and parses the HTTP request. The request gets handed to the middleware layer which figures out what to run; then runs whatever is needed to serve the request. The middleware might talk to your database and other external resources like files or remote web services. The requests bounces back to the web server which sends out any resulting HTML. The HTML includes references to other resources living on your web server, like CSS-, JS- or image files and the process starts anew for every resource. A little different each time, but in general, all requests are similar. And along the way there are caches to store intermediate results to avoid expensive recomputation.

Comment on topic or styleThat's a lot of moving parts. Getting a top-to-bottom profile of all components to figure out where bottlenecks lie is pretty complex (but nice to have). I start making up numbers now, the absolute values are not important, only numbers relative to each other. Say a request takes 1.5 seconds (1500ms) to be fully rendered in a browser.

Comment on topic or styleIn a simple case like Craigslist there is the initial HTML, a CSS file, a JS file and the favicon. Except for the HTML, these are all static resources and involve reading some data from a disk (or from memory) and serve it to the browser who then renders it. The most notable things to do for performance are keeping data small (gzip compression, high jpg compression) and avoiding requests all together (HTTP level caching in the browser). Making the web server any faster doesn't buy us much (yeah, hand wavey, but I don't want to focus on static resources here. Pete wants his coffee. Let's say all static resources take 500ms to serve & render.

Comment on topic or style(Read all about improving client experience with proper use of HTTP from Steve Sounders. The YSlow tool is indispensable for tuning a web site.)

Comment on topic or styleThat leaves us with 1000ms for the initial HTML. We'll chop off 200ms for network latency [cf. Network Fallacies]. Let's pretend HTTP parsing, middleware routing & execution and database access share equally the rest of the time, 200ms each.

Comment on topic or styleIf you now set out to improve one part of the big puzzle that is your web app and gain 10ms in the database access time, this is probably time not well spent (unless you have the numbers to prove it).

Comment on topic or styleHowever, breaking down a single request like this, looking for how much time is spent in each component, is also misleading. Even if only a small percentage of the time is spent in your database under normal load, that doesn't teach you what will happen during traffic spikes. If all requests are hitting the same database, then any locking there could block many web requests. Your database may have minimal impact on total query time, under normal load, but under spike load it may turn into a bottleneck, magnifying the effect of the spike on the application servers. CouchDB can minimize this by dedicating an Erlang process to each connection, ensuring that all clients are handled, even if latency goes up a bit.

High Performance CouchDB

Comment on topic or styleNow that you see that database performance is only a small part of overall web performance, we'll give you some tips to squeeze the most out of CouchDB.

Comment on topic or styleCouchDB is designed from the ground up to service highly concurrent use cases, which make up the majority of web application load. However, sometimes we need to import a large batch of data into CouchDB, or initiate transforms across an entire database. Or maybe we're building a custom Erlang application that needs to link into CouchDB at a lower level than HTTP.

Hardware

Comment on topic or styleInvariably people will want to know what type of disk they should use, how much RAM, what sort of CPU. The real answer is that CouchDB is flexible enough to run on everything from a smart phone to a cluster, so the answers will vary alot.

Comment on topic or styleMore RAM is better, because CouchDB makes heavy use of the file system cache.

Comment on topic or styleCPU cores are more important for building views than serving documents.

Comment on topic or styleSSDs are pretty sweet because they can append to a file while loading old blocks, with a minimum of overhead. As they get faster and cheaper, they'll be really handy for CouchDB.

An Implementation Note

Comment on topic or styleWe're not going to rehash append-only B-Trees here, but understanding CouchDB's data format is key to gaining an intuition about which strategies yield the best performance. Each time an update is made, CouchDB loads from disk the B-tree nodes that point to the updated documents, or the key range where a new document's `_id` would be found.

Comment on topic or styleThis loading will normally come fromt the file system cache, except when updates are made to documents in regions of the tree that have not been touched in a long time. In those cases, the disk has to seek, which can block writing, and have other ripple effects. Preventing these disk seeks is the name of the game in CouchDB performance.

Comment on topic or styleWe'll have some numbers in this chapter, that come from a JavaScript test suite. It's not the most accurate, but the strategy it uses (counting the # of docs that can be saved in 10 seconds) makes up for the JavaScript overhead. The hardware the benchmarks were run on is modest: Just an old white MacBook Intel Core 2 Duo (remember those?).

Comment on topic or styleYou can run the benchmarks yourself by changing to the `bench/` directory of CouchDB's trunk, and running `./runner.sh` while CouchDB is running on port 5984.

Bulk Inserts and Mostly Monotonic DocIds

Comment on topic or styleBulk Inserts are the best way to have seekless writes.

Comment on topic or styleRandom IDs forces seeking after the file is bigger than can be cached.

Comment on topic or styleRandom IDs also make for a bigger file because in a large database you'll rarely have multiple documents in one B-Tree leaf.

Used By View and Replication

Comment on topic or styleIf you're curious what a good performance profile is for CouchDB, look at how views and replication are done. Triggerd replication applys updates to the database in large batches, to minimize disk chatter. Currently the 0.11.0 development trunk boasts an additional 3-5x speed increase over 0.10's view generation.

Comment on topic or styleViews load a batch of updates from disk, pass them through the view engine, and then write the view rows out. Each batch is of a few 100 documents, so the writer can take advantage of bulk efficiencies we see next.

Bulk Document Inserts

Comment on topic or styleThe fastest mode for importing data into CouchDB via HTTP is the `_bulk_docs` endpoint. Bulk docs accepts a collection of documents in a single `POST` request, and stores them all to CouchDB in a single index operation.

Comment on topic or styleBulk docs is the API to use when you are important a corpus of data using a scripting language. It can be 10 to 100 times faster than individual Bulk Updates, and is just as easy to work with from most languages.

Comment on topic or styleThe main factor influencing performance of bulk operations is the size of the update, both in terms of total data transfered, as well as the number of documents included in an update.

Comment on topic or styleHere are sequential bulk document inserts at four different granularities, from array of 100 docs, up through 1000, 5000, and 10k.

```
bulk_doc_100
4400 docs
437.37574552683895 docs/sec

bulk_doc_1000
17000 docs
1635.4016354016355 docs/sec

bulk_doc_5000
30000 docs
2508.1514923501377 docs/sec

bulk_doc_10000
30000 docs
2699.541078016737 docs/sec
```

Comment on topic or styleSo you can see that larger batches yield better performance, with an upper limit in this test of 2.7k docs/second. With larger documents, we might see that smaller batches are more useful. For references, all the docs look like this: `{"foo":"bar"}`

Comment on topic or style2700 docs / second is fine, but we want more power! Next up we'll explore running bulk docs in parallel.

Comment on topic or styleWith a different script (using bash and curl with `benchbulk.sh` in the same directory) I'm inserting large batches of documents in parallel to CouchDB. With batches of 1000 docs, ten at any given time, averaged over 10 rounds, I see about 3,650 docs / second on a MacBook Pro. Benchbulk also uses sequential ids.

Comment on topic or styleSo we see that with proper use of bulk docs and sequential ids, we can insert more than 3000 docs / second even using scripting languages.

Batch Mode

Comment on topic or styleTo avoid the indexing and disk sync overhead associated with individual document writes, there is an option that allows CouchDB to build up batches of documents in memory, flushing them to disk when a certain threshold has been reached, or when triggered by the user. The batch option does not give the same data integrity gauarantees that normal updates provide, so it should only be used when the potential loss of recently updates is acceptable.

Comment on topic or styleBecause batch mode only stores updates in memory until a flush occurs, updates which are saved to CouchDB directly proceeding a crash can be lost. By default CouchDB flushes the in-memory updates once per second, so in the worst case, data loss is still minimal. To reflect the reduced integrity gaurantees when `batch=ok` is used, the HTTP response code is 202 Accepted, as opposed to 201 Created.

Comment on topic or styleThe ideal use for batch mode is for logging type applications, where you have many distributed writers each storing discrete events to CouchDB. In a normal logging scenario, losing a few updates on rare occasions is worth the tradeoff for increased storage throughput.

Comment on topic or styleThere is a pattern for reliable storage using batch mode. It’s the same pattern as is used when data needs to be stored reliably to multiple nodes before acknowledging success to the saving client. In a nutshell, the application server (or remote client) saves to Couch A using `batch=ok`, and then watches update notifications from Couch B, only considering the save successful when Couch B’s `_changes` stream includes the relevant update. We cover this pattern in detail in the Replication chapter.

```
batch_ok_doc_insert
4851 docs
485.00299940011996 docs/sec
```

Comment on topic or styleThis JavaScript benchmark only gets around 500 docs/second, 6 times slower than the bulk document API. However, it has the advantage that clients don’t need to build up bulk batches.

Single Document Inserts

Comment on topic or styleNormal webapp load for CouchDB comes in the form of single document inserts. Because each insert comes from a distinct client, and has the overhead of an entire HTTP request and response, it has generally the lowest throughput for writes.

Comment on topic or styleProbably the slowest possible use case for CouchDB is the case of a writer which has to make many serialized writes against the database. Imagine a case where each write depends on the result of the previous write so that only one writer can run. This sounds like a bad case from the description alone. If you find yourself in this position, there are probably other problems to address as well.

Comment on topic or styleWe can write about 258 docs / second with a single writer in serial. (Pretty much worst-case scenario writer.)

```
single_doc_insert
2584 docs
257.9357157117189 docs/sec
```

Comment on topic or styleDelayed Commit (along with sequential UUIDs) is probably the most important CouchDB configuration setting for performance. When it is set to true (the default), CouchDB allows operations to be run again the disk without an explicit fsync after each operation. Fsync operations take time (the disk may have to seek, on some platforms the hard disk cache buffer is flushed, etc) so requiring an fsync for each update deeply limits CouchDB’s performance for non-bulk writers.

Comment on topic or styleDelayed commit should be left set to true in the configuration settings, unless you are in an environment where you absolutely need to know when updates have been recieved (eg, when CouchDB is running as part of a larger transaction). It is also possible to trigger an fsync (for instance after a few operations) using the `_ensure_full_commit` API.

Comment on topic or styleWhen delayed-commit is disabled CouchDB actually writes data to the actual disk before it responds to the client. (Except in batch=ok mode). It's a simpler code path, so it has less overhead when running at high throughput levels. However, for individual clients, it can seem slow. Here's the same benchmark in full-commit mode.

```
single_doc_insert
46 docs
4.583042741855135 docs/sec
```

Comment on topic or styleLook how slow `single_doc_insert` is with full-commit enabled. 4 or 5 docs / sec — wowsers! That's 100% a result of the fact that OSX has a real *fsync* so be thankful! Don't worry, the full-commit story gets better as we move into bulk operations.

Comment on topic or styleOn the other hand, we're getting better times for large bulks with delayed-commit off, which lets us know that *tuning for your application* will always bring better results than following a cookbook.

Hovercraft

Comment on topic or styleHovercraft is a library for accessing CouchDB from within Erlang. Hovercraft benchmarks should show the fastest possible performance of CouchDB's disk and index subsystems, as it avoids all HTTP connection and JSON conversion overhead.

Comment on topic or styleHovercraft is useful primarily when the HTTP interface doesn't allow for enough control, or is otherwise redunant. For instance, persisting Jabber instant messages to CouchDB might use ejabberd and Hovercraft. The easiest way to create a failure-tolerant message queue is probably a combination of RabbitMQ and Hovercraft.

Comment on topic or styleHovercraft was extracted from a from client project that used CouchDB to store massive amounts of email as document attachments. HTTP doesn't have an easy mechanism to allow a combination of bulk updates with binary attachments, so we used Hovercraft to connect an Erlang SMTP server directly CouchDB, to stream attachments directly to disk while maintaining the efficiency of bulk index updates.

Comment on topic or styleHovercraft includes a basic benchmarking feature, we see with that, that we can get a lot of docs per second.

```
> hovercraft:lightning().
Inserted 100000 docs in 9.37 seconds with batch size of 1000.
(10672 docs/sec)
```

Trade Offs

Comment on topic or styleTool X might give you 5ms response times and this is an order of magnitude faster than anything else on the market. Programming is all about trade-offs and everybody is bound by the same laws.

Comment on topic or styleOn the outside it might appear that everybody who is not using Tool X is a moron. But speed & latency are only part of the picture. We already established that going from 5ms to

50ms might not even be noticeable by anyone using your product. The expense for speed can be multiple things:

Comment on topic or styleMemory; instead of doing computations over and over, Tool X might have a cute caching layer that saves recomputation by storing results in memory. If you are CPU bound, that might be good, if you are memory bound it might not. A trade off.

Comment on topic or styleConcurrency; the clever data structures in Tool X are extremely fast when only one request at a time is processed, and because it is so fast most of the time, it appears as if it would process multiple request in parallel. Eventually though, a high number of concurrent requests fill up the request queue and response time suffers. — A variation on this is that Tool X might work exceptionally well on a single CPU or core, but not on many, leaving your beefy servers idling.

Comment on topic or styleReliability; making sure data is actually stored is an expensive operation. Making sure a data store is in a consistent state and not corrupted is another. There are two trade offs here: Buffers that store data in memory before committing it to disk to ensure a higher data throughput. In case of a power loss or crash (hard- or software), the data is gone. This may or may not be acceptable for your application. The other is a consistency check that is required to run after a failure. If you have a lot of data, this can take days. If you can afford to be offline, that's okay, but maybe you can't afford it.

Comment on topic or styleMake sure to understand what requirements you have and pick the tool that complies instead of taking the one that has the prettiest numbers. Who's the moron when your web application is offline for a fix up for a day and your customers impatiently wait to get their job done; or worse, you lose their data.

But… My Boss Wants Numbers!

Comment on topic or styleYeah, you want to know which one of these databases, caches, programming language, language constructs or tools are faster, harder, stronger. Numbers are cool and you can draw pretty graphs that management types can compare and make decisions from.

Comment on topic or styleFirst thing a good exec knows is that she's operating on insufficient data (aside, everybody does all the time, but sometimes it is just not apparent to you) and diagrams drawn from numbers are a very distilled view of reality. And graphs from numbers that are effectively made up by bad profiling are not much more than a fairy tale.

Comment on topic or styleIf you are going to produce numbers, make sure you understand how much is and isn't covered by your results. Before passing them on, make sure the receiving person knows as much. Again, the best thing to do is test with something as close to real-world load as possible. And this isn't easy.

A Call to Arms

Comment on topic or styleI'm in the market for databases and key-value stores. Every solution has a sweet spot in terms of data, hardware, setup and operation and there are enough permutations that you can pick the one that is closest to your problem. But how to find out? Ideally, you download & install all possible candidates, create a profiling test suite with proper testing data, make extensive tests and compare the results. This can easily take weeks and you might not have that much time.

Comment on topic or styleI would like to ask developers [*] of storage systems to compile a set of profiling suites that simulate different usage patterns of their system (read-heavy & write-heavy loads, fault tolerance, distributed operation and a lot more). A fault tolerance suite should include steps necessary to get data live again, like any rebuild or checkup time. I would like users of these systems to help their developers to find out how to reliably measure different scenarios.

Comment on topic or styleI'm working on CouchDB and I'd like to have such a suite very much!

Comment on topic or styleEven better, developers could agree (hehe) on a set of benchmarks that objectively measure performance for easy comparison. I know this is a lot of work and the results can still be questionable (you read the above part, did you?), but it'll help our users a great when figuring out what to use.

Recipes

Comment on topic or styleThis chapter shows some common tasks and how to solve them with CouchDB using best practices and easy to follow step-by-step instructions.

Banking

Comment on topic or styleBanks are serious business. They need serious databases to store serious transactions and serious account information. They can't lose any money. Ever. They can also not create money. A bank must be in balance. All the time.

Comment on topic or styleConventional wisdom says a database needs to support *transactions* to be taken seriously. CouchDB does not support transactions in the traditional sense (although it works transactional), so you could conclude CouchDB is not well suited to store bank data. Besides, would you trust your money to a couch? Well, we would. This chapter explains why.

Accountants don't Use Erasers

Comment on topic or styleSay you want to give $100 to your cousin Paul for the New York Cheesecake he sent to you. Back in the day, you had to travel all the way to New York and hand Paul the money or you could send it via (paper) mail. Both are considerably inconvenient so people started looking for alternatives. At one point, banks started to offer they'd take care of the money arriving at Paul's safely without headaches. Of course they charged for the convenience, but you're happy to pay a little fee if you could save a trip to New York. Behind the scenes though, the bank would send somebody with your money to give it to Paul's bank. The same procedure, but somebody else is dealing with the trouble. Also, banks sending money could be batched, instead of sending each order on its own, they could collect all transfers to New York over a week and send them over all in one go. In case of any problems, say the recipient is no longer a customer of the bank (remember, it took weeks to get from one coast to the other 150 years ago), the money was sent back to the originating account.

Comment on topic or styleEventually, the modern banking system was put in place and the actual sending money back and forth could be stopped (much to the disdain of road-thieves). Banks had money *on paper* which they could send around without actually sending valuables. The old concept is stuck in our heads though. To send somebody some of our money from our bank account, the bank needs to take the notes out of the account and bring it to the receiving account. But nowadays we're also used to things happen instantaneously. It just takes a few clicks to order goods from eBay and have them placed into the mail, why should a banking transaction take any longer?

Comment on topic or styleBanks are all electronic these days (and have been for a while). When we issue a money transfer, we expect it to go through immediately and we expect it to work in the way it worked back in the day. Take money from my account, add it to Paul's account, if anything goes wrong, put it back in my account. While this is logically what happens, that’s not how it works behind the scenes; and haven’t since way before computers were used for banking.

Comment on topic or styleWhen you go to your bank and ask it to send money to Paul, the accountant will *start a transaction* by noting down that you ordered the sending of the money. The transaction will include the date, amount and recipient. Remember that banks always need to be in balance. The money taken from your account cannot vanish. The accountant will move the money into an *in transit* account, that the bank maintains for you. Your account balance at this point is an aggregation of your current balance and the transactions in the in transit account. Now the bank goes and sees if Paul’s account is what you say it is and if the money could arrive there safely. If that’s the case, the money is moved in another single transaction from the in transit account to Paul’s account. Everything is in balance. Notice how there are multiple independent transactions and not one big transaction that combines a number of actions.

Comment on topic or styleNow let’s consider an error case: Paul’s account no longer exists. When the banks finds out while performing the batch operation of all the in transit transactions that need to be performed. A second transaction is generated that moves they money back from the in transit account to your bank account. Note that the transaction that moved the money off your account is *not* undone. A second transaction that does the reverse action is created.

Comment on topic or styleAnother error case is you not having the sufficient funds to send $100 to Paul. This will checked by the accountant (or software) before creating any money-deducting transaction.

Comment on topic or styleThis is where the title of this section comes into play. For accountability a bank cannot pretend an action didn’t happen, but has to record every action minutely in a log. Undoing is done explicitly by performing a reverse action, not by reverting or removing an existing transaction.

Comment on topic or styleThe title of this section quotes Pat Helland, a senior architect of transactional systems who worked at Microsoft and Amazon (read: he knows his shit).

Comment on topic or styleTo rehash the *transaction* (we know the terminology is a bit misleading here) of moving money between accounts is broken up into smaller transactions that can be guaranteed to succeed or fail and if they fail, reverse actions are taken. That way the task of guaranteeing money arriving that is hard if not broken up, becomes manageable.

Comment on topic or styleTurns out, these smaller transactions are possible to model in CouchDB. Above we mentioned that your account balance is an aggregated value. If we stick to this picture, things become downright easy. Instead of updating the balance of two accounts (yours and Paul’s, or yours and the in transit account), we simply create a single transaction document that describes what we’re doing and use a view to aggregate your account balance.

Comment on topic or styleLet’s consider a bunch of transactions:

```
...
{"from":"Jan","to":"Paul","amount":100}
{"from":"Paul","to":"Steve","amount":20}
{"from":"Work","to":"Jan","amount":200}
...
```

Comment on topic or styleSingle document writes in CouchDB are atomic. This guarantees that our bank is always in balance. There are many more transactions of course, but these will do for illustration purposes. How do we read the current account balance? Easy, create a Map/Reduce view:

Map Function

```
function(transaction) {
  emit(transaction.from, transaction.amount * -1);
  emit(transaction.to, transaction.amount);
}
```

Reduce Function

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

Comment on topic or styleDoesn't look too hard, does it? We'll store this in a view `balance` in a `_design/account` document. Let's find out Jan's balance:

```
curl 'http://127.0.0.1:5984/bank/_design/account/_view/balance?key="Jan"'
```

Comment on topic or styleCouchDB replies:

```
{"rows":[
{"key":null,"value":100}
]}
```

Comment on topic or styleLooks good!

Comment on topic or styleNow let's see if our bank is actually in balance. The sum of all transactions should be zero:

```
curl http://127.0.0.1:5984/bank/_design/account/_view/balance
```

Comment on topic or styleCouchDB replies:

```
{"rows":[
{"key":null,"value":0}
]}
```

Wrapping Up

Comment on topic or styleThis should explain that applications with strong consistency requirements can use CouchDB if it is possible to break up bigger transactions into smaller ones. A bank is a good enough approximation of a serious business, so that you can be safe modelling your important business logic into small CouchDB transactions.

Ordering Lists

Comment on topic or styleViews let you sort things by any value of your data, even complex JSON-keys are possible as we've seen in earlier chapters. By-data sorting is very useful to allow users to find things quickly, a name is much easier to find in a list of names that is sorted alphabetically. Humans naturally resort to a divide-an-conquer algorithm (sounds familiar? :) and don't consider a large part of the input set because they know the name won't show up there. Likewise, sorting by number and date helps a great deal to let users manage their ever increasing amounts of data.

Comment on topic or styleThere's another sorting type that is a little more fuzzy. Search engines show you results in order of relevance. That relevance is what the search engine thinks is most relevant to you given your search term (and potential search and surfing history). There are other systems trying to infer from earlier data what is most relevant to you, but they have the near-to-impossible task to guess what a user is interested in. Computers are notoriously bad at guessing.

Comment on topic or styleThe easiest way for a computer to figure out what’s most relevant for a user is letting the user prioritize things. Take a todo application for example: it allows users to reorder todo items so they know what they need to work on next. The underlying problem, keeping a user-defined sorting order can be found in a number of other places.

A List of Integers

Comment on topic or styleLet’s stick with the todo application. The naïve approach is pretty easy: with each todo item we store an integer that specifies the location in a list. We use a view to get all todo items in the right order.

Comment on topic or styleFirst, we need some example documents:

A Bunch of Todo Item Documents

```
{
  "title":"Remember the Milk",
  "date":"2009-07-22T09:53:37",
  "sort_order":2
}

{
  "title":"Call Fred",
  "date":"2009-07-21T19:41:34",
  "sort_order":3
}

{
  "title":"Gift for Amy",
  "date":"2009-07-19T17:33:29",
  "sort_order":4
}

{
  "title":"Laundry",
  "date":"2009-07-22T14:23:11",
  "sort_order":1
}
```

Comment on topic or styleNext, we create a view with a simple map function that emits rows that are then sorted by the `sort_order` field of our documents. The view’s result looks like we’d expect it.

A View to Sort Todo Items

```
function(todo) {
  if(todo.sort_order && todo.title) {
    emit(todo.sort_order, todo.title);
  }
}
```

The Views' Result

```
{
  "total_rows": 4,
  "offset": 0,
  "rows": [
    {
      "key":1,
      "value":"Laundry",
      "id":"..."
    },
    {
      "key":2,
      "value":"Remember the Milk",
      "id":"..."
    },
    {
      "key":3,
      "value":"Call Fred",
      "id":"..."
    },
    {
      "key":4,
      "value":"Gift for Amy",
      "id":"..."
    }
  ]
}
```

Comment on topic or styleThat looks reasonably easy but can you spot the problem? Here's a hint: what do you have to do if getting a gift for Amy becomes a higher priority than the milk? Conceptually, the work required is simple:

1. Comment on topic or style Assign "Gift for Amy" the `sort_order` of "Remember the Milk".
2. Comment on topic or style Increment the `sort_order` of "Remember the Milk" and **all** items that follow by one.

Comment on topic or styleUnder the hood, this is a lot of work. With CouchDB you'd have to load every document, increment the `sort_order` and save it back. If you have a lot of todo items (I do), then this is some significant work. Maybe there's a better approach.

A List of Floats

Comment on topic or styleThe fix is simple: Instead of using an integer to specify the sort order, we use a float:

```
{
  "title":"Remember the Milk",
  "date":"2009-07-22T09:53:37",
  "sort_order":0.2
}

{
  "title":"Call Fred",
  "date":"2009-07-21T19:41:34",
  "sort_order":0.3
}

{
  "title":"Gift for Amy",
  "date":"2009-07-19T17:33:29",
  "sort_order":0.4
}

{
  "title":"Laundry",
  "date":"2009-07-22T14:23:11",
  "sort_order":0.1
}
```

Comment on topic or styleThe view stays the same. Reading this is as easy as the previous approach. Reordering becomes much easier now. The application frontend can keep a copy of the `sort_order` values around, so when we move an item and store the move, we do not only have available the new position but also the `sort_order` value for the two new surrounding items.

Comment on topic or styleLet’s move "Gift for Amy" above "Remember the Milk". The surrounding `sort_order` in the target position are `0.1` and `0.2`. To store "Gift for Amy" with the correct `sort_order`, we simply use the median of the two surrounding values: `(0.1 + 0.2) / 2 = 0.3 / 2 = 0.15`.

Comment on topic or styleIf we query the view again now we get the desired result:

The New Views Result

```
{
  "total_rows": 4,
  "offset": 0,
  "rows": [
    {
      "key":0.1,
      "value":"Laundry",
      "id":"..."
    },
    {
      "key":0.15,
      "value":"Gift for Amy",
      "id":"..."
    },
    {
      "key":0.2,
      "value":"Remember the Milk",
      "id":"..."
    },
    {
      "key":0.3,
      "value":"Call Fred",
      "id":"..."
    }
  ]
}
```

Comment on topic or styleThe downside of this approach is that with an increasing number of reorderings, float precision can become an issue as digits “grow” infinitely. One solution is to not care and expect that a single user will not exceed any limits. Alternatively, an administrative task can reset the whole list to single-decimals when a user is not active.

Comment on topic or styleThe advantage of this approach is that you only have to touch a single document which is efficient for storing the new ordering of a list and updating the view that maintains the ordered index since only only the changed document has to be incorporated into the index.

Pagination

Comment on topic or styleThis receipt explains how to paginate over view results. Pagination is a user interface (UI) pattern that allows the display of a large number of rows (the *result set*) without loading all the rows into the UI at once. A fixed-size subset, the *page*, is displayed along with *next* and *previous* links or buttons that can move the *viewport* over the result set to a adjacent page.

Comment on topic or styleWe assume you're familiar with creating and querying documents and views as well as the multiple view query options.

Example data

Comment on topic or styleTo have some data to work with, we're creating a list of bands. A document per band:

```
{ "name":"Biffy Clyro" }

{ "name":"Foo Fighters" }

{ "name":"Tool" }

{ "name":"Incubus" }

{ "name":"Helmet" }

{ "name":"The Fuckin Hate" }

{ "name":"Future of the Left" }

{ "name":"A Perfect Circle" }

{ "name":"Silverchair" }

{ "name":"Queens of the Stone Age" }

{ "name":"Kerub" }
```

A View

Comment on topic or styleWe need a simple map function that gives us an alphabetical list of band names. This should be easy, but we're adding extra smarts to filter out "The" and "A" in front of band names to put them into the right position.

```
function(doc) {
  if(doc.name) {
    var name = doc.name.replace(/^(A|The) /, "");
    emit(name, doc);
  }
}
```

Comment on topic or styleThe views result is an alphabetical list of band names. Now say we want to display band names five at a time and have a link pointing to the next five names that make up one page; and a link for the previous five, if we're not on the first page.

Comment on topic or styleWe already learned how to use the `startkey`, `limit`, and `skip` parameters in earlier chapters. We'll use these again here. First, let's have a look at the full result set:

```
{"total_rows":11,"offset":0,"rows":[
{"id":"a0746072bba60a62b01209f467ca4fe2","key":"Biffy Clyro","value":null},
{"id":"b47d82284969f10cd1b6ea460ad62d00","key":"Foo Fighters","value":null},
{"id":"45ccde324611f86ad4932555dea7fce0","key":"Fuckin Hate","value":null},
{"id":"d7ab24bb3489a9010c7d1a2087a4a9e4","key":"Future of the Left","value":null},
{"id":"ad2f85ef87f5a9a65db5b3a75a03cd82","key":"Helmet","value":null},
{"id":"a2f31cfa68118a6ae9d35444fcb1a3cf","key":"Incubus","value":null},
{"id":"67373171d0f626b811bdc34e92e77901","key":"Kerub","value":null},
{"id":"3e1b84630c384f6aef1a5c50a81e4a34","key":"Perfect Circle","value":null},
{"id":"84a371a7b8414237fad1b6aaf68cd16a","key":"Queens of the Stone
Age","value":null},
{"id":"dcdaf08242a4be7da1a36e25f4f0b022","key":"Silverchair","value":null},
{"id":"fd590d4ad53771db47b0406054f02243","key":"Tool","value":null}
]}
```

Set Up

Comment on topic or styleThe mechanics of paging are very simple:

- Comment on topic or style Display first page.
- Comment on topic or style If there are more rows to show, show *next* link.
- Comment on topic or style Draw subsequent page
- Comment on topic or style If this is not the first page, show a *previous* link.
- Comment on topic or style If there are more rows to show, show *next* link.

Comment on topic or styleOr in a pseudo-JavaScript snippet:

```
var result = new Result();
var page = result.getPage();

page.display();

if(result.hasPrev()) {
  page.display_link('prev');
}

if(result.hasNext()) {
  page.display_link('next');
}
```

Slow Paging (Don't)

Comment on topic or styleDon't use this method! We just show it because it might seem natural to use this and you do want to know why it is a bad idea. To get the first five rows from our view result, you use the `?limit=5` query parameter:

```
curl -X GET http://127.0.0.1:5984/artists/_design/artists/_view/by-name?limit=5
```

Comment on topic or styleThe result:

```
{"total_rows":11,"offset":0,"rows":[
{"id":"a0746072bba60a62b01209f467ca4fe2","key":"Biffy Clyro","value":null},
{"id":"b47d82284969f10cd1b6ea460ad62d00","key":"Foo Fighters","value":null},
{"id":"45ccde324611f86ad4932555dea7fce0","key":"Fuckin Hate","value":null},
{"id":"d7ab24bb3489a9010c7d1a2087a4a9e4","key":"Future of the Left","value":null},
{"id":"ad2f85ef87f5a9a65db5b3a75a03cd82","key":"Helmet","value":null}
]}
```

Comment on topic or styleBy comparing the `total_rows` value to our `limit` value, we can determine if there are more pages to display. We also know by the `offset` member that we are on the first page. We can also calculate the value for `skip=` to get the results for the next page:

```
[source,javascript]
var rows_per_page = 5;
var page = (offset / rows_per_page) + 1; // == 1
var skip = page * rows_per_page; // == 5 for the first page, 10 for the second ...
```

Comment on topic or styleSo we query CouchDB with:

```
curl -X GET 'http://127.0.0.1:5984/artists/_design/artists/_view/by-name?
limit=5&skip=5'
```

Comment on topic or styleNote we had to use `'` (quotes), to escape the `&` character, that is special to the shell we execute `curl` in.

Comment on topic or styleThe result:

```
{"total_rows":11,"offset":5,"rows":[
{"id":"a2f31cfa68118a6ae9d35444fcb1a3cf","key":"Incubus","value":null},
{"id":"67373171d0f626b811bdc34e92e77901","key":"Kerub","value":null},
{"id":"3e1b84630c384f6aef1a5c50a81e4a34","key":"Perfect Circle","value":null},
{"id":"84a371a7b8414237fad1b6aaf68cd16a","key":"Queens of the Stone
Age","value":null},
{"id":"dcdaf08242a4be7da1a36e25f4f0b022","key":"Silverchair","value":null}
]}
```

Comment on topic or styleImplementing the `hasPrev()` and `hasNext()` methods is pretty straightforward:

```
function hasPrev()
{
  return page > 1;
}

function hasNext()
{
  var last_page = Math.floor(total_rows / rows_per_page) +
    (total_rows % rows_per_page);
  return page != last_page;
}
```

The Dealbreaker

Comment on topic or styleThis all looks easy and straightforward, but it has one fatal flaw. Remember how view results are generated from the underlying b-tree index: CouchDB jumps to the first row (or the first row that matches `startkey`, if provided) and reads one row after the other from the index until there are no more rows (or `limit` or `endkey` match, if provided).

Comment on topic or style`skip` works like this: In addition to going to the first row and start reading, `skip` will skip as many rows as specified, but CouchDB will still read from the first row, it just won't return any values for the skipped rows. If you specify `skip=100`, CouchDB will read 100 rows and not create output for them. This doesn't sound too bad, but it is *very* bad, when you use `1000` or even `10000` as `skip` values. CouchDB will have to look at a lot of rows unnecessarily.

Comment on topic or styleAs a rule of thumb, skip should only be used with single digit values. While possible that there are legit use-cases where you specify a larger value, they are a good indicator for potential problems with your solution.

Comment on topic or styleFinally, for the calculations to work, you need to add a reduce function and make two calls to the view per page to get all the numbering right and there's still a potential for error.

Fast Paging (Do)

Comment on topic or styleThe correct solution is not much harder either. Instead of slicing the result set into equally sized pages, we look at ten rows at a time and use `startkey` to jump to the next ten rows. We even use `skip`, but only with the value `1`.

Comment on topic or styleHere is how it works:

- Comment on topic or style Request `rows_per_page` + 1 rows from the view
- Comment on topic or style Display `rows_per_page` rows, store + 1 row as `next_startkey` and `next_startkey_docid`
- Comment on topic or style As *page information*, keep `startkey` and `next_startkey`
- Comment on topic or style Use the `next_*` values to create the next link, use the others to create the previous link

Comment on topic or styleThe trick to find the next page is pretty simple. Instead of requesting ten rows for a page, you request eleven rows, but only display ten and use the values in the eleventh row as the `startkey` for the next page. Populating the link to the previous page is as simple as carrying the current `startkey` over to the next page. If there's no previous `startkey`, we are on the first page. We stop displaying the link to the next page if we get `rows_per_page` or less rows back.

Comment on topic or styleThis is called *linked list pagination*, as we go from page to page, or list item to list item, instead of jumping directly to a pre-computed page. There is one caveat though. Can you spot it?

Comment on topic or styleCouchDB view keys do not have to be unique, you can have multiple index entries *red*. What if you have more index entries for a key than rows that should be on a page? `startkey` jumps to the first row and you'd be screwed if CouchDB wouldn't have an additional parameter for you to use. All view keys with the same value are internally sorted by `docid`, that is, the id of the document that created that view row. You can use the `startkey_docid` and `endkey_docid` parameters to get subsets of these rows. For pagination, we still don't need `endkey_docid`, but `startkey_docid` is very handy. In addition to `startkey` and `limit`, you also use `startkey_docid` for pagination if, and only if, the extra row you fetch to find the next page has the same key as the current startkey.

Comment on topic or styleIt is important to note that the `*_docid` parameters only work *in addition* to the `*key` parameters and are only useful to further narrow down the result set of a view for a single key. They do not work on their own (the one exception being the built-in _all_docs view that already sorts by document id).

Comment on topic or styleThe advantage of this approach is that all they key operations can be performed on the super fast b-tree index behind the view. Looking up a page doesn't include scanning through hundreds and thousands of rows unnecessarily.

Jump to Page

Comment on topic or styleOne drawback of the linked list style pagination is that because you can't pre-compute the rows for a particular page from the page number and the rows per page. Jumping to a specific page doesn't really work. Our gut reaction, if that concern is raised is: "Not even Google is doing that!" and we tend to get away with that. Google always pretends on the first page to find 10 more pages of results. Only if you click on the second page (something very few people actually do), Google might display a reduced set of pages. If you page through the result you get links for the previous and next 10 pages, but no more. Pre-computing the necessary `startkey` and `startkey_docid` for 20 pages is a feasible operation and a pragmatic optimization to know the rows for every page in a result set that is potentially tens of thousands of rows long, or more.

Comment on topic or styleIf you really do need jump to page over the full range of documents (we have seen applications require that), you can still maintain an integer value index as the view index and have a hybrid approach at solving pagination.

Installing on Unix-like systems

Debian GNU/Linux

Comment on topic or styleYou can install the CouchDB package by running:

Comment on topic or stylesudo aptitude install couchdb

Comment on topic or styleWhen this completes, you should have a copy of CouchDB running on your machine. Be sure to read through the Debian specific system documentation that can be found under `/usr/share/couchdb`.

Ubuntu

Comment on topic or styleYou can install the CouchDB package by running:

Comment on topic or stylesudo aptitude install couchdb

Comment on topic or styleWhen this completes, you should have a copy of CouchDB running on your machine. Be sure to read through the Ubuntu specific system documentation that can be found under `/usr/share/couchdb`.

Gentoo Linux

Comment on topic or styleEnable the development ebuild of CouchDB by running:

Comment on topic or stylesudo echo dev-db/couchdb >> /etc/portage/package.keywords

Comment on topic or styleCheck the CouchDB ebuild by running:

Comment on topic or styleemerge -pv couchdb

Comment on topic or styleBuild and install CouchDB ebulid by running:

Comment on topic or stylesudo emerge couchdb

Comment on topic or styleWhen this completes, you should have a copy of CouchDB running on your machine.

Problems

Comment on topic or styleSee the Installing from Source chatper if your distribution doesn't have a CouchDB package.

JSON Primer

Comment on topic or styleCouchDB uses *JavaScript Object Notation (JSON)* for data storage, a lightweight format based on a subset of JavaScipt syntax. One of the best bits about JSON is that it's easy to read and write by hand, much more so than something like XML. We can parse it naturally with JavaScript, because it shares part of the same syntax. This comes in really handy when we're building dynamic Web applications and we want to fetch some data from the server.

Comment on topic or styleHere's a sample JSON document:

```
{
    "Subject": "I like Plankton",
    "Author": "Rusty",
    "PostedDate": "2006-08-15T17:30:12-04:00",
    "Tags": [
        "plankton",
        "baseball",
        "decisions"
    ],
    "Body": "I decided today that I don't like baseball. I like plankton."
}
```

Comment on topic or styleYou can see that the general structure is based around key/value pairs, and lists of things. There's a few extra things we need to know though, before we can call ourselves JSON masters. JSON has a number of basic data types that you can use. We'll cover them all in this appendix.

Data Types

Numbers

Comment on topic or styleYou can have positive integers:

```
"Count": 253
```

Comment on topic or styleOr negative integers:

```
"Score": -19
```

Comment on topic or styleOr floating point numbers:

```
"Area": 456.31
```

Comment on topic or styleOr scientific notation:

```
"Density": 5.6e+24
```

Comment on topic or styleThere is a subtle, but important, difference between floating point numbers and decimals. When you use a number like `15.7`, this will be interpreted as `15.699999999999999` by most clients, which may be problematic for your application. For this reason, currency values are usually better represented as strings in JSON. A string like `"15.7"` will be interpreted as `"15.7"` by every JSON client.

Strings

Comment on topic or styleYou can use strings for values:

```
"Author": "Rusty"
```

Comment on topic or styleYou have to escape some special characters, like tabs or new lines:

```
"poem": "May I compare thee to a\n\tsalty plankton."
```

Comment on topic or styleThe JSON site has details on what needs to be escaped.

Booleans

Comment on topic or styleYou can have boolean true values:

```
"Draft": true
```

Comment on topic or styleOf boolean false values:

```
"Draft": false
```

Arrays

Comment on topic or styleAn array is a list of values:

```
"Tags": ["plankton", "baseball", "decisions"]
```

Comment on topic or styleAn array can contain any other data type, including arrays:

```
"Context": ["dog", [1, true], {"Location": "puddle"}]
```

Objects

Comment on topic or styleAn object is a list of key/value pairs:

```
{"Subject": "I like Plankton", "Author": "Rusty"}
```

Nulls

Comment on topic or styleYou can have null values:

```
"Surname": null
```

The Power of B-Trees

Comment on topic or styleCouchDB uses a data-structure called a B-Tree to index its documents and views. We'll look at B-Trees enough to understand the types of queries they support, and how they are a good fit for CouchDB.

Comment on topic or styleThis is our first foray into CouchDB internals. To use CouchDB, you don't need to know what's going on under the hood, but if you understand how CouchDB performs its magic you'll be able to pull tricks of your own. Additionally, if you understand the consequences of your ways of using CouchDB you will end up with smarter systems.

Comment on topic or styleIf you were not looking closely, CouchDB would appear to be a B-tree manager with an HTTP interface.

CouchDB is actually using a B+-tree which is a slight variation of the B-tree that trades a bit of (disk-) space for speed. When we say *B-tree* we mean CouchDB's *B+-tree*.

Comment on topic or styleA B-tree is an excellent data structure to store huge amounts of data for fast retrieval. Millions and billions of items in a B-tree is where they get fun. B-trees are usually a shallow but wide data structure. While other trees can grow very high a typical B-tree has a single-digit height even with millions of entries. This is particularly interesting for CouchDB where the leaves of the tree are stored on a slow medium like a hard drive. Accessing any part of the tree for reading or writing requires visiting only a few nodes which translates to a few head seeks (which are what makes a hard drive slow) and because the operating system is likely to cache the upper tree nodes anyway, only the seek to the final leaf node is needed.

Comment on topic or styleFrom a practical point of view B-trees, therefore, guarantee an access time of less than 10 ms even for extremely large datasets.

— *Dr. Rudolf Bayer, Inventor of the B-tree*

Comment on topic or styleCouchDB's B-tree implementation is a bit different from the original. While it maintains all of the important properties, it adds Multi Version Concurrency Control (MVCC) and an append-only design. B-trees are used to store the main database file as well as view indexes. One database is one B-tree and one view index is one B-tree.

Comment on topic or styleMVCC allows concurrent reads and writes without using a locking system. Writes are serialized, allowing only one write operation at any point in time, for any single database. Write operations do not block reads and there can be any number of read operations at any time. Each read operation is guaranteed a consistent view of the database, How this is accomplished, is at the core of CouchDB's storage model.

Comment on topic or styleThe short answer is that because CouchDB uses append-only files, the B-tree root node must be rewritten every time the file is updated. However, old portions of the file will never change, so every old B-tree root, should you happen to have a pointer to it, will also point to a consistent snapshot of the database.

Comment on topic or styleEarly in the book we explained how the MVCC system uses the document's `_rev` value to ensure that only one person can change a document version. The B-tree is used to look up the existing `_rev` value for comparison. By the time a write is accepted, the B-tree can expect it to be an authoritative version.

Comment on topic or styleSince old versions of documents are not overwritten or deleted when new versions come in, requests that are reading a particular version do not care if new ones are written at the same time. With an often changing document, there could be readers reading three different versions at the same time. Each version was the latest one when a particular client started reading it, but new versions were being written. From the point when a new version is *committed*, new readers will read the new version while old readers keep reading the old version.

Comment on topic or styleIn a B-tree, data is only kept in leaf nodes. CouchDB B-trees only append data to the database file that keeps the B-tree on disk and only grows at the end. Add a new document? The file grows at the end. Delete a document? That gets recorded at the end of the file. The consequence is a robust database file. Computers fail and there are plenty of reasons like power-loss or failing hardware. Since CouchDB does not overwrite any existing data, it cannot corrupt anything that has been written and *committed* to disk already.

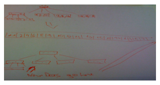


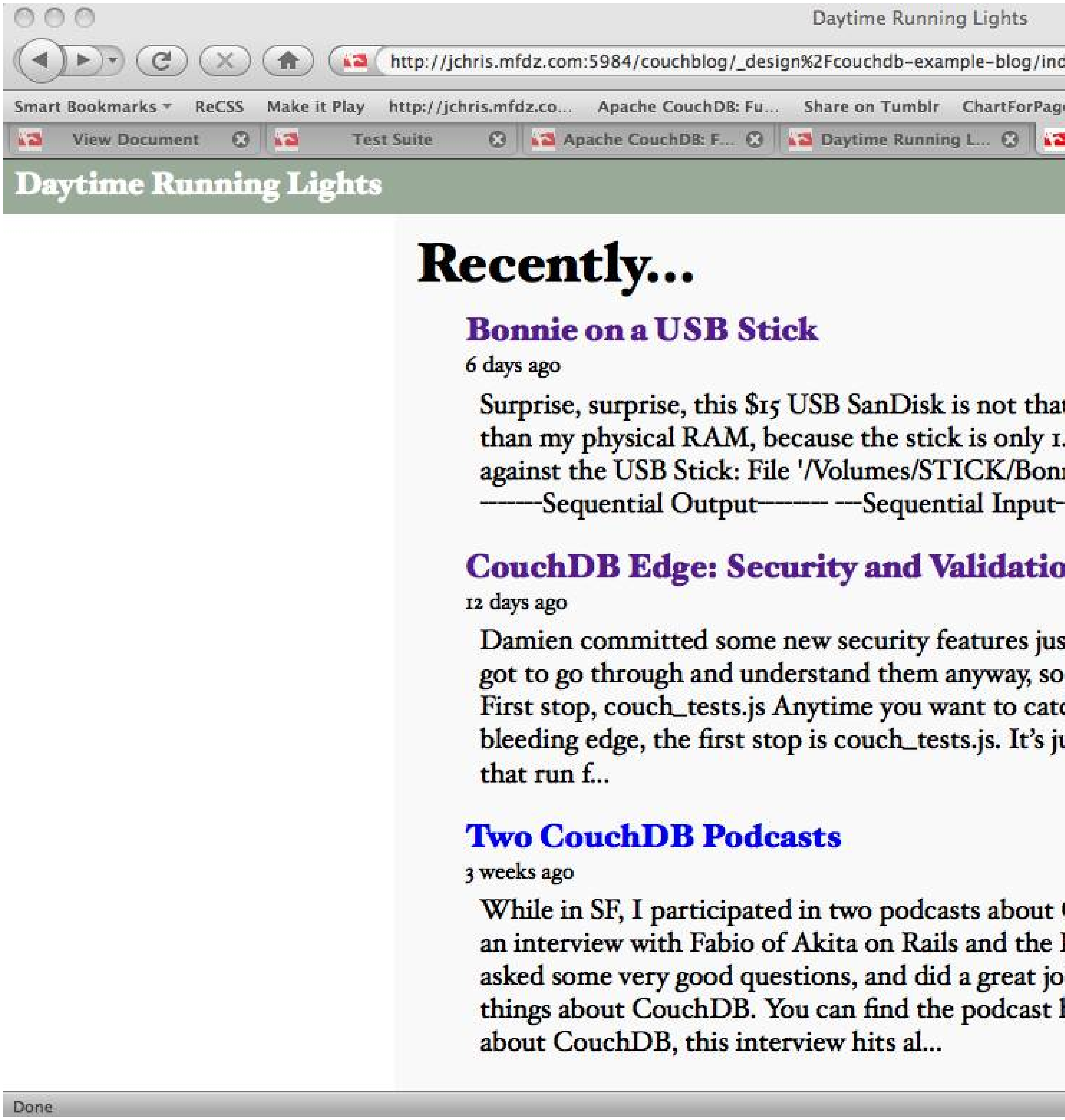
Figure: Flat B-tree and Append Only

Comment on topic or styleCommitting is the process of updating the database file to reflect changes. This is done in the file *footer* which is the last 4k of the database file. The footer is 2k in size and written twice in succession. First CouchDB appends any changes to the file and then records the file's new length in the first database footer. It then force-flushes all changes to disk. It then copies the first footer over to the second 2k of the file and force-flushes again.

Comment on topic or styleIf anywhere in this process, a problem occurs, say power is cut off and CouchDB is restarted later, the database file is in a consistent state and doesn't need a check-up. CouchDB starts reading the database file backwards. When it finds a footer-pair it makes some checks: if the first 2k are corrupt (a footer includes a checksum), CouchDB replaces it with the second footer and all is well. If the second footer is corrupt, CouchDB copies the first 2k over and all is well again. Only once both footers are flushed to disk successfully will CouchDB acknowledge that a write operation was successful. Data is never lost and data on disk is never corrupted. This design is the reason for CouchDB having no *off*-switch. You just terminate it when you are done.

Comment on topic or styleThere's a lot more to say about B-trees in general and if and how Solid State Drives (SSDs) change the runtime behaviour. The Wikipedia article on B-tree is a good starting point for further investigations. The Scholarpedia (a Wikipedia-like website with peer-reviewed articles) includes notes by Dr. Rudolf Bayer, Inventor of the B-tree.

The Final Result [#]



Daytime Running Lights

http://jchris.mfdz.com:5984/couchblog/_design%2Fcouchdb-example-blog/ind

Smart Bookmarks ReCSS Make it Play http://jchris.mfdz.co... Apache CouchDB: Fu... Share on Tumblr ChartForPag

View Document Test Suite Apache CouchDB: F... Daytime Running L...

Daytime Running Lights

Recently...

Bonnie on a USB Stick

6 days ago

Surprise, surprise, this $15 USB SanDisk is not that
than my physical RAM, because the stick is only 1.
against the USB Stick: File '/Volumes/STICK/Bon
------Sequential Output------ ---Sequential Input-

CouchDB Edge: Security and Validatio

12 days ago

Damien committed some new security features jus
got to go through and understand them anyway, so
First stop, couch_tests.js Anytime you want to cat
bleeding edge, the first stop is couch_tests.js. It's j
that run f...

Two CouchDB Podcasts

3 weeks ago

While in SF, I participated in two podcasts about
an interview with Fabio of Akita on Rails and the
asked some very good questions, and did a great jo
things about CouchDB. You can find the podcast
about CouchDB, this interview hits al...

Done

Screenshot: The Rendered Index Page

Comment on topic or styleThis is our final list of blog posts. That wasn't too hard, was it? We now have the front page of the blog, we know how to query single documents as well as views and how to pass arguments to views. We'll go on describing how you can accept reader comments on your blog.