CouchDB Database for the Web

Contents

Preface

CouchDB, is an open source database that focuses on ease of use and on being “a database that completely embraces the web”. It is a NoSQL database that uses JSON to store data, JavaScript as its query language using MapReduce, and HTTP for an API. One of its distinguishing features is multi-master replication. CouchDB was first released in 2005 and later became an Apache project in 2008.

This is a hands-on course on CouchDB. You will learn how to install and configure CouchDB and how to perform common operations with it.

Additionally, you will build an example application from scratch and then finish the course with more advanced topics like scaling, replication and load balancing.

About the Author

Born in Kolkata, India in 1977, Piyas De made a headstrong effort to learn, develop, deliver, teach and share his knowledge on different type of software languages and technologies especially on Java/J2EE and related open source technologies.

Being A Sun Microsystems Certified Enterprise Architect with more than 10 long years of professional IT experience in various areas such as Architecture Definition, Define Enterprise Application, Client-server/ e-business solutions, he possess hands on experience to handle a wide range of database ranging from PostGreSQL, SQL Server7.0/2000, Oracle 8i, 10g to Sybase, MySQL and NoSQL databases like MongoDB.

CMM Level 3 Process orientation proved to be a major turning point for him as Project Manager as it has given him the opportunity to explore various languages or frameworks - to name a few GWT, Struts, Spring, Hibernate, Tiles, Oracle ADF, J2EE (Java), PL/SQL etc.

Some of his career's executed projects are the following:

- subscriptions.abp.in - a media company subscription portal
- healthscribes.com - a doctor's and patient's portal
- Social Media Mashup Project - Revvo (ongoing)
- Health Care Solution for Government Authorites
- NoSQL usage in server creation as per PRODML specification

He learns and writes about different aspects of open source technologies like Angular.js, Node.js, MongoDB, Google DART, Apache Lucene, Text Analysis with GATE and related Big Data technologies in his blog (www.phloxblog.in).

Apart from his professional excellence, he is happily married with Ketaki and has a son named Titas. Also, he is an enthusiast in the field of teaching and a humble book worm who takes immense pleasure reading books not only on technologies but also on humour, suspense, comedy and many more. Impeccable affinity towards knowing the distant corners of technologies became the actual force of penning down fresh technological outlooks.

Chapter 1

Installation - How to Install CouchDB

1.1 Introduction

Apache CouchDB is an open source NoSQL database that uses JSON to store data, JavaScript as its query language, also uses MapReduce, and HTTP for an API. In CouchDB, each database is a collection of documents. Each document maintains its own data and self-contained schema. An application may access multiple databases on different servers. Document metadata contains the necessary revision information to make the merging possible in case the databases were disconnected.

In CouchDB, every document has a unique id and there is no required document schema. CouchDB can handle a high volume of concurrent readers and writers without conflict. Stored data is structured using views. Each view is constructed by a JavaScript function that acts as the Map. The function takes a document and transforms it into a single value which it returns. CouchDB can index views and keep those indexes updated as documents are added, removed, or updated.

CouchDB is designed with replication and off-line operation in mind. Here, multiple replicas can have their copies of the same data, modify it, and then sync those changes at a later time. All operations have a unique URI that gets exposed via HTTP. REST APIs use the HTTP methods POST, GET, PUT and DELETE for the four basic CRUD (Create, Read, Update, Delete) operations on all resources.

CouchDB is able to replicate to devices that can go offline and handle data sync when the devices get back online.

1.2 Couch DB Installation on Mac OS:

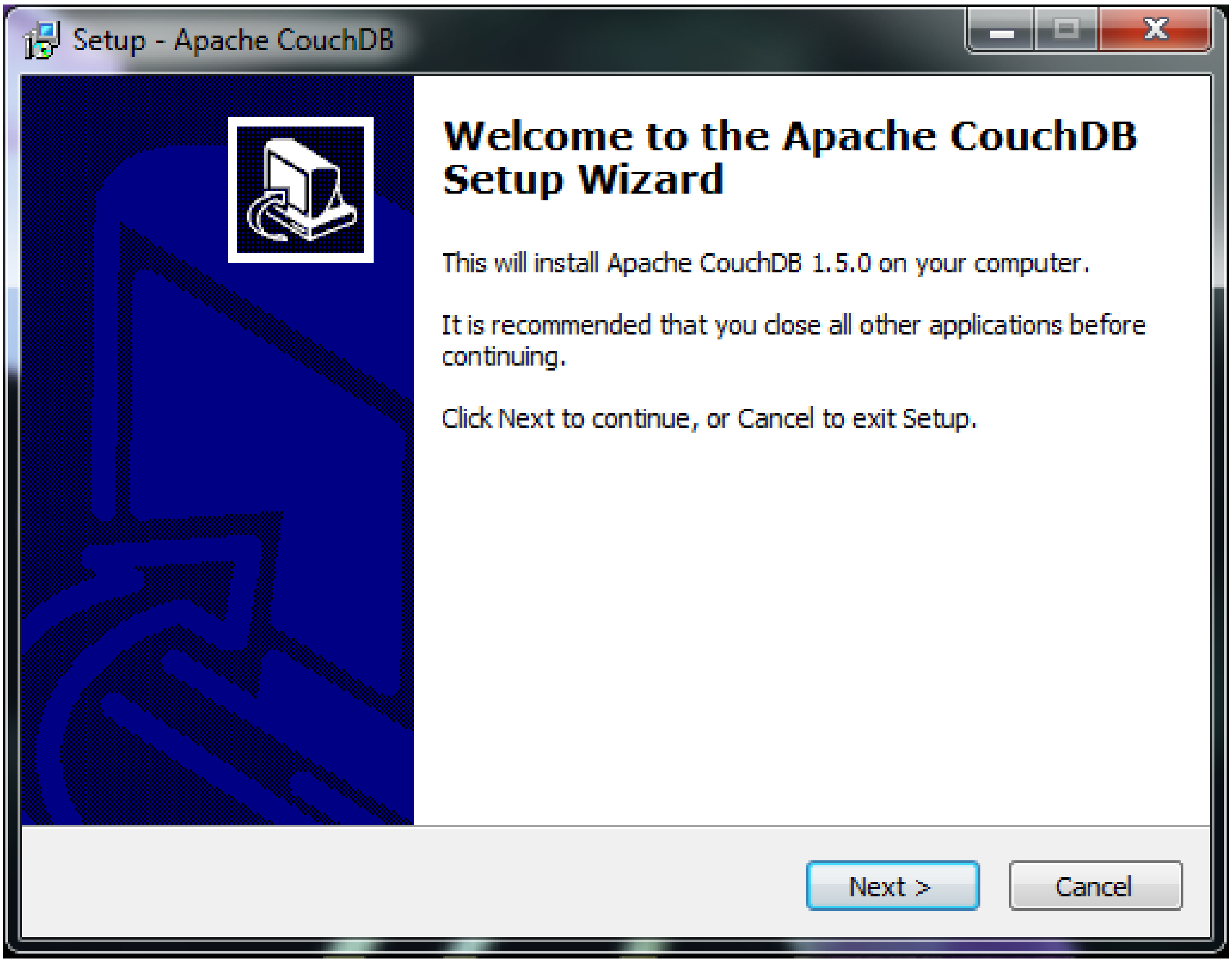
To install CouchDB on a Mac machine, follow the steps given below:

- Step 1: Download the Apache Couch DB first. Click on the following link, http://couchdb.apache.org/
- Step 2: Click on the download button to get the latest version of CouchDB.
- Step 3: Download the version according to MAC OS platform.
- Step 4: It will download a .zip file.
- Step 5: Extract the .zip file.
- Step 6: Copy the CouchDb file & paste it in the application folder of your Mac OS Machine.
- Step 7: Run Apache CouchDB application.
- Step 8: To check the installation, Go to the url, http://localhost:5984/_utils/ to view the CouchDB Admin console.

1.3 Couch DB Installation on Windows:

- Get the latest Windows binaries from the CouchDB web site (http://couchdb.apache.org/).

Follow the installation wizard steps:



Setup - Apache CouchDB

Welcome to the Apache CouchDB Setup Wizard

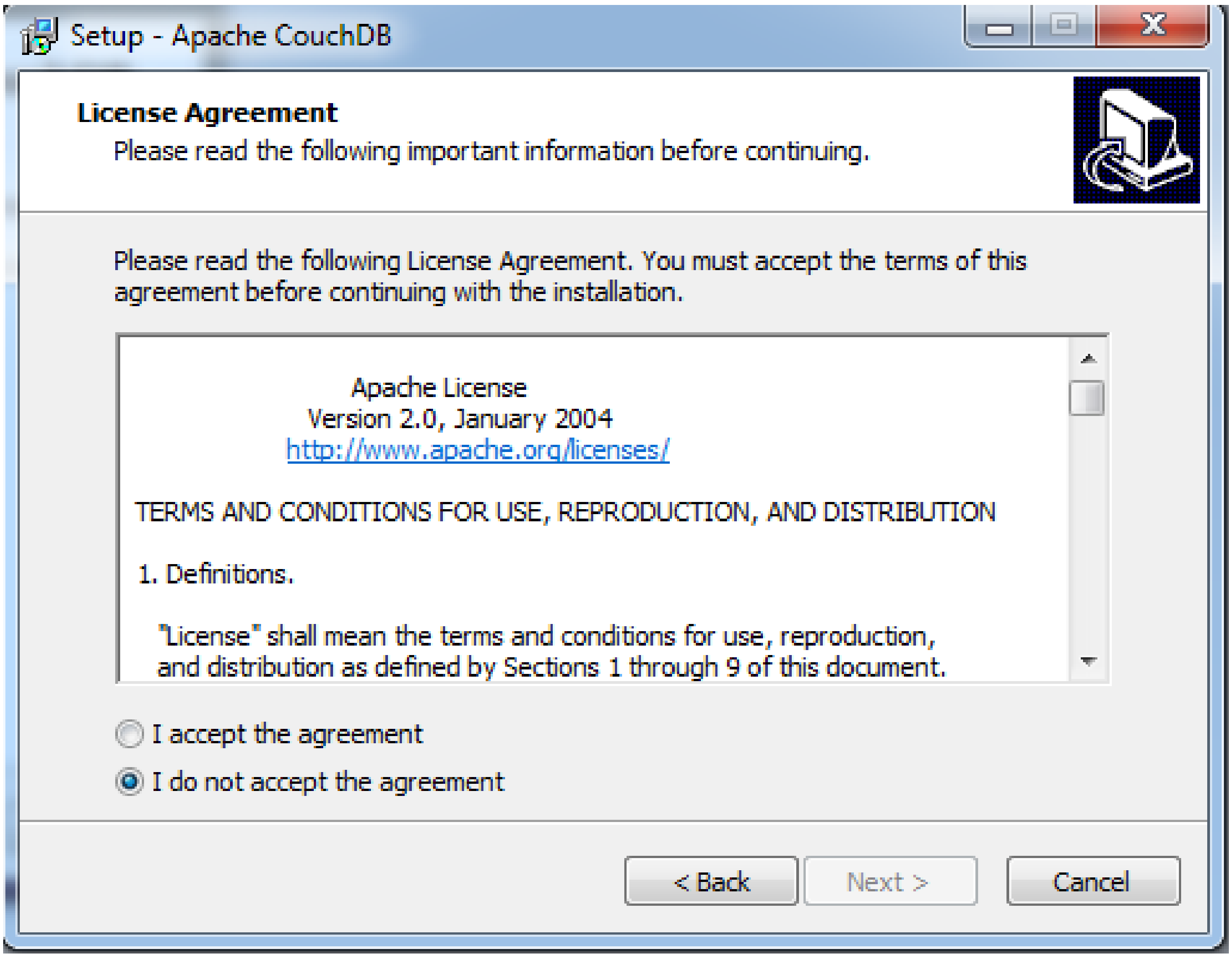
This will install Apache CouchDB 1.5.0 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Figure 1.1: screenshot

- On the “Welcome” screen, Accept the License agreement



License Agreement

Please read the following important information before continuing.

Please read the following License Agreement. You must accept the terms of this agreement before continuing with the installation.

Apache License
Version 2.0, January 2004
http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

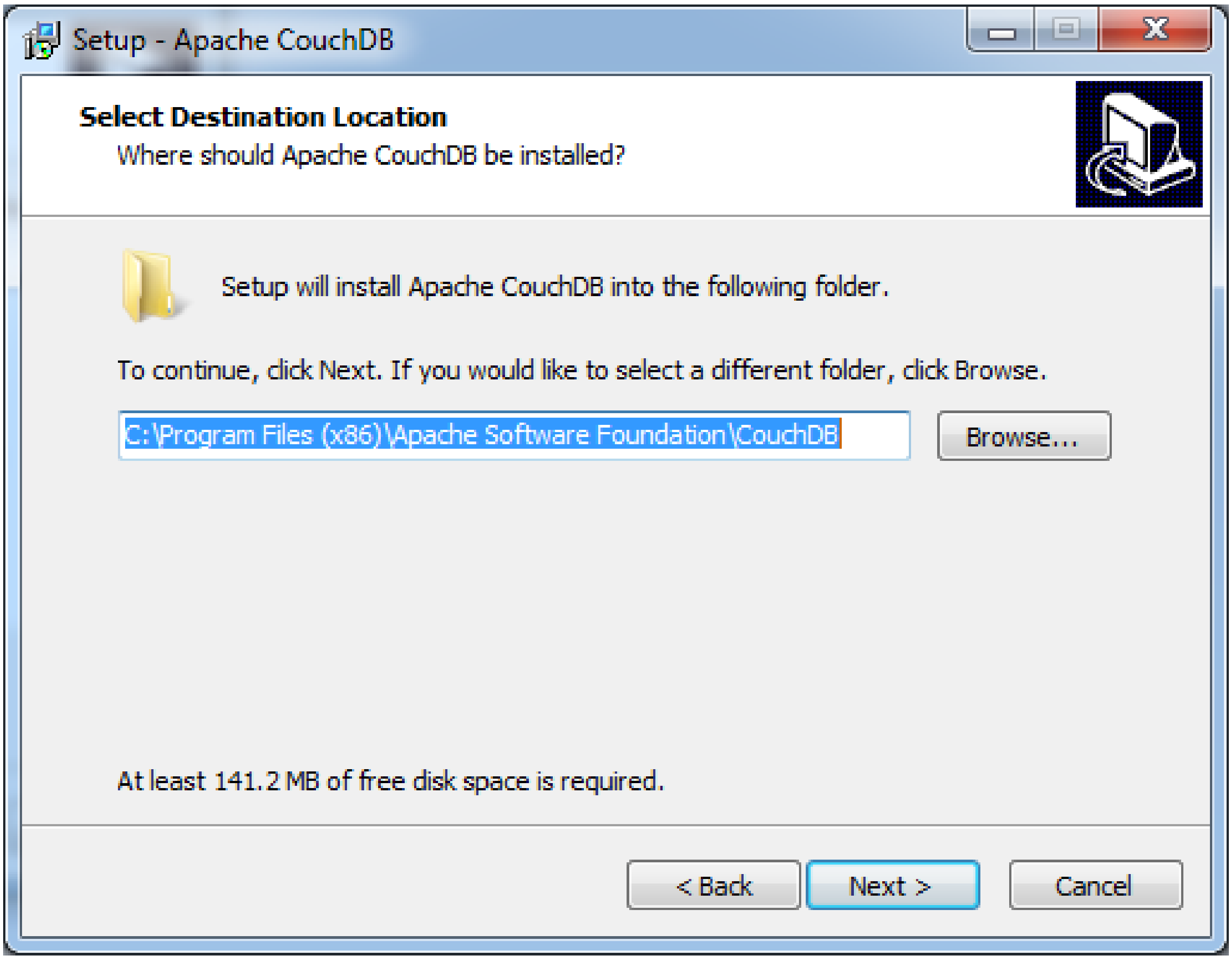
1. Definitions.

"License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

I accept the agreement [✓ I do not accept the agreement]

Figure 1.2: screenshot

- Select the installation directory



Select Destination Location

Where should Apache CouchDB be installed?

Setup will install Apache CouchDB into the following folder.

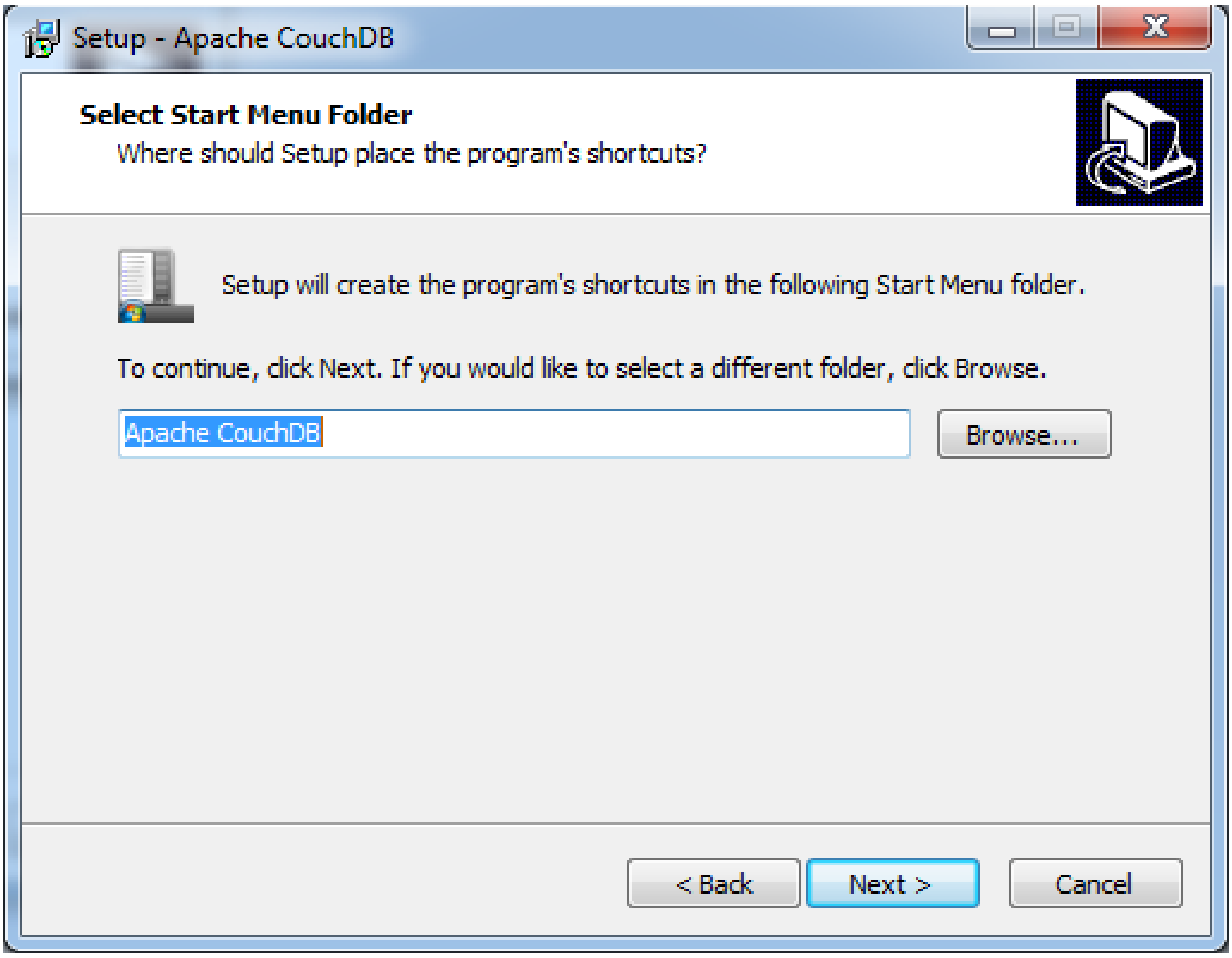
To continue, click Next. If you would like to select a different folder, click Browse.

C:\Program Files (x86)\Apache Software Foundation\CouchDB

At least 141.2 MB of free disk space is required.

Figure 1.3: screenshot

- Specify the "Start Menu" group name


Setup - Apache CouchDB

Select Start Menu Folder

Where should Setup place the program's shortcuts?

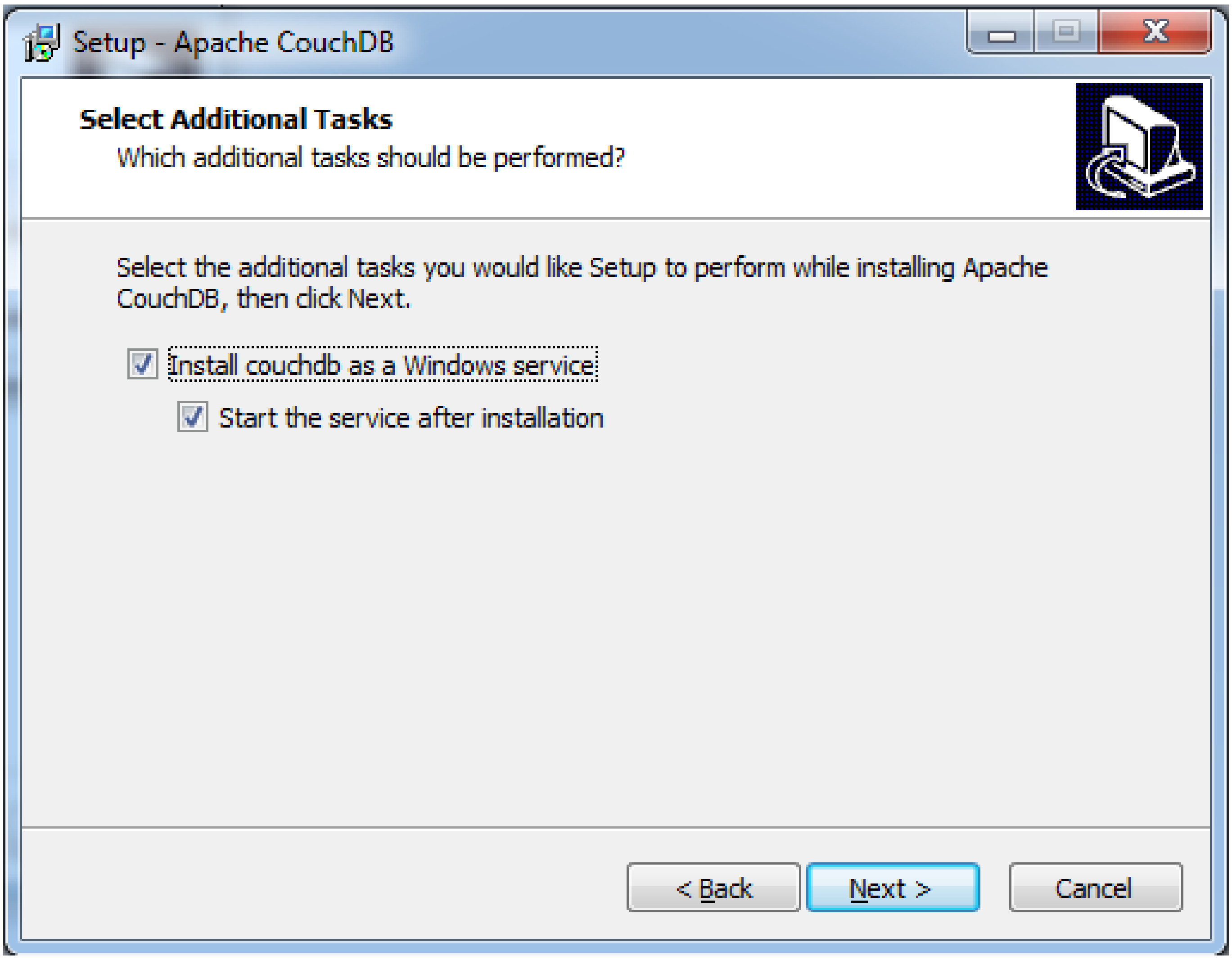
Setup will create the program's shortcuts in the following Start Menu folder.

To continue, click Next. If you would like to select a different folder, click Browse.

Apache CouchDB

Figure 1.4: screenshot

- Approve the installation of CouchDB as service and it will be started automatically after installation


Setup - Apache CouchDB

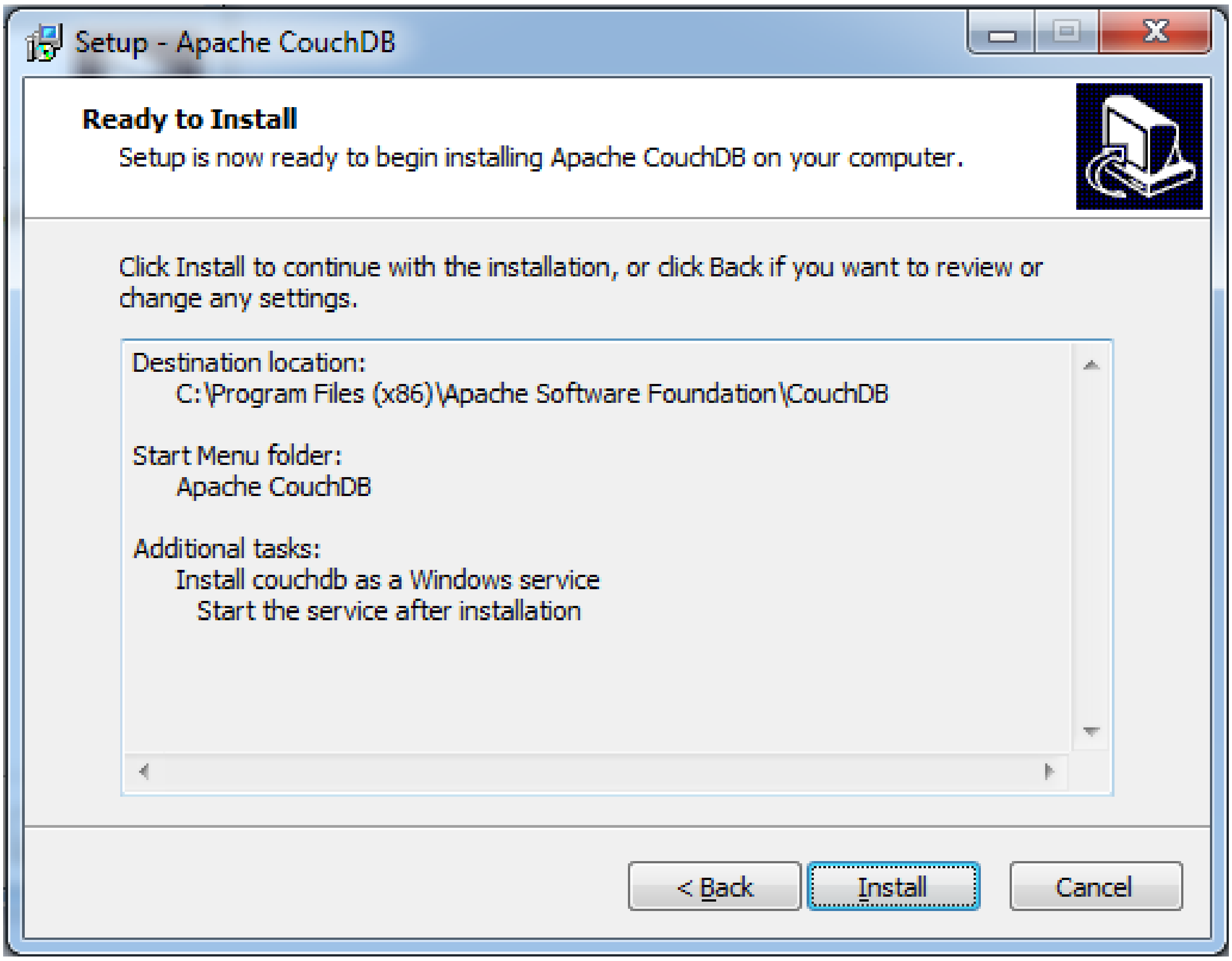
Select Additional Tasks
Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Apache CouchDB, then click Next.

[✓] Install couchdb as a Windows service
[✓] Start the service after installation

Figure 1.5: screenshot

- Verify installation settings



Ready to Install

Setup is now ready to begin installing Apache CouchDB on your computer.

Click Install to continue with the installation, or click Back if you want to review or change any settings.

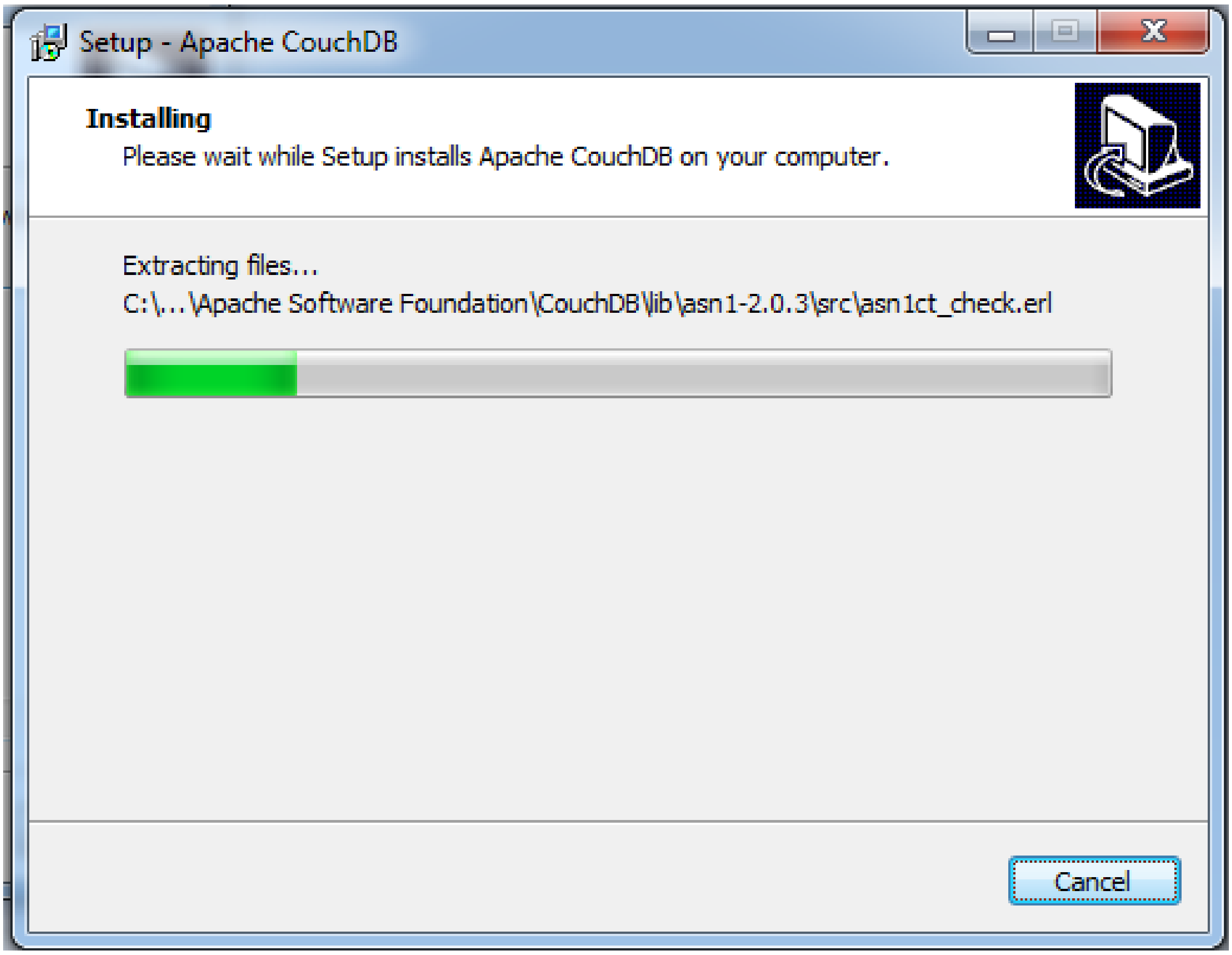
Destination location:
C:\Program Files (x86)\Apache Software Foundation\CouchDB

Start Menu folder:
Apache CouchDB

Additional tasks:
Install couchdb as a Windows service
Start the service after installation

Figure 1.6: screenshot

- Click install for installing CouchDB



Installing

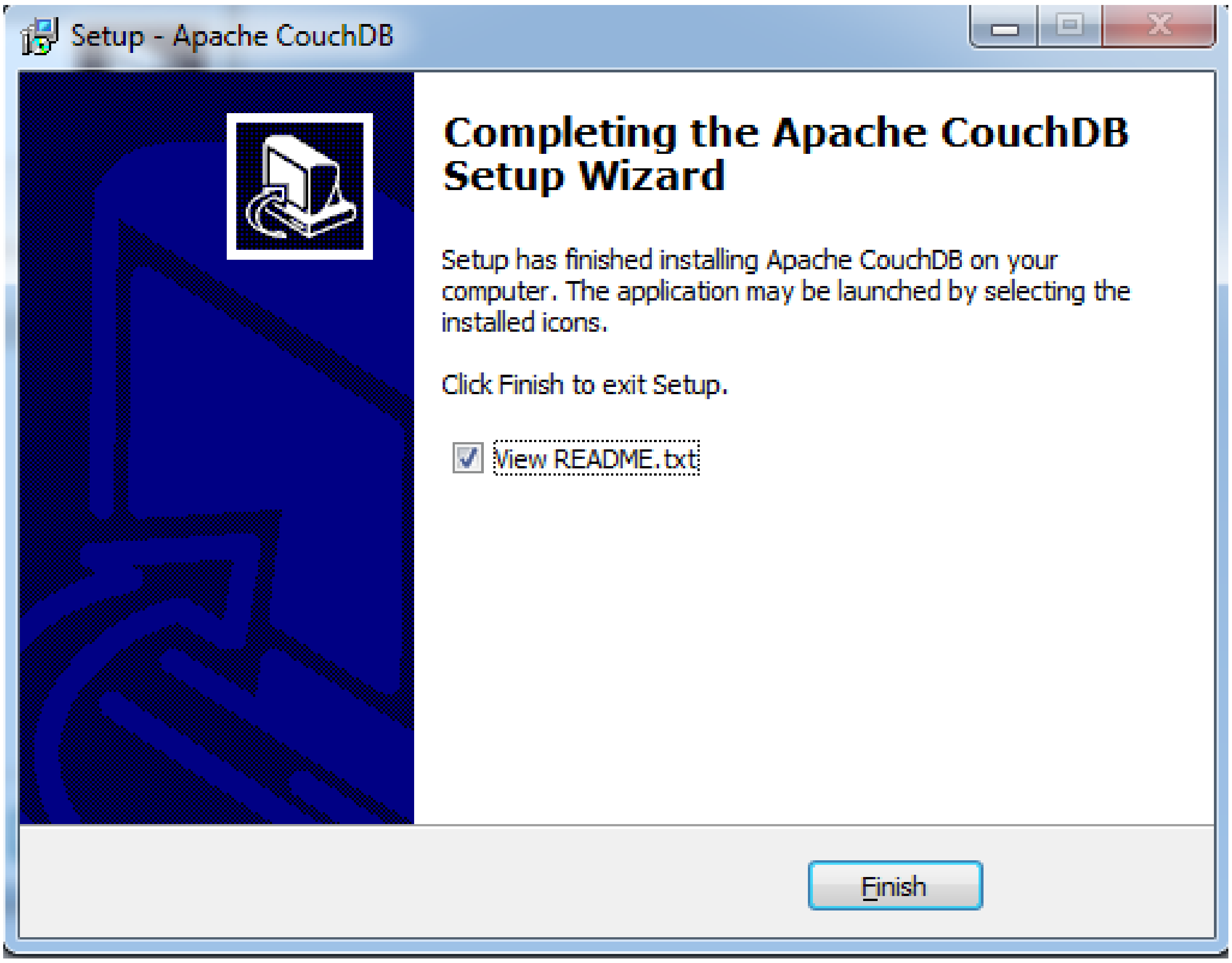
Please wait while Setup installs Apache CouchDB on your computer.

Extracting files...

C:\...\Apache Software Foundation\CouchDB\lib\asn1-2.0.3\src\asn1ct_check.erl

Figure 1.7: screenshot

- After completion of installation, click finish.


Setup - Apache CouchDB

Completing the Apache CouchDB Setup Wizard

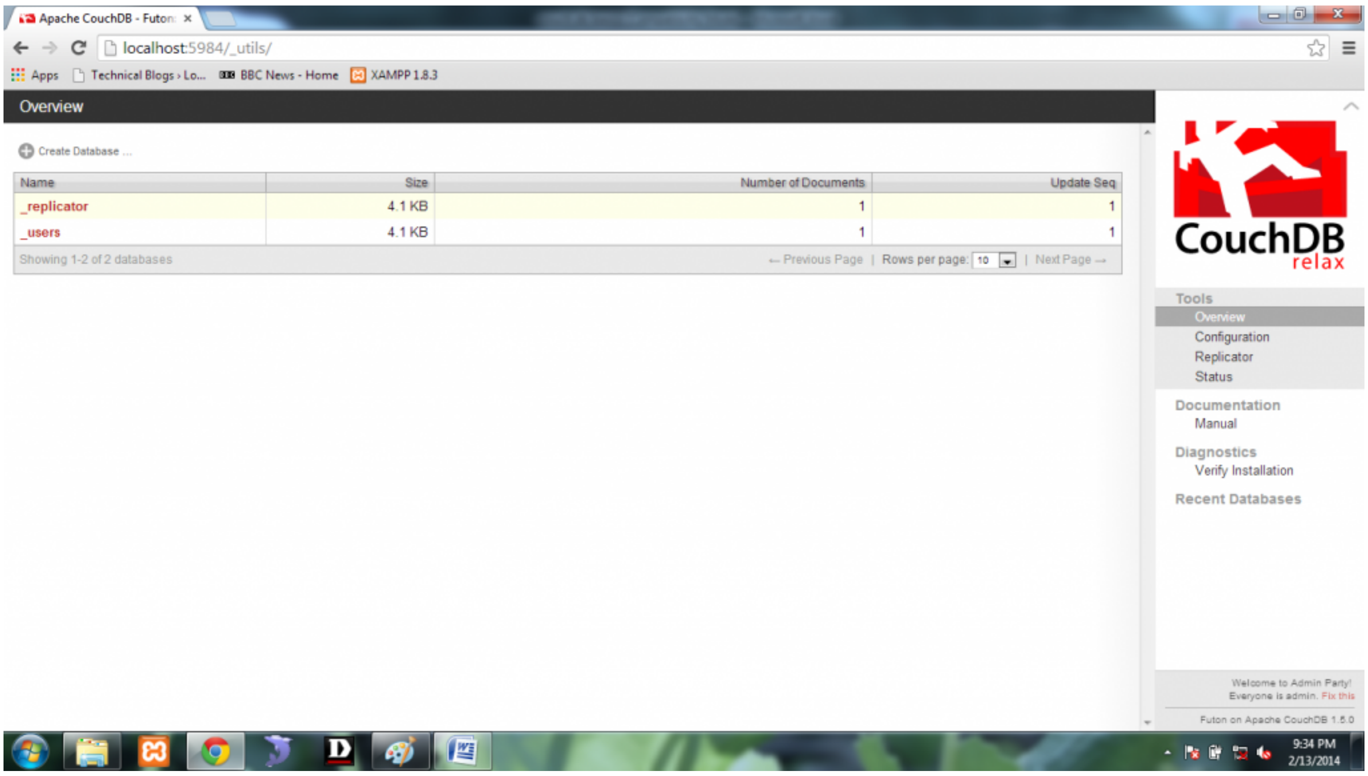
Setup has finished installing Apache CouchDB on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

[✓] View README.txt

Figure 1.8: screenshot

- Open up the Futon Admin UI (if CouchDB is not autostarted after installation, you have to start it first manually)



Name	Size	Number of Documents	Update Seq
_replicator	4.1 KB	1	1
_users	4.1 KB	1	1

Showing 1-2 of 2 databases ← Previous Page | Rows per page: 10 | Next Page →

Figure 1.9: screenshot

1.4 Couch DB Installation on Ubuntu:

Depending on rhe Ubuntu release, CouchDB availability varies. Newer versions of Ubuntu have a recent CouchDB included in their respective software repositories. We can install CouchDB with the Ubuntu Software Center, or from the command line with the apt-get or aptitude utilities. However, to get the newest version of CouchDB we may have to install from source, or other package repositories that have newer pre-built CouchDB packages.

1.4.1 Installing using an existing package

Open a Terminal and type:

```
sudo apt-get install couchdb -y
```

Troubleshooting: If the aptitude/apt-get installation gives an error message then CouchDB might not have access to its pid file in Ubuntu Machine.

In order to resolve this, type in a Terminal:

```
sudo chown -R couchdb /var/run/couchdb
```

Then rerun the setup script:

```
sudo dpkg --configure couchdb
```

1.4.2 Installing from Source on Precise, Quantal, Raring, and Saucy

Download the CouchDB sources from an apache mirror (http://www.apache.org/dyn/closer.cgi?path=couchdb/source/1.4.0/-apache-couchdb-1.4.0.tar.gz).

- make sure you have a couchdb user for the daemon and the couchb group too
- get developer tools dependencies
- sudo apt-get install -y g++
- sudo apt-get install -y erlang-dev erlang-manpages erlang-base-hipe erlang-eunit erlang-nox erlang-xmerl erlang-inets

Chapter 2

Basics and Operations

2.1 Introduction

As we saw in our introductory lesson, Apache CouchDB is an open source NoSQL database that uses JSON to store data, JavaScript as its query language and HTTP for an API. In CouchDB, each database is actually a collection of documents. Each document maintains its own data and a self-contained schema. An application may access multiple databases on different servers and Document metadata contain revision information in order to make merging possible in case the databases get disconnected.

In CouchDB, all operations have a unique URI that gets exposed via HTTP. The REST APIs use the typical HTTP methods POST, GET, PUT and DELETE for the four basic CRUD (Create, Read, Update, Delete) operations on all resources. Finally, Futon is CouchDB's web-based administration console. Let's see more details for those components.

2.2 Futon

Futon will can be accessed by a browser via the following address: http://localhost:5984/_utils/

The main overview page provides a list of the databases and provides the interface for querying the database and creating and updating documents.

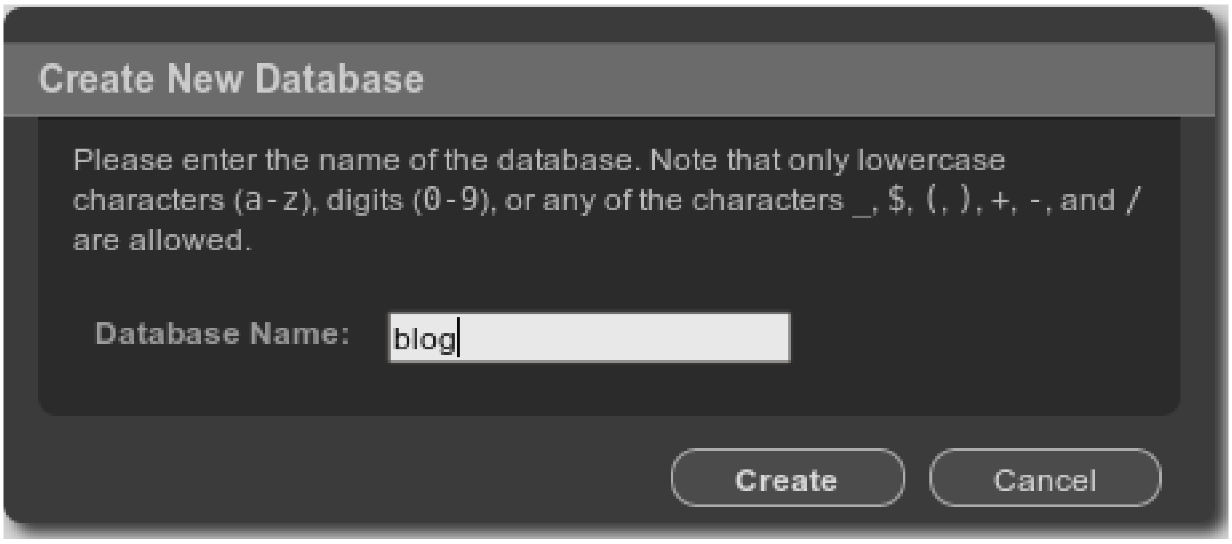
The main sections in Futon are:

- **Configuration:** An interface into the configuration of Wer CouchDB installation. The interface allows us to edit the different configurable parameters.
- **Replicator:** An interface to the replication system, enabling us to initiate replication between local and remote databases.
- **Status:** Displays a list of the running background tasks on the server. Background tasks include view index building, compaction and replication. The Status page is an interface to the Active Tasks API call.
- **Verify Installation:** The Verify Installation allows us to check whether all of the components of CouchDB installation are correctly installed.
- **Test Suite:** The Test Suite section allows us to run the built-in test suite. This executes a number of test routines entirely within browser to test the API and functionality of CouchDB installation.

2.2.1 Managing Databases and Documents

We can manage databases and documents within Futon using the main Overview section of the Futon interface.

To create a new database, click the Create Database Ellipsis button. We will be prompted for the database name, as shown in the figure below.



Create New Database

Please enter the name of the database. Note that only lowercase characters (a-z), digits (0-9), or any of the characters _, $, (,), +, -, and / are allowed.

Database Name: blog

Figure 2.1: Create CouchDB Database

Type the database name (i.e. *blog* here) in the textbox that we want to create.

Once we have created the database (or selected an existing one), we will be shown a list of the current documents. If we create a new document, or select an existing document, we will be presented with the edit document display.

Editing documents within Futon requires selecting the document and then editing (and setting) the fields for the document individually before saving the document back into the database.

For example, the figure below shows the editor for a single document, a newly created document with a single ID, the document _id field (click on the image to show in full size).

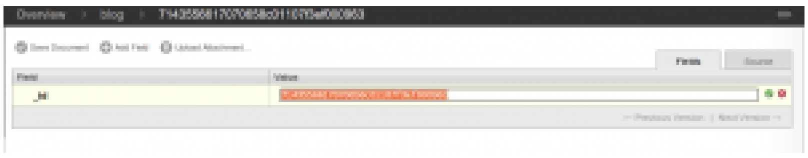
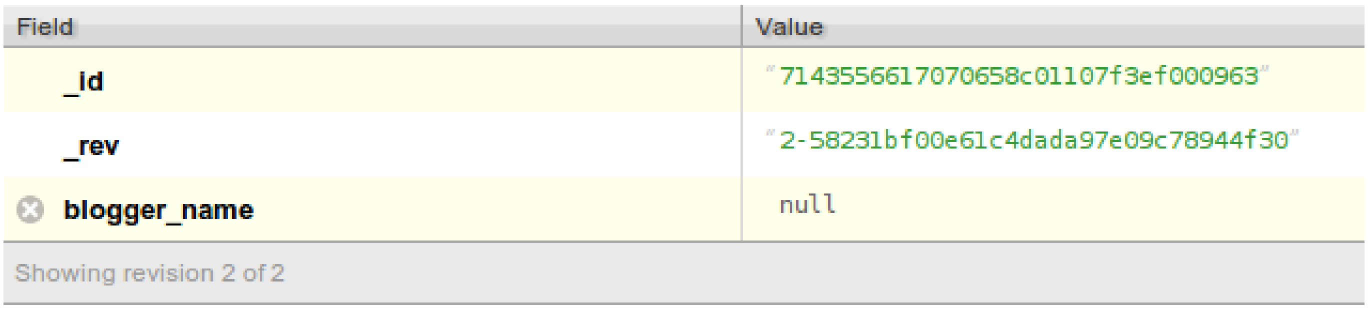


Figure 2.2: New document

To add a field to the document:

- Click Add Field.
- In the fieldname box, enter the name of the field. For example, “blogger_name”.
- Click the green tick next to the field name to confirm the field name change.

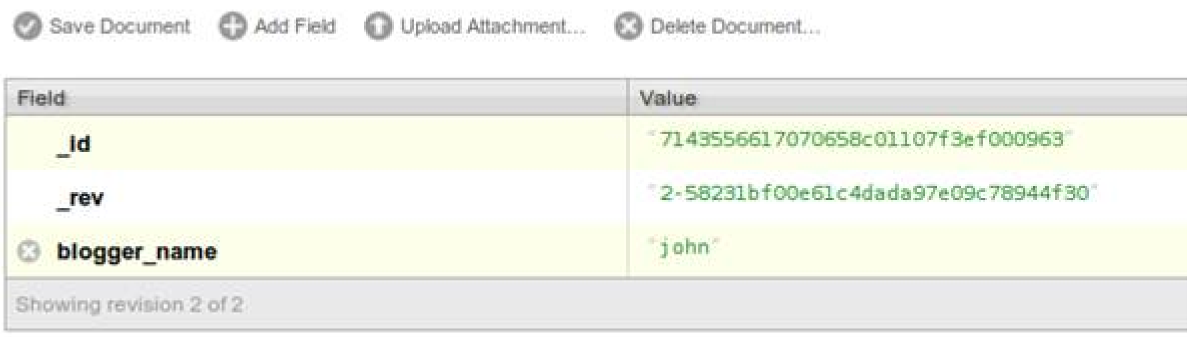


Field	Value
_id	"7143556617070658c01107f3ef000963"
_rev	"2-58231bf00e61c4dada97e09c78944f30"
blogger_name	null
Showing revision 2 of 2	

Figure 2.3: Changing a field name

-Double-click the corresponding Value cell.

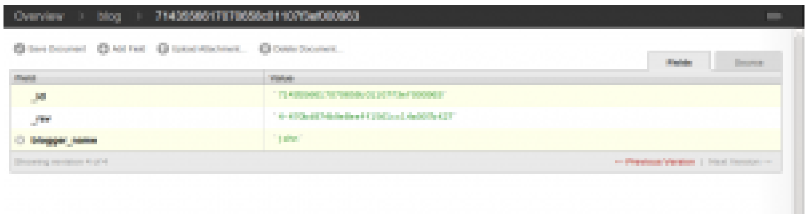
- Enter a company name, for example “john”.



Field	Value
_id	"7143556617070658c01107f3ef000963"
_rev	"2-58231bf00e61c4dada97e09c78944f30"
blogger_name	"john"
Showing revision 2 of 2	

Figure 2.4: Field name changed

- Click the green tick next to the field value to confirm the field value.
- We must explicitly save the document by clicking the Save Document button at the top of the page. This will save the document, and then display the new document with the saved revision information (the _rev field). (Click on image to show in full size)



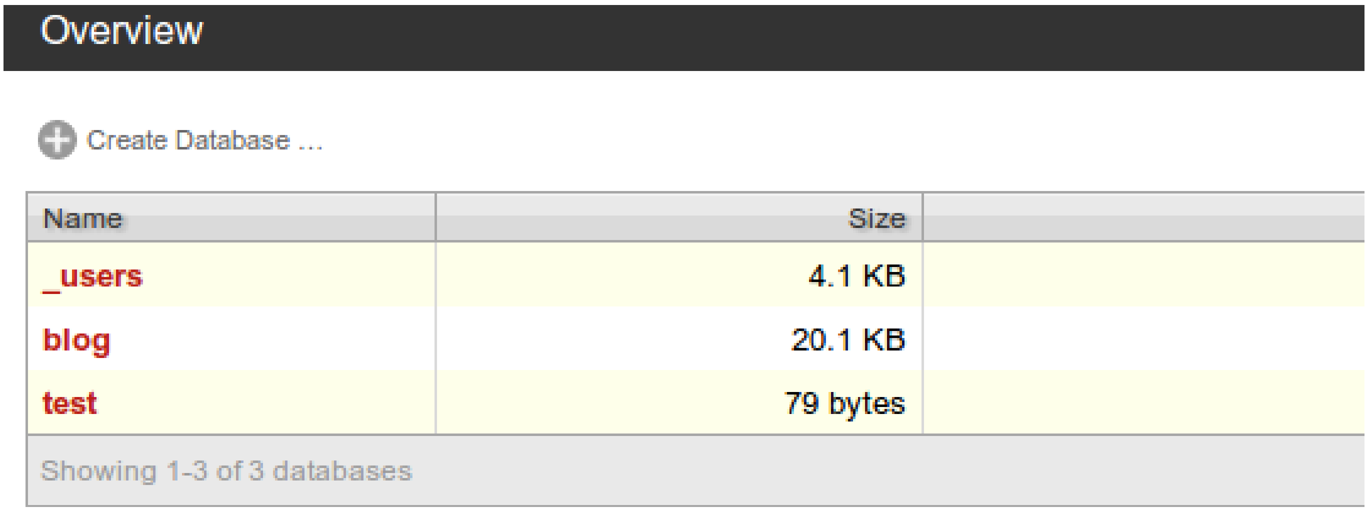
Field	Value
_id	[illegible]
_rev	[illegible]
blogger_name	"john"

Figure 2.5: Document changes

2.2.2 Configuring Replication

When we click the Replicator option within the Tools menu we are presented with the Replicator screen. This allows us to start replication between two databases by filling in or select the appropriate options within the form provided.

For example, let's say we have another database *test*. So the list of databases are the following:



Name	Size
_users	4.1 KB
blog	20.1 KB
test	79 bytes

Showing 1-3 of 3 databases

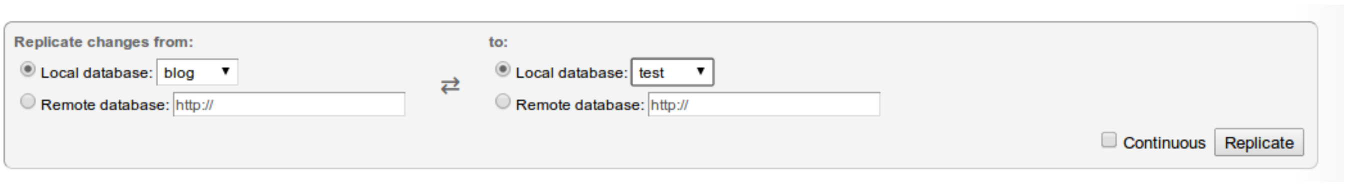
Figure 2.6: Databases Overview

Now we are going to replicate database from *blog* to *test*.

To do that, click on the *Replicator* on the right side panel.

To start the replication process, either select the local database or enter a remote database name into the corresponding areas of the form. The replication occurs from the database on the left to the database on the right.

If we are specifying a remote database name, we must specify the full URL of the remote database (including the host, port number and database name). If the remote instance requires authentication, we can specify the username and password as part of the URL, for example http://username:pass@remotehost:5984/blog. (Click on image to show in full size)


Replicate changes from:
[✓ Local database:] blog
Remote database: http://

to:
[✓ Local database:] test
Remote database: http://

Continuous

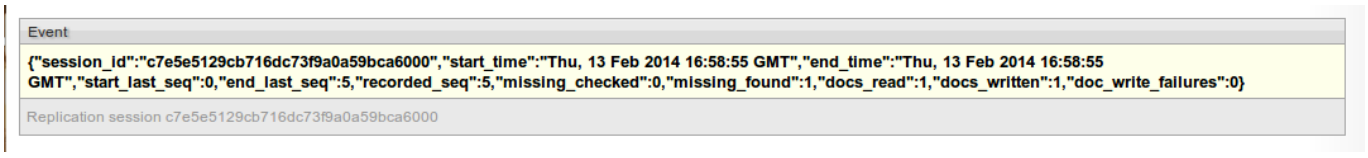
Figure 2.7: Remote CouchDB Replication

To enable continuous replication, click the Continuous checkbox. Click on the *Replicate* button.

The replication process should start and will continue in the background. If the replication process takes a long time, we can monitor the status of the replication using the Status option under the Tools menu.

Once the replication has been completed, the page will show the information using the CouchDB API.

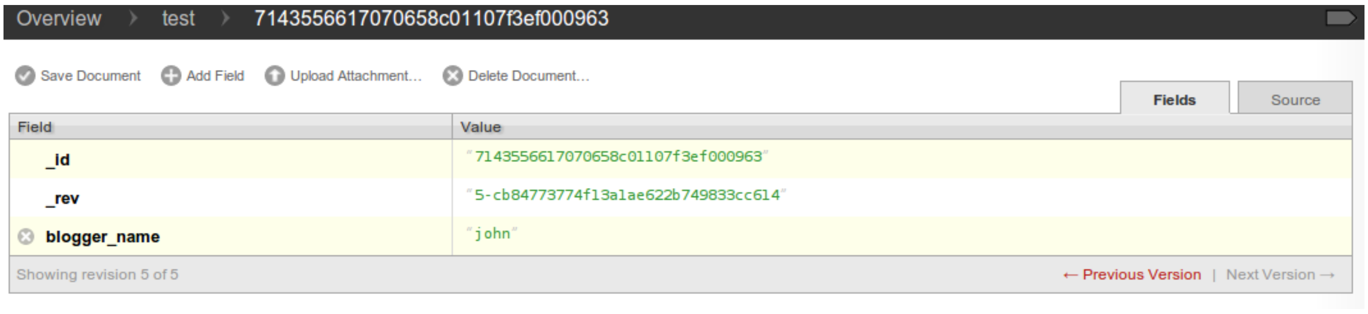
The result will be shown like in the following image (Click on image to show in full size):



Event
{"session_id":"c7e5e5129cb716dc73f9a0a59bca6000","start_time":"Thu, 13 Feb 2014 16:58:55 GMT","end_time":"Thu, 13 Feb 2014 16:58:55 GMT","start_last_seq":0,"end_last_seq":5,"recorded_seq":5,"missing_checked":0,"missing_found":1,"docs_read":1,"docs_written":1,"doc_write_failures":0}
Replication session c7e5e5129cb716dc73f9a0a59bca6000

Figure 2.8: Replication Result

If we now open the *test* database, we will find an exact replica of the *blog* database:



Field	Value
_id	"7143556617070658c01107f3ef000963"
_rev	"5-cb84773774f13a1ae622b749833cc614"
blogger_name	"john"

Showing revision 5 of 5 ← Previous Version | Next Version →

Figure 2.9: Successful Replication

2.3 CRUD Operations

Next, we will have a quick look at CouchDB's bare-bones Application Programming Interface (API) by using the command-line utility "curl". It gives us control over raw HTTP requests and we can see exactly what is going on the database.

Make sure CouchDB is still running and then from the command line execute the following:

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

This issues a GET request to the newly installed CouchDB instance. The reply should look something like:

```
{"couchdb":"Welcome","version":"1.0.1"}
```

Next, we can get a list of the existing databases:

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

Note that we added the "_all_dbs" string to the initial request.. The response should look like:

```
["_users","blog","test"]
```

It is showing our 2 databases named blog and test which were created earlier via the Futon UI.

Let's create another database, using the API this time:

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

Executing this, the CouchDB will reply with:

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

Retrieving the list of databases again shows some useful results:

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

The output shows:

```
["_users","blog","shopcart","test"]
```

Let's create another database with the same database name:

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

CouchDB will reply with:

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

We 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/bookstore
```

CouchDB will reply with:

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

Retrieving the list of databases yet again shows some useful results:

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

CouchDB will respond with:

```
["bookstore","_users","blog","shopcart","test"]
```

To round things off, let's delete the second database:

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

CouchDB will reply with:

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

The list of databases is now the same as it was before:

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

CouchDB will respond with:

```
["_users","blog","shopcart","test"]
```

Everything is done using the standard HTTP methods, GET, PUT, POST, and DELETE with the appropriate URI.

2.3.1 Documents

Documents are CouchDB's central data structure. To better understand and use CouchDB, we need to think in terms of documents. In this chapter we will walk though the lifecycle of designing and saving a document. We'll follow up by reading documents and aggregating and querying them with views.

Documents are self-contained units of data. The 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}*).

How many data items can be put into the documents depends on the application and a bit on how we want to use views. Generally, a document corresponds to an object instance in the programming language.

Documents differ subtly from garden-variety objects in that they usually have authors and CRUD operations (create, read, update, delete). Document-based software (like the word processors and spreadsheets) build their storage model around saving documents so that authors get back what they created.

Validation functions are available so that we don't have to worry about bad data causing errors in our system. Often in document-based software, the client application edits and manipulates the data, saving it back.

Let's suppose a user can comment on the item ("lovely book"); we 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.

Instead of storing a list on the item document, in this case it may be acutally better to model comments into a collection of documents. There are patterns for accessing collections, which CouchDB makes easy. We likely want to show only 10 or 20 at a time and provide previous and next links. By handling comments as individual entities, we can group them with views. A group could be the entire collection or slices of 10 or 20, sorted by the item they apply to so that it's easy to grab the set we need.

Everything that will be handled separately in the application should be broken up into documents. Items are single, and comments are single, but we don't need to break them into smaller pieces. Views provide a convenient way to group our documents in meaningful ways.

2.4 Common HTTP operations

We start out by revisiting the basic operations we ran in the last chapter, looking behind the scenes. We will also discover what Futon runs in the background in order to give us the nice features we saw earlier.

While 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 why CouchDB works the way it does.

The API can be subdivided into the following sections. We'll explore them individually:

- Server
- Databases
- Documents
- Replication
- Server

This 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/
```

CouchDB replies, all excited to get going:

```
{"couchdb":"Welcome","version":"1.0.1"}
```

We get back a JSON string, which, if parsed into a native object or data structure of our programming language, gives us access to the welcome string and version information.

This is not terribly useful, but it illustrates nicely the way CouchDB behaves. We send an HTTP request and we receive a JSON string in the HTTP response as a result.

2.4.1 Databases

Strictly speaking, 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 in detail. 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, so don’t get confused by it. In general, it should be clear from the context if we are talking about the whole of CouchDB or a single database within CouchDB.

Now let’s make one! 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/student
```

CouchDB replies:

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

That’s it. We created a database and CouchDB told us that all went well. What happens if we 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/student
```

CouchDB replies:

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

We get back an error. This is pretty convenient. CouchDB stores each database in a single file.

Let’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/student-backup
```

Curl elaborates:

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

*   Trying 127.0.0.1... connected

> PUT /student-backup HTTP/1.1

> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4  ↩
   libidn/1.23 librtmp/2.3

> Host: 127.0.0.1:5984

> Accept: */*

>

< HTTP/1.1 201 Created

< Server: CouchDB/1.0.1 (Erlang OTP/R14B)

< Location: http://127.0.0.1:5984/student-backup

< Date: Sat, 15 Feb 2014 17:50:51 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
```

Let's step through this line by line in order to understand what's going on and find out what's important. Once we've seen this output a few times, we'll be able to spot the important bits more easily.

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

This is curl telling us that it is going to establish a TCP connection to the CouchDB server we specified in our request URI. Not at all important, except when debugging networking issues.

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

Curl tells us it successfully connected to CouchDB. Again, not important if there is no problem with the network. The 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 /student-backup HTTP/1.1
```

This initiates an HTTP request. Its method is PUT, the URI is /student-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 We should be using HTTP/1.1.

Next, we see a number of request headers. These are used to provide additional details about the request to CouchDB.

```
> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4  ↩
    libidn/1.23 librtmp/2.3
```

The User-Agent header tells CouchDB which piece of client software is doing the HTTP request. It's the curl program. 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, the User-Agent header is not very relevant.

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

The Host header is required by HTTP 1.1. It tells the server the hostname that came with the request.

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

The Accept header tells CouchDB that curl accepts any media type. We'll look into why this is useful a little later.

```
>
```

An empty line denotes that the request headers are now finished 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
```

The 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 behavior for response codes. CouchDB fully follows the RFC. The 201 Created status code tells the client that the resource the request was made against was successfully created. No surprise here, but if we remember that we got an error message when we tried to create this database twice, we now know that this response could include a different response code.

Acting upon responses based on response codes is a common practice. For example, all response codes of 400 (or greater than that) inform us that some error occurred. If we want to shortcut the logic and immediately deal with the error, we could just check a >= 400 response code.

```
< Server: CouchDB/0.10.1 (Erlang OTP/R13B)
```

The Server header is good for diagnostics. It tells us which CouchDB version and which underlying Erlang version we are talking to. In general, we can ignore this header, but it is good to know it's there if We need it.

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

The Date header tells the time of the server. Since client and server time are not necessarily synchronized, this header is purely informational. We shouldn't build any critical application logic on top of this!

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

The Content-Type header tells which MIME type the HTTP response body uses and what encoding is used for it. We already know that 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 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 browsers will display the JSON as text.

There are some extensions that make our browser JSON-aware, but they are not installed by default. For more information, look at the popular JSONView extension, available for both Firefox and Chrome.

If we send Accept: application/json in our request, CouchDB knows that we can deal with a pure JSON response with the proper Content-Type header and will use it instead of text/plain.

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

The Content-Length header simply tells us how many bytes the response body has.

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

This Cache-Control header tells us, or any proxy server between CouchDB and us, not to cache this response.

```
<
```

This empty line tells us we're done with the response headers and what follows now is the response body.

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

The 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.

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

This deletes a CouchDB database. The request will remove the file that the database contents are stored in. We need to use this command with care, as our data will be deleted without a chance to bring it back easily if we don't have a backup copy.

The console will look like this:

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

*   Trying 127.0.0.1... connected

> DELETE /student-backup HTTP/1.1

> User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4  ↩
   libidn/1.23 librtmp/2.3

> Host: 127.0.0.1:5984

> Accept: */*
```

```
>

< HTTP/1.1 200 OK

< Server: CouchDB/1.0.1 (Erlang OTP/R14B)

< Date: Sat, 15 Feb 2014 17:53:58 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
```

This section went knee-deep into HTTP and set the stage for discussing the rest of the core CouchDB API. Next stop: documents.

2.4.2 Documents

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.

We'll add some more of our favorite music albums. Get a fresh UUID from the /_uuids resource. If we don't remember how that works, w 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}'
```

Now with the -v option, CouchDB's reply (with only the important bits shown) 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"}
```

We're getting back the 201 Created HTTP status code in the response headers, as we saw 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. 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. 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.

2.4.3 Attachments

CouchDB 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 consist of any type of data. It is easier 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.

Attachments get their own URL where we can upload data. Let's suppose we want to add the album artwork to the 6e1295ed6c29495e54c
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"
```

The --data-binary @ 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 JPEG 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 will render the image instead of offering the data for download. This will come in handy later. Note that we need to provide the current revision number of the document we're attaching the artwork to, just as if we would update the document.

If we request the document again, we will see a new member:

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

CouchDB 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}}}
```

_attachments is a list of keys and values where the values are JSON objects containing the attachment metadata. stub=true tells us that this entry is just the metadata. If we use the ?attachments=true HTTP option when requesting this document, we'd get a Base64-encoded string containing the attachment data.

We'll have a look at more document request options later as we explore more features of CouchDB, such as replication, which is the next topic.

2.4.4 Replication

CouchDB replication is a mechanism to synchronize databases. Much like rsync synchronizes two directories locally or over a network, replication synchronizes two databases locally or remotely.

Using a simple POST request, we tell CouchDB the source and the target of a replication and CouchDB will figure out which documents and new document revisions exist on the source DB and they are not yet on the target DB, and will proceed to move the missing documents and revisions over.

First, we'll create a target database. Note that CouchDB won't automatically create a target database and will return a replication failure if the target doesn't exist:

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

Now 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"}' -H "Content-Type: application/json"
```

CouchDB replies (this time we formatted the output so We can 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
}
```

CouchDB 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 we have a lot of documents, it'll take a while until they are all replicated and we won't get back the replication response until all documents are replicated. It is important to note that replication replicates the database only as it was at the point in time when replication was started. So, any additions, modifications, or deletions subsequent to the start of replication will not be replicated.

We'll punt on the details again: the "ok": true at the end tells us all went well. If we have a look at the albums-replica database, we should see all the documents that we created in the albums database.

In CouchDB terms, we created a local copy of a database. This is useful for backups or to keep snapshots of a specific state of data around for later.

There 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). We can also specify a remote database as the target:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"albums","target":"http:// ↩
    example.org:5984/albums-replica"}' -H "Content-Type: application/json"
```

Using a local source and a remote target database is called push replication. We're pushing changes to a remote server.

We 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://example.org:5984/ ↩
    albums-replica","target":"albums"}' -H "Content-Type: application/json"
```

Finally, we can run remote replication, which is mostly useful for management operations:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"http://example.org:5984/ ↩
    albums","target":"http://example.org:5984/albums-replica"}' -H "Content-Type:  ↩
    application/json"
```

2.5 JSON

CouchDB uses JavaScript Object Notation (JSON) for data storage. JSON is a lightweight format based on a subset of JavaScript 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 really comes in handy when we're building dynamic web applications and we want to fetch some data from the server.

Here'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."
}
```

We can see that the general structure is based around key/value pairs and lists of things.

2.5.1 Data Types

JSON has a number of basic data types We can use. We'll cover them all here.

2.5.2 Numbers

We can have positive integers: "Count": 253

Or negative integers: "Score": -19

Or floating-point numbers: "Area": 456.31

Or scientific notation: "Density": 5.6e+24

2.5.3 Strings

We can use strings for values:

```
"Author": "Rusty"
```

We have to escape some special characters, like tabs or newlines:

```
"poem": "May I compare thee to somentsalty plankton."
```

The JSON site has details on what needs to be escaped.

2.5.4 Booleans

We can have boolean true values:

```
"Draft": true
```

Or boolean false values:

```
"Draft": false
```

Arrays

An array is a list of values:

```
"Tags": ["plankton", "baseball", "decisions"]
```

An array can contain any other data type, including arrays:

```
"Context": ["dog", [1, true], {"Location": "puddle"}]
```

Objects

An object is a list of key/value pairs:

```
{"Subject": "I like Plankton", "Author": "Rusty"}
```

Nulls

We can have null values:

```
"Surname": null
```

2.6 Documents

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

Each document in CouchDB has an ID. This ID is unique per database. We are free to choose any string to be the ID, but for best results we recommend a UUID (or GUID), i.e., a Universally (or Globally) Unique IDentifier. UUIDs are random numbers that have such a low collision probability. We can make thousands of UUIDs per 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 we care what somebody else is doing? For one, somebody else could be at a later time or on a different computer; secondly, CouchDB replication let's us share documents with others and using UUIDs ensures that it all works. But more on that later; let's make some documents:

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

CouchDB replies:

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

The curl command appears complex, but let's break it down. First, -X PUT tells curl to make a PUT request. It is followed by the URL that specifies Wer 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 Wer document's ID. Finally, the -d flag tells curl to use the following string as the body for the PUT request. The string is a simple JSON structure including title and artist attributes with their respective values.

A CouchDB document is simply a JSON object. We can use any JSON structure with nesting. We can fetch the document's revision information by adding ?revs_info=true to the get request.

To get a UUID, we use:

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

CouchDb will reply us back, like this:

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

Here are two simple examples of documents:

```
{
 "_id":"discussion_tables",
 "_rev":"D1C946B7",
 "Sunrise":true,
 "Sunset":false,
 "FullHours":[1,2,3,4,5,6,7,8,9,10],
 "Activities": [
   {"Name":"Football", "Duration":2, "DurationUnit":"Hours"},
   {"Name":"Breakfast", "Duration":40, "DurationUnit":"Minutes", "Attendees":["Jan", " ↩
       Damien", "Laura", "Gwendolyn", "Roseanna"]}
 ]
}
```

```
{
 "_id":"some_doc_id",
 "_rev":"D1C946B7",
 "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."
}
```

2.6.1 Special Fields

Note that any top-level fields within a JSON document containing a name that starts with a _ prefix are reserved for use by CouchDB itself. Also see Reserved_words. Currently (0.10+) reserved fields are:

Table 2.1: datasheet

Field Name	Description
_id	The unique identifier of the document (**mandatory** and **immutable**)
_rev	The current MVCC-token/revision of this document (**mandatory** and **immutable**)
_attachments	If the document has attachments, _attachments holds a (meta-)data structure (see section on HTTP_Document_API#Attachments)
_deleted	Indicates that this document has been deleted and previous revisions will be removed on next compaction run
_revisions	Revision history of the document
_revs_info	A list of revisions of the document, and their availability
_conflicts	Information about conflicts
_deleted_conflicts	Information about conflicts
_local_seq	Sequence number of the revision in the database (as found in the _changes feed)

To request a special field to be returned along with the normal fields we get when we request a document, add the desired field as a query parameter without the leading underscore in a GET request:

```
curl -X GET 'http://localhost:5984/my_database/my_document?conflicts=true'
```

This request will return a document that includes the special field *_conflicts* which contains all the conflicting revisions of "my_document".

Document IDs don't have restrictions on what characters can be used. Although it should work, it is recommended to use non-special characters for document IDs. By using special characters, we have to be aware of proper URL en-/decoding. Documents prefixed with _ are special documents:

Table 2.2: datasheet

Document ID prefix	Description
_design/	are DesignDocuments
_local/	are not being replicated (local documents) and used for Replication checkpointing.

We can have / as part of the document ID but if We refer to a document in a URL We must always encode it as %2F. One special case is _design/ documents, those accept either / or %2F for the / after _design, although / is preferred and %2F is still needed for the rest of the DocID.

2.6.2 Working With Documents Over HTTP

GET

To retrieve a document, simply perform a GET operation at the document's URL:

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

Here is the server's response:

```
{"_id":"6e1295ed6c29495e54cc05947f18c8af","_rev":"1-4b39c2971c9ad54cb37e08fa02fec636"," ↩
    title":"There is Nothing Left to Lose","artist":"Foo Fighters"}
```

2.7 Revisions

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

If we want to update or delete a document, CouchDB expects us to include the _rev field of the revision we 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 without us knowing before we got to request the document update, CouchDB will not accept our update because we are likely to overwrite data we didn't know that even 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/shopcart/6e1295ed6c29495e54cc05947f18c8af -d '{"title":" ↩
    There is Nothing Left to Lose","artist":"Foo Fighters","year":"1997"}'
```

CouchDB replies:

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

If we see this, add the latest revision number of your document to the JSON structure:**

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

Now we 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"}
```

2.7.1 Accessing Previous Revisions

The above example gets the current revision. We may be able to get a specific revision by using the following syntax:

```
GET /somedatabase/some_doc_id?rev=946B7D1C HTTP/1.0
```

To find out what revisions are available for a document, we can do:

```
GET /somedatabase/some_doc_id?revs_info=true HTTP/1.0
```

This returns the current revision of the document, but with an additional _revs_info field, whose value is an array of objects, one per revision. For example:

```
{
  "_revs_info": [
    {"rev": "3-ffffff", "status": "available"},
    {"rev": "2-eeeeee", "status": "missing"},
    {"rev": "1-dddddd", "status": "deleted"},
  ]
}
```

Here, available means the revision content is stored in the database and can still be retrieved. The other values indicate that the content of that revision is not available.

Alternatively, the _revisions field, used by the replicator, can return an array of revision IDs more efficiently. The numeric prefixes are removed, with a "start" value indicating the prefix for the first (most recent) ID:

```
{
  "_revisions": {
    "start": 3,
    "ids": ["fffff", "eeeee", "ddddd"]
  }
}
```

We can fetch the bodies of multiple revisions at once using the parameter open_revs=["rev1","rev2",...], or We can fetch all leaf revisions using open_revs=all. The JSON returns an array of objects with an "ok" key pointing to the document, or a "missing" key pointing to the rev string.

```
[
{"missing":"1-fbd8a6da4d669ae4b909fcdb42bb2bfd"},
{"ok":{"_id":"test","_rev":"2-5bc3c6319edf62d4c624277fdd0ae191","hello":"foo"}}
]
```

2.7.2 HEAD

A HEAD request returns basic information about the document, including its current revision.

```
HEAD /somedatabase/some_doc_id HTTP/1.0
HTTP/1.1 200 OK
Etag: "946B7D1C"
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Content-Length: 256
```

2.7.3 PUT

To create new document we can either use a POST operation or a PUT operation. To create/update a named document using the PUT operation, the URL must point to the document's location.

The following is an example HTTP PUT. It will cause the CouchDB server to generate a new revision ID and save the document with it.

```
PUT /somedatabase/some_doc_id HTTP/1.0
Content-Length: 245
Content-Type: application/json

{
  "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."
}
```

Here is the server's response.

```
HTTP/1.1 201 Created
Etag: "946B7D1C"
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
```

```
Connection: close

{"ok": true, "id": "some_doc_id", "rev": "946B7D1C"}
```

To update an existing document, we also issue a PUTrequest. In this case, the JSON body must contain a _rev property, which lets CouchDB know which revision the edits are based on. If the revision of the document currently stored in the database doesn't match, then a 409 conflict error is returned.

If the revision number does match what's in the database, a new revision number is generated and returned to the client.

For example:

```
PUT /somedatabase/some_doc_id HTTP/1.0
Content-Length: 245
Content-Type: application/json

{
  "_id":"some_doc_id",
  "_rev":"946B7D1C",
  "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."
}
```

Here is the server's response if what is stored in the database is a revision 946B7D1C of document some_doc_id.

```
HTTP/1.1 201 Created
Etag: "2774761002"
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{"ok":true, "id":"some_doc_id", "rev":"2774761002"}
```

And here is the server's response if there is an update conflict (what is currently stored in the database is not revision 946B7D1C of document some_doc_id).

```
HTTP/1.1 409 Conflict
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Length: 33
Connection: close

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

There is a query option batch=okwhich can be used to achieve higher throughput at the cost of lower guarantees. When a PUT(or a document POSTas described below) is sent using this option, it is not immediately written to disk. Instead it is stored in memory on a per-user basis for a second or so (or the number of docs in memory reaches a certain point). After the threshold has passed, the docs are committed to disk. Instead of waiting for the doc to be written to disk before responding, CouchDB sends an HTTP 202 Accepted response immediately.

batch=ok is not suitable for crucial data, but it ideal for applications like logging which can accept the risk that a small proportion of updates could be lost due to a crash. Docs in the batch can also be flushed manually using the _ensure_full_commit API.

2.7.4 POST

The POSToperation can be used to create a new document with a server generated DocID. To do so, the URL must point to the database's location. To create a named document, use the PUTmethod instead.

It is recommended that we avoid POSTwhen possible, because proxies and other network intermediaries will occasionally resend POSTrequests, which can result in duplicate document creation. If our client software is not capable of guaranteeing uniqueness of generated UUIDs, use a GETto /_uuids?count=100to retrieve a list of document IDs for future PUT requests. Please note that the /_uuids-call does not check for existing document ids; collision-detection happens when We are trying to save a document.

The following is an example HTTP POST. It will cause the CouchDB server to generate a new DocID and revision ID and save the document with it.

```
POST /somedatabase/ HTTP/1.0
Content-Length: 245
Content-Type: application/json

{
  "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."
}
```

Here is the server’s response:

```
HTTP/1.1 201 Created
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{"ok":true, "id":"123BAC", "rev":"946B7D1C"}
```

As of 0.11 CouchDB supports handling of multipart/form-data encoded updates. This is used by Futon and not considered a public API. All such requests must contain a valid Referer header.

2.7.5 DELETE

To delete a document, perform a DELETE operation at the document’s location, passing the rev parameter with the document’s current revision. If successful, it will return the revision id for the deletion stub.

```
DELETE /somedatabase/some_doc?rev=1582603387 HTTP/1.0
```

As an alternative we can submit the rev parameter with the etag header field If-Match.

```
DELETE /somedatabase/some_doc HTTP/1.0
If-Match: "1582603387"
```

And the response:

```
HTTP/1.1 200 OK
Etag: "2839830636"
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{"ok":true,"rev":"2839830636"}
```

Note: Deleted documents remain in the database forever, even after compaction, to allow eventual consistency when replicating. If we delete using the DELETE method above, only the _id, _rev and a deleted flag are preserved. If we deleted a document by adding "_deleted":true then all the fields of the document are preserved. This is to allow, for example, to record the time we deleted a document, or the reason we deleted it.

2.7.6 COPY

Note that this is a non-standard extension to HTTP.

We can copy documents by sending an HTTP COPY request. This allows us to duplicate the contents (and attachments) of a document to a new document under a different document id without first retrieving it from CouchDB. Use the Destination header to specify the document that We want to copy to (the target document).

It is not possible to copy documents between databases and it is not (yet) possible to perform bulk copy operations.

```
COPY /somedatabase/some_doc HTTP/1.1
Destination: some_other_doc
```

If we want to overwrite an existing document, we need to specify the target document's revision with a rev parameter in the Destination header:

```
COPY /somedatabase/some_doc HTTP/1.1
Destination: some_other_doc?rev=rev_id
```

The response in both cases includes the target document's revision:

```
HTTP/1.1 201 Created
Server: CouchDB/0.9.0a730122-incubating (Erlang OTP/R12B)
Etag: "355068078"
Date: Mon, 05 Jan 2009 11:12:49 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 41
Cache-Control: must-revalidate

{"ok":true,"id":"some_other_doc","rev":"355068078"}
```

2.7.7 All Documents

2.7.7.1 all_docs

To get a listing of all documents in a database, use the special _all_docs URI. This is a specialized View so the Querying Options of the HTTP_view_API apply here.

```
GET /somedatabase/_all_docs HTTP/1.0
```

This will return a listing of all documents and their revision IDs, ordered by DocID (case sensitive):

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
  "total_rows": 3, "offset": 0, "rows": [
    {"id": "doc1", "key": "doc1", "value": {"rev": "4324BB"}},
    {"id": "doc2", "key": "doc2", "value": {"rev":"2441HF"}},
    {"id": "doc3", "key": "doc3", "value": {"rev":"74EC24"}}
  ]
}
```

Use the query argument descending=true to reverse the order of the output table:

Will return the same as before but in reverse order:

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
  "total_rows": 3, "offset": 0, "rows": [
    {"id": "doc3", "key": "doc3", "value": {"rev":"74EC24"}},
    {"id": "doc2", "key": "doc2", "value": {"rev":"2441HF"}},
    {"id": "doc1", "key": "doc1", "value": {"rev": "4324BB"}}
  ]
}
```

The query string parameters startkey, endkey and limit may also be used to limit the result set. For example:

```
GET /somedatabase/_all_docs?startkey="doc2"&amp;limit=2 HTTP/1.0
```

Will return:

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
  "total_rows": 3, "offset": 1, "rows": [
    {"id": "doc2", "key": "doc2", "value": {"rev":"2441HF"}},
    {"id": "doc3", "key": "doc3", "value": {"rev":"74EC24"}}
  ]
}
```

Use endkey if you are interested in a specific range of documents:

```
GET /somedatabase/_all_docs?startkey="doc2"&amp;endkey="doc3" HTTP/1.0
```

This will get keys inbetween and including doc2 and doc3; e.g. doc2-b and doc234.

Both approaches can be combined with descending:

```
GET /somedatabase/_all_docs?startkey="doc2"&amp;limit=2&amp;descending=true HTTP/1.0
```

Will return:

```
HTTP/1.1 200 OK
Date: Thu, 17 Aug 2006 05:39:28 +0000GMT
Content-Type: application/json
Connection: close

{
  "total_rows": 3, "offset": 1, "rows": [
    {"id": "doc3", "key": "doc3", "value": {"rev":"74EC24"}},
    {"id": "doc2", "key": "doc2", "value": {"rev":"2441HF"}}
  ]
}
```

If we add include_docs=true to a request to _all_docs not only metadata but also the documents themselves are returned.

2.7.7.2 _changes

This allows us to see all the documents that were updated and deleted, in the order these actions are done:

```
GET /somedatabase/_changes HTTP/1.0
```

Will return something of the form:

```
HTTP/1.1 200 OK
Date: Fri, 8 May 2009 11:07:02 +0000GMT
Content-Type: application/json
Connection: close

{"results":[
{"seq":1,"id":"fresh","changes":[{"rev":"1-967a00dff5e02add41819138abb3284d"}]},
{"seq":3,"id":"updated","changes":[{"rev":"2-7051cbe5c8faecd085a3fa619e6e6337"}]},
{"seq":5,"id":"deleted","changes":[{"rev":"2-eec205a9d413992850a6e32678485900"}],"deleted": ↩
    true}
],
"last_seq":5}
```

All the view parameters work on _changes, such as startkey, include_docs etc. However, note that the startkey is exclusive when applied to this view. This allows for a usage pattern where the startkey is set to the sequence id of the last doc returned by the previous query. As the startkey is exclusive, the same document won't be processed twice.

2.8 Replication

CouchDB replication is a mechanism to synchronize databases. Much like _rsync_synchronizes two directories locally or over a network, replication synchronizes two databases locally or remotely.

In a simple POST request, we tell CouchDB the source and the target of a replication and CouchDB will 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.

First, we will create a target database. Note that CouchDB would not automatically create a target database for us, and will return a replication failure if the target doesn't exist:

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

Now we can use the database *albums-replica* as a replication target:

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"shopcart","target":"shopcart- ↩
    replica"}' -H "Content-Type: application/json"
```

Chapter 3

Design documents

3.1 Introduction

Design documents are a special type of CouchDB document that contains application code. As it runs inside a database, the application API is highly structured. In this article, we'll take a look at the function APIs, and talk about how functions in a design document are related within applications.

3.2 Show

CouchDB is a document database which one would call key/value store. It allows for storage of JSON documents that are uniquely identified by keys.

CouchDB is build on the web and for the web. Besides the JSON storage structure and its innate ability to scale horizontally, the CoucbDB creators have build some pretty awesome features that make it very appealing for a particular type of an application. The task is to decide whether the application you're building is that application.

CouchDB exposes a RESTful API, so it is rather easy to use it from any language which supports HTTP. Most popular languages have abstraction libraries on top of that, to abstract away the HTTP layer.

Here is a list of available clients: http://wiki.apache.org/couchdb/Basics.

For our purposes we're going to use curl, a command line utility which allows us to make HTTP requests. So let's see how we can easily accomplish this with CouchDB.

Now that we have installed CouchDB and it is successfully running, let's create a database and insert some sample data.

```
curl -X PUT http://localhost:5984/sample_db
```

The line above create a database called sample_db. If the command is successful, we will see the following output: {``ok":true}

Now lets add three files to this database.

```
curl -X PUT -d @rec1.json http://localhost:5984/sample_db/record1
curl -X PUT -d @rec2.json http://localhost:5984/sample_db/record2
curl -X PUT -d @rec3.json http://localhost:5984/sample_db/record3
```

Again, each command should yield a JSON response with ``ok" set to true if the addition succeeded. Here is what one would expect from the first command:

```
{"ok":true,"id":"record1"?"rev":"1-7c15e9df17499c994439b5e3ab1951d2"?}
```

Again, ok is set to true making this a success response. The id field is set to the name of the record which we created. You can see that names are set through the URL as they are just resources in the world of REST. The rev field displays the revision of this document. CouchDB's concurrency model is based on MVCC, though it versions the documents as it updates them, so each document modification gets it's unique revision id.

Below are the JSon Files:

rec1.json

```
{
  "name": "John Doe",
  "date": "2001-01-03T15:14:00-06:00",
  "children": [
    {"name": "Brian Doe", "age": 8, "gender": "Male"},
    {"name": "Katie Doe", "age": 15, "gender": "Female"}
  ]
}
```

rec2.json

```
{
  "name": "Ilya Sterin",
  "date": "2001-01-03T15:14:00-06:00",
  "children": [
    {"name": "Elijah Sterin", "age": 10, "gender": "Male"}
  ]
}
```

rec3.json

```
{
  "name": "Emily Smith",
  "date": "2001-01-03T15:14:00-06:00",
  "children": [
    {"name": "Mason Smith", "age": 3, "gender": "Male"},
    {"name": "Donald Smith", "age": 2, "gender": "Male"}
  ]
}
```

CouchDB supports views. They are used to query and report on the data stored in the database. Views can be permanent, meaning they are stored in CouchDB as named queries and are accessed through their name. Views can also be temporary, meaning they are executed and discarded.

CouchDB computes and stores view indexes, so view operations are very efficient and can span across remote nodes. Views are written as map/reduce operations, though they land themselves well for distribution.

3.3 Shows & lists

There are two really cool features, which allow for more effective data filtering and transformation. These features are shows and lists. The purpose of shows and lists is to render a JSON document in a different format. Shows allow to transform a single document into another format. A show is similar to a view function, but it takes two parameters function (doc, req), doc is the document instance being iterated and request is an abstraction over CouchDB request object.

Here is a simple show function

```
function(doc, req) {
  var person = <person />;
  person.@name = doc.name;
  person.@joined = doc.date;
  person.children = <children />;
  if (doc.children) {
    for each (var chldInst in doc.children) {
      var child = <child />;
      child.text()[0] = chldInst.name;
      child.@age = chldInst.age;
      child.@gender = chldInst.gender;
      person.children.appendChild(child);
    }
  }
  return {
    'body': person.toXMLString(),
    'headers': {
      'Content-Type': 'application/xml'
    }
  }
}
```

This show function takes a particular JSON record and turns it into XML. Creating a show is pretty simple, you just encapsulate the function above into a design document and create the record through PUT.

Here is the design document for the show above, xml_show.json:

```
{
  "shows": {
    "toxml": "Here you inline the show function above.  Make sure all double quotes are ↩
        escaped..."
  }
}
```

Once we have the design document, we can create it:

```
curl -X PUT -H "Content-Type: application/json" -d '@xml_show.json' http://localhost:5984/ ↩
    sample_db/_design/shows
```

Note: In (../_design/shows), shows is just a name of the design document, we can call it.

Now let's invoke the show, as follows:

```
curl -X GET http://localhost:5984/sample_db/_design/shows/_show/toxml/record1
```

Here is the output:

```
<person name="John Doe" joined="2001-01-03T15:14:00-06:00">
  <children>
    <child age="8" gender="Male">Brian Doe</child>
    <child age="15" gender="undefined">Katie Doe</child>
  </children>
</person>
```

So, how would I transform a record collection or view results into a different format? Well, this is where lists come in. Lists are similar to shows, but they are applied to the results of an already present view. Here is a sample list function.

```
function(head, req) {
  start({'headers': {'Content-Type': 'application/xml'}});
  var people = <people/>;
  var row;
  while (row = getRow()) {
    var doc = row.value;
    var person = <person />;
    person.@name = doc.name;
    person.@joined = doc.date;
    person.children = <children />;
    if (doc.children) {
      for each (var chldInst in doc.children) {
        var child = <child />;
        child.text()[0] = chldInst.name;
        child.@age = chldInst.age;
        child.@gender = chldInst.gender;
        person.children.appendChild(child);
      }
    }
    people.appendChild(person);
  }
  send(people.toXMLString());
}
```

Again, we can encapsulate this list function into a design document, along with a simple view function:

```
xml_list.json
{
  "views": {
    "all": {
      "map": "function(doc) { emit(null, doc); }"
    }
  },
  "lists": {
    "toxml": "Here you inline the show function above.  Make sure all double quotes are ↩
        escaped as it must be stringified due to the fact that JSON can't store a function ↩
        type."
  }
}
```

Now, we create the design document:

```
curl -X PUT -H "Content-Type: application/json" -d @xml_list.json http://localhost:5984/ ↩
    sample_db/_design/lists
```

Once the design document is created, we can request our xml document listing all person records:

```
curl -X GET http://localhost:5984/sample_db/_design/lists/_list/toxml/all
```

And the output is:

```
<people>
  <person name="John Doe" joined="2001-01-03T15:14:00-06:00">
    <children>
      <child age="8" gender="Male">Brian Doe</child>
      <child age="15" gender="Female">Katie Doe</child>
    </children>
  </person>
  <person name="Ilya Sterin" joined="2001-01-03T15:14:00-06:00">
    <children>
      <child age="10" gender="Male">Elijah Sterin</child>
    </children>
  </person>
  <person name="Emily Smith" joined="2001-01-03T15:14:00-06:00">
    <children>
      <child age="3" gender="Male">Mason Smith</child>
      <child age="2" gender="Male">Donald Smith</child>
    </children>
  </person>
</people>
```

With this example we can see how shows and lists are really useful and provide a convenient way to transform views into different formats.

3.4 Map Reduce

For experienced relational database programmers, MapReduce can take some time 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.

Map 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.

CouchDB views are stored as rows that 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, our primary goal is to build an index that stores related data under nearby keys.

Before we can run an example MapReduce view, we'll need some data to run it on. We will 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

        }

}
```

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
        }

}
```

And finally, the document for bananas:

```
{

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

Imagine 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.

Edit 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);
        }
        }
}
```

This is a java 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.

Click "Run" and we should see result rows 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. In this case, we'll emit an array of [item, price] so that CouchDB groups by item type and price.

Let'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);
        }
        }
}
```

Here, 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 java 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.

Once 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.

View 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.

3.5 Validation

CouchDB 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 be authored only 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.

3.5.1 Document Validation Functions

To ensure that users may save only documents that provide these fields, we can validate their input by adding another member to the _design/ document: the validate_doc_update function. CouchDB sends functions and documents to a java interpreter. This mechanism is what allows us to write our document validation functions in java. 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.

Document validation is optional. If we 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 we 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 of them pass the validation, does the write succeed. The order of the validation execution is not defined. Each validation function must act on its own.

Validation 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, we will use java code like the following:

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

When we are trying to prevent an authorized user from saving invalid data, we will use :

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

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

If no exceptions are thrown, CouchDB expects the incoming document to be valid and will write it to the database. By using java to validate JSON documents, we can deal with any structure a document might have. Validation can also be a valuable form of documentation.

3.5.2 Validation's Context

Before 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.

Validation 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).

3.5.3 Writing One

The 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) {}
```

Above is the simplest possible validation function, which 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'});
}
```

Note that if we install this function in database, we 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.

We can see from these examples that the return value of the function is ignored. Validation functions prevent document updates by raising errors. When the validation function passes without raising errors, the update is allowed to proceed.

3.5.4 Type

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

A common pattern in most languages, frameworks, and databases is using types to distinguish between subsets of our data.

CouchDB itself has no notion of types, but they are a convenient shorthand for use in application code, including MapReduce 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 we use.

Here's an example validation function that runs only on posts:

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

Since 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
  }
}
```

It 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 :

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

This 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.

In practice, many applications end up using a mix of typed and untyped validations. 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.

3.5.5 Required Fields

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

Required fields also make display logic much simpler. If we know for certain that all documents will have a field, we can avoid lengthy conditional statements to render the display differently depending on document structure.

If a design document requires a different set of fields on posts and comments. Here's a subset of the 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");
  }
}
```

This is our first look at actual validation logic. We can see that the actual error throwing code has been wrapped in a helper function. Helpers like the require function just shown 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 it ensures that the field is not empty or blank.

Once 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.

3.5.6 Timestamps

Timestamps are an interesting problem in validation functions. Because validation functions are run at replication time as well as during normal client access, we can not have as a requirement the timestamps to be set close to the server's system time. We require two things: that timestamps do not change after they are initially set, and that they are well formed.

First, 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");
}
```

The unchanged helper is more complex than the require helper. 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 fields it is used on are the same between the new and the old version of the document.

java's equality test is not well suited to working with deeply nested objects. We use CouchDB's java runtime's built-in toJSON function in our equality test, which is better than testing for raw equality.

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

java 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, java 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 objects have the same contents. This is not guaranteed to work for deeply nested objects, as toJSON may serialize objects in an undefined order.

Chapter 4

Building a Blog Application with CouchDB

4.1 Introduction

The current chapter is an effort to build a blog application using the Javascript based Web Server Node.js along with CouchDB.

To make this application, we have selected:

- The swig client side javascript template engine (Refer to http://paularmstrong.github.io/swig/ for more documentation about this)
- Node.js for server side development
- Middleware handling with express.js
- CouchDb as the database
- Node.js Cradle Module Extension (to make communication with CouchDB)

Since this lessons is primarily about couchdb, we will discuss the CouchDB related functionalities in detail. We will also discuss about the related blog application functionality.

Please refer to the couchdb-blog.zip file for the source code of the application.

Node.js is a javascript runtime. Unlike traditional web servers, there is no separation between the web server and the code, and we do not have to customize configuration files (XML or Property Files) to get the Node.js Web Server up. With Node, we can create the web server with minimal code and deliver content with the code. We will describe in this lesson how to create a web server with Node and how to work with static and dynamic file content. Additionally, we will talk a bit about performance tuning in the Node.js Web server.

We have used the Node.js server for the web controller and the routing of the contents. The persistence and fetching of the data will be done through CouchDB - a package for couchdb handling (cradle) will be installed through NPM (Node Package Manager Registry). The front end rendering will be performed by Swig (a JavaScript Template Engine), that also is installed with the node package manager consolidated module (swig comes with this).

Below is the app architecture:

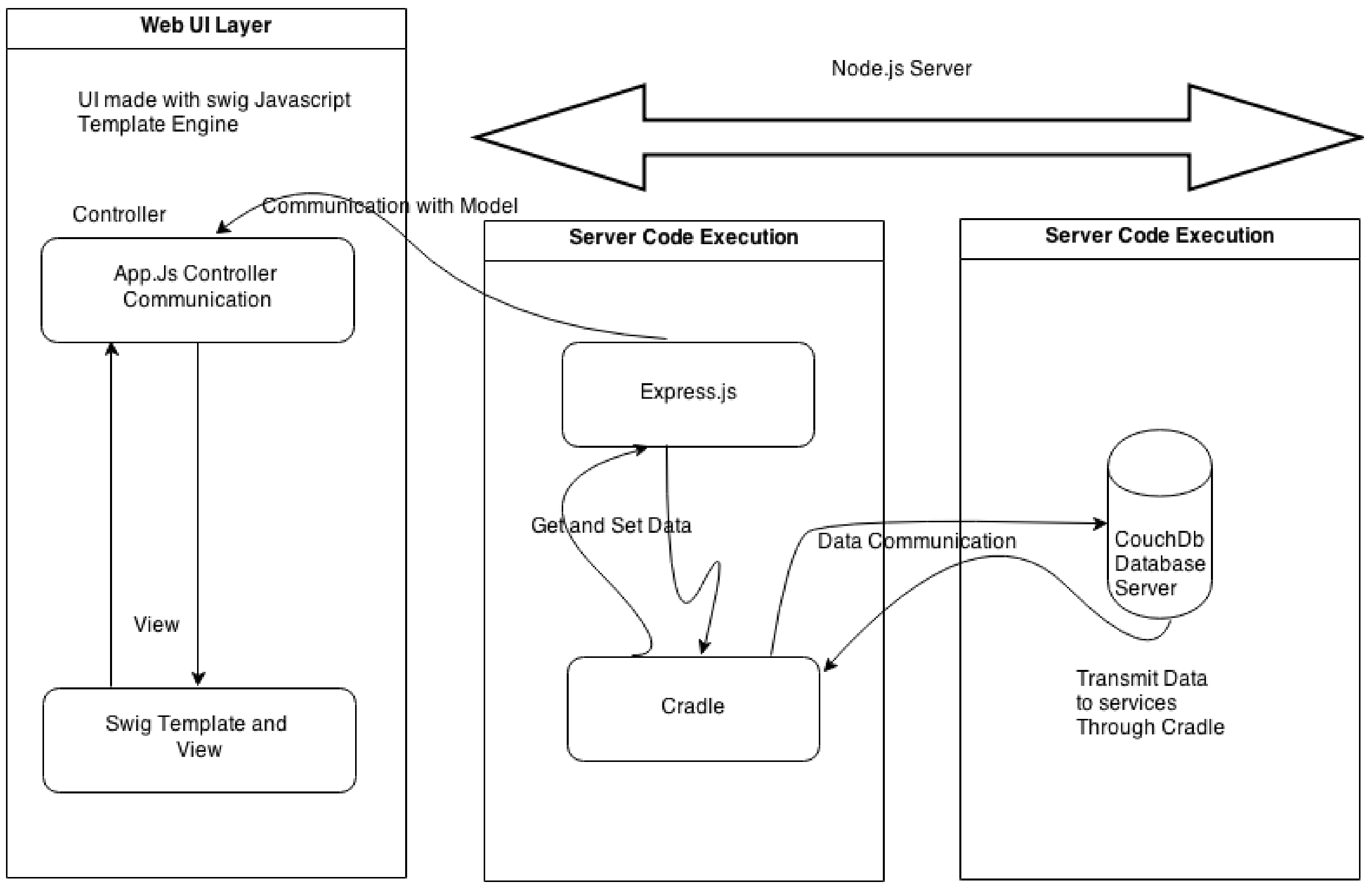


Figure 4.1: screenshot

In the application directory (see attached file), the "**package.json**" can be found. Additionally, all the node.js library dependencies are also there.

To run all the required libraries, we need to run "**npm install**" in the terminal. We need to start the application using **node app.js**

4.2 The App.js functionality

```
var express = require('express')
  , app = express() // Web framework to handle routing requests
  , cons = require('consolidate') // Templating library adapter for Express
  , routes = require('./routes'); // Routes for our application
var cradle = require('cradle'); // Driver for node.js Couchdb Driver
```

The above code is used for the initialization of the libraries.

Following are the database credentials. Please note that before running the code below, the database must be present (i.e. to be created in couchdb). We can easily accomplish this with futon (couchdb administrative console):

```
var databaseUrl = "blogdb";

var connection = new(cradle.Connection)('http://piyas:secure2013@localhost', 5984, {
      auth: { username: 'piyas', password: 'secure2013' }
  });

var db = connection.database(databaseUrl);
```

Below is the code for registering the swig templating engine in node.js express middleware.

```
app.engine('html', cons.swig);
app.set('view engine', 'html');
app.set('views', __dirname + '/views');
```

4.3 Notes about the Express.js module

Express.js is a powerful web development framework for the Node.js (Node) platform. It comes with the Node.js middleware modules. These components are JavaScript components which can be used in Express.js based web applications to make the application modular and structured in layers.

With express.js, other node.js core APIs can also be called except for the express.js apis. The express.js framework can be used to develop any kind of web application - simple to complex. With Express.js development, we have to keep the asynchronous behavious of the application in mind.

4.4 Express.js Objects

4.4.1 The application object

The application object is an instance of Express, generally presented by the variable named *app*. This is the main object of our Express application. All of the application functionality is built using this object.

Following, we create an instance of the Express.js module within the node application:

```
var express = require('express');
```

4.4.2 The request object

Now, when a web client makes a request to the Express application, the HTTP request object is created. All the callbacks in the application, where the request objects are passed as reference, are represented with a conventional variable *req*. This request object holds all the HTTP stack related variables, such as header informations, HTTP methods and related properties for a particular request from the web client.

Below we present some methods of the Request Object which are important in our web application development:

- **req.params**: Holds the values of all the parameters of the request object
- **req.params(name)**: Returns the value of a specific parameter from the GET params or POST params
- **req.query**: Takes values of a GET method submission
- **req.body**: Takes values of a POST form submission
- **req.get(header)**: Gets the request HTTP header
- **req.path**: The request path
- **req.url**: The request path with query parameters

4.4.3 The response object

The response object is created along with the request object and is generally represented by a variable named *res*. In the HTTP Request-Response model, all the express middleware functions work on the request and the response object, while passing the control one after another.

Some methods of the Response Object which are important in our web application development -

- **res.status(code)**: The HTTP response code
- **res.attachment([filename])**: The response HTTP header Content-Disposition to attachment
- **res.sendfile(path, [options])**: Sends a file to the client [callback]
- **res.download(path, [filename])**: Prompts the client to download from [callback]
- **res.render(view, [locals])**: Renders a view callback

4.5 Concepts used in Express

4.5.1 Asynchronous JavaScript

Node.js programming is mainly done with Asynchronous Javascript Programming. All of the modules in node.js are built based on an asynchronous nature. So, the execution of code from one layer to another generally occurs within callback functions. Node and Express are built on the concept of async operations, and all the results are handled in callback functions.

As a node.js program executes in an event loop, the end user generally does not have to wait for a response from the view layer i.e web browser or mobile browser etc. Generally, the callback function is passed to an async function to be executed and this returns the result to an upper function, when the execution of code is completed within the callback function.

All the programs within express.js and associated programs are installed on the node.js environment as node modules. For any node.js application, the deployment configurations are written in package.json file. If we need to install the application as a node module in the node.js environment, i.e. through the npm install command, we should include the package.json file.

4.5.2 Middlewares in node.js applications

A middleware in node.js application context is a JavaScript function to handle HTTP requests to an Express.js application. It will be able to handle the request and the response objects from the HTTP request, perform some operation on the request, send the response to the client and will be able to pass the objects/results to the next middleware.

Middlewares are loaded in an Express application with app.use() method.

A basic example of a middleware can be for a GET method of a request object, as follows:

```
app.use(express.cookieParser());
app.use(express.bodyParser());
```

The majority of the Express.js functionality is implemented with its built-in middlewares. One example of an Express.js middleware is the router middleware, which is responsible for routing the HTTP requests to Express applications and to the appropriate data handler functions. From a user perspective, it is the navigational functionality in a web application.

The destinations of the HTTP request URIs are defined via routes in the application. Routes are the controlling points for the response from a request, i.e they decide where to route a specific request by analysing the data in the request object. In traditional web application, like in a J2ee Application, this functionality is handled by the Controller in the application. Route handlers may be defined in the app.js file or loaded as a Node module.

Now let’s see the routes function:

```
// Application routes
routes(app, db);
```

This code resides by default in the index.js file (inside the routes folder).

We can see the code in index.js:

```
app.get('/newpost', contentHandler.displayNewPostPage);
```

So, whenever the application gets a request with */newpost*, it will go to the displayNewPostPage function of the contentHandler function i.e. in the content.js file.

Now if we can go to displayNewPostPage function, we can see:

```
this.displayNewPostPage = function(req, res, next) {
        "use strict";

        if (!req.username) return res.redirect("/login");

        return res.render('newpost_template', {
            subject: "",
            body: "",
            errors: "",
            tags: "",
            username: req.username
        });
    }
```

Here, if the *username* variable is not present in request object, the application will redirect the user to the *login* action. Otherwise, the response object will render the template *newpost_template*, which will show the html file *newpost_template.html* in *views* folder. Almost all the web flow executes in a similar way in this application. All the web flows are documented within *index.js* of *routes* folder.

Now we configure the application to listen in a specific port using the following code:

```
app.listen(8082);
```

4.5.3 Works in index.js

Generally the index.js contains all the controller functionality. Let's see the following example:

```
app.post('/newpost', contentHandler.handleNewPost);
```

Here the function in content.js is the following:

```
this.handleNewPost = function(req, res, next) {
        "use strict";

        var title = req.body.subject
        var post = req.body.body
        var tags = req.body.tags

        if (!req.username) return res.redirect("/signup");

        if (!title || !post) {
            var errors = "Post must contain a title and blog entry";
            return res.render("newpost_template", {subject:title, username:req.username, ↩
                body:post, tags:tags, errors:errors});
        }

        var tags_array = extract_tags(tags)

        var escaped_post = sanitize(post).escape();
        var formatted_post = escaped_post.replace(/r?n/g,'<br>');

        posts.insertEntry(title, formatted_post, tags_array, req.username, function(err,  ↩
            permalink) {
            "use strict";

            if (err) return next(err);

            //redirect to the blog permalink
            return res.redirect("/post/" + permalink)
        });
    }
```

Note that the request object values are taken in the function through the *req* object. All the required validations are also performed. Then, the *insertEntry* function of posts.js will be called to save the data. This function will be described later.

All the functions related to couchdb database handling are explained below.

4.6 CouchDB Handling

Before we are able to fetch data from CouchDB, we will need to create the appropriate views for CouchDB key,value stores. Please refer to our previous douments for view creation and map function creation in couchdb.

Before working on the application we have to create the following views:

- To access user data through user name, we have to create the following design document in the *blogdb* database:

```
db.save('_design/user', {
    views: {
      byUsername: {
        map: 'function (doc) { if (doc.type === "user") { emit(doc.username, doc) } }'
      }
    }
  });
```

This will create the map function with a key as username and a value as document. * To access user data specific to one session, we have to create the following design document in the *blogdb* database:

```
db.save('_design/session', {
    views: {
      bySessionid: {
        map: 'function (doc) { if (doc.type === "session") { emit(doc._id, doc) } }'
      }
    }
  });
```

- To access posts in the couchdb database, we have to create the following design document in the *blogdb* database:

```
db.save('_design/post', {
    views: {
      byPosts: {
        map: 'function (doc) { if (doc.type === "post") { emit(doc._id, doc) } }'
      },
      byTags: {
        map: 'function (doc) { if (doc.type === "post") { emit(doc.tags, doc) } }'
      },
      byPermalinks: {
        map: 'function (doc) { if (doc.type === "post") { emit(doc.permalink, doc) } }'
      }

    }
  });
```

Note the following:

- *byPosts* function will create *_id* as the key and whole document as value.
- *byTags* function will create *tags* as the key and whole document as value.
- *byPermalinks* function will create *tags* as the key and whole document as value.

Now let's discuss the main functions for CouchDB handling, invoked through cradle:

4.6.1 Reference - posts.js

- function PostsDAO(db) - We can have a reference of the Database through this constructor (db).
- Insert the post (Creation of the Document, simple json Document):

```
var post = {"title": title,
                "author": author,
                "body": body,
                "permalink":permalink,
                "tags": tags,
                "comments": [],
                "type":"post",
                "date": new Date()}
```

We handle the document persistence through cradle:

```
// insert the post
        db.save('POSTID_'+Math.random(), post, function (err, res) {
              if (err) {
                  // Handle error
                  res += ' SAVE ERROR: Could not save record!!n';
                  callback(err, null);
              } else {
                  // Handle success
                  res += ' SUCESSFUL SAVEn';
                  callback(err, permalink);
              }
           });
```

The first argument to the *save* function expects a unique id field. The second argument is the *post* variable. The third argument is the handle of a function for the response. This will return the callback to the event or functionality with or without error. * Get all the posts (Function name: getPosts):

```
db.view('post/byPosts', { }, function (err, doc) {
                console.dir(doc);
            "use strict";

            if (err) return callback(err, null);

            console.log("Found " + doc.length + " posts");

            callback(null, doc);

        });
```

The first argument is the design document name. The second argument is the combination of the key and the value by which the document will be accessed. If it is empty json, then it will return all the documents. The third argument is the handle of a function with returned document/documents. * Get all Documents by tags array (Function name: getPostsByTag):

```
db.view('post/byTags', { key : tagArray }, function (err, doc) {
            console.dir(doc);
            "use strict";

            if (err) return callback(err, null);

            console.log("Found " + doc.length + " posts");

            callback(null, doc);

        });
```

The first argument is the design document name. The second argument is the combination of key and value by which the document will be accessed. The third argument is the handle of a function with returned document/documents. * Get all Documents by permalinks (Function name: getPostByPermalink):

```
db.view('post/byPermalinks', { key : permalink }, function (err, doc) {
            console.dir("in permalink --" + doc);
            "use strict";

            if (err) return callback(err, null);

            console.log("Found " + doc.length + " posts");

            callback(null, doc);

        });
```

The first argument is the design document name. The second argument is the combination of key and value by which the document will be accessed. The third argument is the handle of a function with returned document.

4.6.2 Reference - sessions.js

- function SessionsDAO(db) - The reference of the Database is done through this constructor (db).
- save the doucment for Session (Function name: startSession) - Creating a session id:

```
var session_id = crypto.createHash('sha1').update(current_date + random).digest('hex');
```

Creating as session document:

```
var session = {'username': username, '_id': session_id,'type':'session'}
```

Saving a document:

```
db.save(session_id, session, function (err, res) {
                console.log(err);

              if (err) {
                  // Handle error
                  res += ' SAVE ERROR: Could not save record!!n';
              } else {
                  // Handle success
                  res += ' SUCESSFUL SAVEn';
              }
              console.log('session start.2'+res);
```

```
                callback(err, session_id);

            });
```

- get username for a session (Function name: getUsername):

```
db.view('session/bySessionid', { key: session_id }, function (err, doc) {
                console.dir(doc);
            "use strict";

            if (err) return callback(err, null);

            if (!doc) {
                callback(new Error("Session: " + doc + " does not exist"), null);
                return;
            }

            callback(null, doc[0].value.username);

        });
```

The first argument is the design document name. The second argument is the combination of key and value by which the document will be accessed. The third argument is the handle of a function with returned document.

4.6.3 Reference - users.js

Adding a user (Function name: addUser):

```
var user = {'email': email, 'password': password_hash, 'username': username,'type':'user'};
```

Saving the document in CouchDB:

```
db.save('ID_'+Math.random(), user, function (err, res) {
              if (err) {
                  // Handle error
                  res += ' SAVE ERROR: Could not save record!!n';
              } else {
                  // Handle success
                  res += ' SUCESSFUL SAVEn';
              }
              return callback(err, null);
            });
```

validate a user (Function name: validateLogin):

```
db.view('user/byUsername', { key: username }, function (err, doc) {
            if (err) return callback(err, null);
            if (doc) {

                if (bcrypt.compareSync(password, doc[0].value.password)) {
                    console.log('here we are');
                    callback(null, doc);
                }
                else {
                    var invalid_password_error = new Error("Invalid password");
                    // Set an extra field for any error which is not a db error
                    invalid_password_error.invalid_password = true;
                    callback(invalid_password_error, null);
                }
            }
            else {
                var no_such_user_error = new Error("User: " + user + " does not exist");
                no_such_user_error.no_such_user = true;
                callback(no_such_user_error, null);
            }

        });
```

The first argument is the design document name. The second argument is the combination of key and value by which the document will be accessed.The third argument is the handle of a function with returned document.

To summarize, in this article, we have talked about:

- Running an application in Node.js web server
- A minimal routing for data handling
- Handling data from a GET request
- Handling data from a POST request
- Handling CouchDB functions through cradle
- View creation and data access in couchdb

There are rooms for improvement for the node.js code, like adding blog comment support, improving the business logic, like handling duplicate post, adding a category for the blog etc. These enhancements are left as an exercise for the reader.

You may download the source code here.

Chapter 5

Deploying and Optimizing CouchDB

5.1 Scaling

Scaling, or scalability, doesn't refer to a specific technique or technology, but rather is an attribute of a specific architecture.

In this lesson we shall cover the scaling of CouchDB, a popular NoSQL database. For CouchDB, we can scale three general properties:

- Read requests
- Write requests
- Data

5.1.1 Scaling Read Requests

A read request retrieves a piece of information from the database. It follows these stations within CouchDB: First, the HTTP server module needs to accept the request. For that, it opens a socket to send over the data. The next station is the HTTP request handle module, which analyzes the request and directs it to the appropriate submodule inside 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.

All this takes processing time and additionally there must be enough sockets (or file descriptors) available. The storage backend of the server must be able to fulfill all these 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 process only so many concurrent requests.

The 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 speed. Requests that can use this cache don't ever hit database and are thus less IO Operation intensive.

5.1.2 Scaling Write Requests

A write request is similar to a read request which reads a piece of data from disk, but it writes it back after modifying it. Remember, the nice thing about reads is that they're cacheable. A cache must be notified when a write changes the underlying data, or the clients must be notified to not use the cache. If we have multiple servers for scaling reads, a write must occur on all servers.

5.1.3 Scaling Data

The third way of scaling is scaling data. Today's hard drives are cheap and provie a lot of capacity, and they will only get better in the future, but there is only so much data a single server can make sensible use of. It must also maintain one or more indexes to the data, thus it uses even moredisk space.

The solution is to chop the data into manageable chunks and put each chunk on a separate server. In this way, all servers with a chunk will form a cluster that holds all your data.

While we are taking separate looks at scaling of reads, writes, and data, these rarely occur isolated. Decisions to scale one will affect the others.

5.2 Replication

A replicator simply connects to two DBs as a client, then reads from one and writes to the other. Push replication is reading the local data and updating the remote DB; pull replication is vice versa.

- The replicator is actually an independent Erlang application, running on its own process. It connects to both CouchDBs, then reads records from one and writes them to the other.
- CouchDB has no way of knowing who is a normal client and who is a replicator (let alone whether the replication is push or pull). It all looks like client connections. Some of them read records, some of them write records.

The CouchDB Replication protocol is a synchronization protocol for synchronizing documents between 2 peers over HTTP/1.1.

5.2.1 Algorithm

The replication algorithm can be explained as follows:

- Assign an unique identifier to the source Database. Most of the time it will be the URI.
- Save this identifier in a special Document named _local/<uniqueid> on the Target database. This document isn't replicated. It will collect the last Source sequence ID, the Checkpoint, from the previous replication process.
- Get the Source changes feed by passing it the Checkpoint using the since parameter by calling the /<source>/_changes URL. The changes feed only return a list of current revisions.

Note: This step can be performed continuously using the feed=longpoll or feed=continuous parameters. Then the feed will continuously get the changes.

Collect a group of Document/Revisions ID pairs from the changes feed and send them to the target databases on the /<target>/_revs_diffs URL. The result will contain the list of revisions NOT in the Target database.

GET each revision from the source Database by calling the URL /<source>/<docid>?revs=true&rev=<revision> . This will get the document with the parent revisions. Also don't forget to get attachements that aren't already stored at the target. As an optimization, we can use the HTTP multipart API to retrieve them all.

Collect a group of revisions fetched at the previous step and store them on the target database using the Bulk Docs API with the new_edit: false JSON property to preserve their revisions ID.

After the group of revision is stored on the Target Database, save the new Checkpoint on the Source database.

Note: Even if some revisions have been ignored, the sequence should be taken into consideration for the Checkpoint.

To compare non numeric sequence ordering, we will have to keep an ordered list of the sequences IDS as they appear in the _changes feed and compare their indices.

One thing to keep in mind is that the _users database, the design documents and the security attributes of the databases are not being replicated.

For, the _users database and the design documents, there is solution: We just need to run the process as administrator in order to replicate them.

Only server and database admins can create design docs and access views:

```
curl -H 'Content-Type: application/json' -X POST http://localhost:5984/_replicate -d ' {" ↩
    source": "http://admin:admin_password@production:5984/foo", "target": "http://admin: ↩
    admin_password@stage:5984/foo", "create_target": true, "continuous": true} '
```

This POST request will also work with the _users database.

Replication is a one-off operation: we 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"} -H "Content-Type: application/ ↩
    json"
```

This 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."

To send changes from the target to the source database, just make the same HTTP requests, only with source and target database swapped.

```
POST /_replicate HTTP/1.1
Content-Type: application/json
{"source":"http://example.org/database","target":"database"}
```

A remote database is identified by the same URL we use to talk to it. CouchDB replication works over HTTP using the same mechanisms that are available to us. This example shows that replication is a unidirectional process. Documents are copied from one database to another and not automatically vice versa. If we want bidirectional replication, we trigger two replications with source and target swapped.

When we ask CouchDB to replicate one database to another, 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 are.

Databases 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 53 and now?" by returning a list of new and changed documents. Finding the differences between databases this way is an efficient operation. It also adds to the robustness of replication.

We can use replication on a single CouchDB instance to create snapshots of our databases to be able to test code changes without risking data loss or to be able to refer back to older states of our database. But replication gets really fun if we use two or more different computers, potentially geographically spread out.

With 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 CouchDBs in an inconsistent state. Then, when the problems are gone and we trigger replication again, it continues where it left off.

5.2.2 Simple Replication with the Admin Interface

We can run replication from your web browser using Futon, CouchDB's built-in administration interface. Start CouchDB and open the url to http://127.0.0.1:5984/_utils/. On the righthand side, there is a list of things to visit in Futon. Click on "Replication."

Futon will show an interface to start replication. We can specify a source and a target by either picking a database from the list of local databases or filling in the URL of a remote database.

Click on the Replicate button, wait a bit, and have a look at the lower half of the screen where CouchDB gives us some statistics about the replication run or, if an error occurred, an explanatory message.

5.2.3 Replication in Detail

So far, we've skipped over the result from a replication request. Here's an example:

```
{
  "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,
    }
  ]
}
```

The "ok": true part, 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.

The 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.

The session_id is recorded here again for convenience. The start and end time for the replication session are also recorded. The _last_seq denotes the update_seqs that were valid at the beginning and the end of the session. recorded_seq is the update_seq of the target again. It's 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.

The last three-docs_read, docs_written, and doc_write_failures-show how many documents we read from the source, wrote to the target, and how many failed. If all is well, _read and _written are identical and doc_write_failures is 0. If not, 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.

One 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. We have admins and need 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@127.0.0.1:5984/database"}' -H "Content-Type:  ↩
    application/json"
```

5.2.3.1 Continuous Replication

When we add "continuous": true to the replication trigger object, CouchDB will not stop after replicating all missing documents from the source to the target. It will listen on CouchDB's _changes API and automatically replicate over any new docs as they 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.

```
> curl -X POST http://127.0.0.1:5984/_replicate -d '{"source":"db", "target":"db-replica",  ↩
    "continuous":true}' -H "Content-Type: application/json"
```

CouchDB doesn't remember continuous replications over a server restart. For the time being, we need to trigger them again when you restart CouchDB. In the future, CouchDB will allow us to define permanent continuous replications that survive a server restart without having to do anything.

5.3 Conflict management

CouchDB has a mechanism to maintain continuous replication, so one can keep a whole set of computers in sync with the same data, whenever a network connection is available.

When we replicate two databases in CouchDB and we face conflicting changes, CouchDB will detect this and will 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.

CouchDB does not attempt to merge the conflicting revision. Our application dictates how the merging should be done. The choice of picking the winning revision is arbitrary.

Replication 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. Here 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 we ask any instance for a document, we will get the same answer regardless which one we ask.

Whether or not CouchDB picked the version that our application needs, we need to go and resolve the conflict, just as we need to resolve a conflict in a version control system like Subversion by merging them and save it as the now latest revision. Replicate again and our resolution will populate over to all other instances of CouchDB. Our conflict resolving on one node could lead to further conflicts, all of which will need to be addressed, but eventually, we will end up with a conflict-free database on all nodes.

5.4 Load Balancing

5.4.1 Having a Backup

Whatever the cause is, we want to make sure that the service we are providing 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 keep the server from turning off during such an event, most come with at least two power supplies. To take this further, we could get a server where everything is duplicated (or more). It is much cheaper to get two similar servers where the one can take over if the other has a problem. However, we need to make sure both servers have the same set of data in order to switch them without a user noticing.

Removing all single points of failure will give us a highly available or a fault-tolerant system. The order of tolerance is restrained only by our budget. If we can't afford to lose a customer's shopping cart in any event, we need to store it on at least two servers in at least two far apart geographical locations.

Before we dive into setting up a highly available CouchDB system, let's look at another situation. Suppose that an Online Shopping Site suddenly faces a lot more traffic than usual and that the customers are complaining for the site being "slow". Now, a probable solution for such a scenerio would be to setup a second server which will take some load from first server when the load exceeds a certain threshold.

The solution to the outlined problem looks a lot like the earlier one for providing a fault-tolerant setup: install a second server and synchronize all data. The difference is that with fault tolerance, the second server just sits there and waits for the first one to fail. In the server-overload case, a second server helps answer all incoming requests. This case is not fault-tolerant: if one server crashes, the other will get all the requests and will likely break down, or provide a very slow service, neither of which is acceptable.

Keep in mind that although the solutions look similar, high availability and fault tolerance are not the same. We'll get back to the second scenario later on, but first we will take a look at how to set up a fault-tolerant CouchDB system.

5.5 Clustering

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.

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 there is no worry about losing machines. In a large cluster, we should plan for 5-10% of our machines to experience some sort of failure or reduced performance, so cluster design must prevent node failures from affecting reliability. Finally, we'll look at adjusting cluster layout dynamically by splitting or merging nodes using replication.

5.5.1 Introducing CouchDB Lounge

CouchDB 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 another that distributes view requests.

The dumbproxy handles simple requests for anything that isn't a CouchDB view. This comes as a module for nginx, a high-performance reverse HTTP proxy. Because of the way reverse HTTP proxies work, this automatically allows configurable security, encryption, load distribution, compression, and, of course, aggressive caching of our database resources.

The smartproxy handles only 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 programming framework for Python.

5.5.2 Consistent Hashing

CouchDB's storage model uses unique IDs to save and retrieve documents. Sitting at the core of Lounge is a simple method of hashing the document IDs. Lounge then uses the first few characters of this hash to determine which shard to dispatch the request to. We can configure this behavior by writing a shard map for Lounge, which is just a simple text configuration file.

Because Lounge allocates a portion of the hash (known as a keyspace) to each node, we can add as many nodes as we like. Because the hash function produces hexadecimal strings that bear no apparent relation to our DocIDs, and because we dispatch requests based on the first few characters, we ensure that all nodes see roughly equal load. And because the hash function is consistent, Lounge will take any arbitrary DocID from an HTTP request URI and point it to the same node each time.

This idea of splitting a collection 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 the keyspace between two partitions. The hash function maps from document IDs to positions on the ring. The ring is continuous so that we can always add more nodes by splitting a single partition into pieces. With four physical servers, we can allocate the keyspace into 16 independent partitions by distributing them across the servers like so:

Table 5.1: datasheet

A	0,1,2,3
B	4,5,6,7
C	8,9,a,b
D	c,d,e,f

If the hash of the 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 just shown. This could map to http://B.couches.local/db-7/ on our backend cluster. In this way, the hash table is just a mapping from hashes to backend database URIs. Don't worry if this all sounds very complex; all we have to do is provide a mapping of shards to nodes and Lounge will build the hash ring appropriately-so no need to get our hands dirty if we don't want to.

To 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 principle behind REST and is one of the many benefits 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 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 backend CouchDB server.

Consistent hashing is a simple way to ensure that we can always find the documents we saved, while balancing storage load evenly across partitions. Because the hash function is simple (it is based on CRC32), we are free to implement our own HTTP intermediaries or clients that can similarly resolve requests to the correct physical location of our data.

5.6 Distributed load testing

There are many tools available that allow us to create tests customized for our application. However, when creating a distributed system it can be difficult to actually generate enough load to push our system to its maximum capacity.

In order to stress test a distributed system, we will need a distributed load testing tool. Tsung is a distributed load and stress testing tool that we will use for the example this chapter. We will be using Tsung on Ubuntu, but these steps can be easily adapted to other platforms. Tsung can generate GET and POST HTTP requests and PUT and DELETE HTTP requests. Some of Tsung's features include:

- Monitoring of client operating systems' CPU, memory, and network traffic
- Simulation of dynamic sessions, described in an XML configuration file
- Randomized traffic patterns based on defined probabilities
- Recording of HTTP sessions for later playback during a test

HTML reports and graphs

Like CouchDB, Tsung is developed in Erlang. Depending on the number of testing servers used, Tsung can simulate hundreds of thousands of concurrent users. Given enough servers, we could even simulate millions of concurrent users. In addition to being able to test HTTP servers, Tsung can also test WebDAV, SOAP, PostgreSQL, MySQL, LDAP, and Jabber/XMPP servers.

5.6.1 Installing Tsung

In the following examples, we will have two testing clients with domain names of test- a.example.com and test-b.example.com, and we will be testing our couch-proxy.example.com (load balancer), couch-master.example.com (CouchDB master node), couch- a.example.com (CouchDB read-only node), couch-b.example.com (CouchDB read-only node), couch-c.example.com (CouchDB read-only node) servers.

Install Erlang on both test-a.example.com and test-b.example.com:

```
sudo aptitude install erlang
```

Install gnuplot on both test-a.example.com and test-b.example.com:

```
sudo aptitude install gnuplot
```

Install Perl's Template Toolkit on both test-a.example.com and test-b.example.com:

```
sudo aptitude install libtemplate-perl
```

Install Python's Matplotlib on both test-a.example.com and test-b.example.com:

```
sudo aptitude install python-matplotlib
```

Download the latest version of Tsung on both test-a.example.com and test-b.example.com. As of this writing, this was version 1.5.0:

```
wget http://tsung.erlang-projects.org/dist/tsung-1.5.0.tar.gz
```

Extract the downloaded file on both test-a.example.com and test-b.example.com:

```
tar -xzf tsung-1.5.0.tar.gz
```

On both test-a.example.com and test-b.example.com, change into the tsung-1.5.0 directory:

```
cd tsung-1.5.0
```

Configure on both test-a.example.com and test-b.example.com:

```
sudo ./configure
```

Make on both test-a.example.com and test-b.example.com:

```
sudo make
```

Install on both test-a.example.com and test-b.example.com:

```
sudo make install
```

We will be launching our tests from test-a.example.com, so one will need to be able to login from this client to test-b.example.com without using a password. We will do this using public key authentication, but one could instead use ssh-agent or rsh.

Install the OpenSSH server on test-a.example.com and test-b.example.com, if it is not already installed:

```
sudo aptitude install openssh-server
```

Generate an SSH key on test-a.example.com:

```
ssh-keygen
```

Pick the default file in which to save the key, likely ~/.ssh/id_rsa. Enter a passphrase.

From test-a.example.com, copy the ~/.ssh/id_rsa public key to test-b.example.com:

```
scp ~/.ssh/id_rsa.pub test-b.example.com:~
```

If this is the first time using SSH to connect from test-a.example.com to test-b.exam ple.com, we will need to accept the RSA key fingerprint. Enter the user's password and we should see output indicating that the file has been copied.

Log into test-b.example.com and add the public key copied from test-a.example.com to test-b.example.com's list of authorized keys:

cat /id_rsa.pub >> /.ssh/authorized_keys Still on test-b.example.com, remove the public key that was copied over from test- a.example.com: rm ~/id_rsa.pub

To test, try logging into test-b from test-a.example.com, accepting the RSA key fingerprint if prompted, and it should not be prompted for a password: ssh test-b

If there is additional testing clients, repeat the above steps for each, setting up test- a.example.com to be able to log into each testing machine without using a password. The more testing servers you have, the more simulated load can be generated.

Oddly enough, test-a.example.com will also need to be able to log into itself without using a password. To add its own public key to its list of authorized keys, from test- a.example.com:

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

To test, try logging into test-a from test-a.example.com (yes, from itself), accepting the RSA key fingerprint if prompted, and it should not be prompted for a password: ssh test-a

5.6.2 Configuring Tsung

Tsung comes with an example configuration file for doing distributed HTTP testing, which we'll find in /usr/share/doc/tsung/examples/http_distributed.xml. We will create our own configuration file, saved to ~/http_distributed_couch_proxy.xml (see the Tsung User's manual at http://tsung.erlang-projects.org/user_manual.html):

```
<?xml version="1.0"?> <!DOCTYPE tsung SYSTEM "/usr/share/tsung/tsung-1.0.dtd"> <tsung  ↩
    loglevel="notice" version="1.0">
<!-- Client side setup --> <clients>
<client host="test-a" weight="1" maxusers="10000" cpu="4"/>
<client host="test-b" weight="1" maxusers="10000" cpu="4"/> </clients>
```

```
<!-- Server side setup --> <servers>
<server host="couch-proxy" port="80" type="tcp"/> </servers>
<!-- Load setup --> <load>
<arrivalphase phase="1" duration="5" unit="minute"> <users arrivalrate="200" unit="second"> ↩
    </users>
</arrivalphase> </load>
<!-- Sessions setup --> <sessions>
<session name="post_get" probability="2.5" type="ts_http"> <thinktime value="10" random=" ↩
    true"/> <setdynvars sourcetype="random_number" start="2008" end="2011">
<var name="yyyy"/> </setdynvars> <setdynvars sourcetype="random_number" start="10" end="12" ↩
    >
<var name="mm"/> </setdynvars> <setdynvars sourcetype="random_number" start="10" end="28">
<var name="dd"/> </setdynvars> <request subst="true">
<match do="abort" when="nomatch">201 Created</match> <dyn_variable name="id" jsonpath="$.id ↩
    "/> <dyn_variable name="rev" jsonpath="$.rev"/> <http
>
method="POST" url="/api" content_type="application/json" contents="{
&quot;date&quot;:[ &quot;%%_yyyy%%&quot;, &quot;%%_mm%%&quot;, &quot;%%_dd%%&quot;      ]
}"
<http_header name="Accept" value="application/json"/> </http>
</request> <for from="0" to="9" incr="1" var="x">
<thinktime value="10" random="true"/> <request subst="true">
<match do="abort" when="nomatch">304 Not Modified</match> <http method="GET" url="/api/%% ↩
    _id%%">
<http_header name="If-None-Match" value="&quot;%%_rev%%&quot;"/>
<http_header name="Accept" value="application/json"/> </http>
</request> </for>
</session>
<session name="put_get" probability="2.5" type="ts_http"> <thinktime value="10" random=" ↩
    true"/> <setdynvars sourcetype="random_string" length="32">
<var name="id"/> </setdynvars> <setdynvars sourcetype="random_number" start="2008" end=" ↩
    2011">
<var name="yyyy"/> </setdynvars> <setdynvars sourcetype="random_number" start="10" end="12" ↩
    >
<var name="mm"/> </setdynvars> <setdynvars sourcetype="random_number" start="10" end="28">
<var name="dd"/> </setdynvars>
<request subst="true"> <match do="abort" when="nomatch">201 Created</match> <dyn_variable  ↩
    name="rev" jsonpath="$.rev"/> <http
method="PUT" url="/api/%%_id%%" content_type="application/json" contents="{
&quot;date&quot;:[ &quot;%%_yyyy%%&quot;, &quot;%%_mm%%&quot;, &quot;%%_dd%%&quot;
] }"
<http_header name="Accept" value="application/json"/> </http>
</request> <for from="0" to="9" incr="1" var="x">
<thinktime value="10" random="true"/> <request subst="true">
<match do="abort" when="nomatch">304 Not Modified</match> <dyn_variable name="rev" jsonpath ↩
    ="$._rev"/> <http method="GET" url="/api/%%_id%%">
<http_header name="If-None-Match" value="&quot;%%_rev%%&quot;"/>
<http_header name="Accept" value="application/json"/> </http>
</request> </for>
</session>
<session name="view_pagination" probability="20" type="ts_http"> <thinktime value="10"  ↩
    random="true"/> <request subst="true">
<http method="GET" url="/api/_design/default/_view/dates?reduce=false&amp;skip=0&amp;limit ↩
    =10"
>
<http_header name="Accept" value="application/json"/> </http>
</request> <for from="10" to="90" incr="10" var="skip">
<thinktime value="10" random="true"/> <request subst="true">
<http method="GET" url="/api/_design/default/_view/dates?reduce=false&amp;skip=%%_skip%%
&amp;limit=10&amp;stale=ok" >
<http_header name="Accept" value="application/json"/> </http>
</request> </for>
```

```
</session>
<session name="view_grouped" probability="75" type="ts_http"> <thinktime value="10" random= ↩
    "true"/> <request>
<http method="GET" url="/api/_design/default/_view/dates?group_level=1"
>
<http_header name="Accept" value="application/json"/> </http>
</request> <thinktime value="10" random="true"/> <request>
<http method="GET" url="/api/_design/default/_view/dates?group_level=2&amp;stale=ok"
>
<http_header name="Accept" value="application/json"/> </http>
</request> </session>
</sessions> </tsung>
```

Let's walk through some parts of this configuration file. First, the clients element:

```
<!-- Client side setup --> <clients>
<client host="test-a" weight="1" maxusers="10000" cpu="4"/>
<client host="test-b" weight="1" maxusers="10000" cpu="4"/> </clients>
```

This clients element contains a list of clients from which tests may be launched. The more clients, the greater the simulated load that can be generated. Each client needs to be configured using its local hostname or IP address using the host attribute. The weight attribute assigns a relative weight to the client since some clients may be faster and able to start more sessions than other clients. The maxusers attribute defines a maximum number of users to simulate on this client. The cpu attribute declares how many Erlang virtual machines Tsung should use and should be the same as the number of CPUs that are available to the client.

The servers element:

```
<!-- Server side setup --> <servers>
<server host="couch-proxy" port="80" type="tcp"/> </servers>
```

The servers element contains a list of servers to be tested. Each server needs to be configured using its local hostname or IP address using the host attribute. The port attribute indicates the TCP/IP port number to use. The type attribute can either be tcp or udp. Since HTTP uses TCP, we're using tcp as the value here.

The load element:

```
<!-- Load setup --> <load>
<arrivalphase phase="1" duration="5" unit="minute"> <users arrivalrate="200" unit="second"> ↩
    </users>
</arrivalphase> </load>
```

The load element contains a list of arrivalphase elements, each simulating various types of load. The arrivalphase element's phase attribute represents the sequential number of the arrival phase. Here we are only defining one arrival phase. The duration attribute defines how long the arrival phase should last and the unit attribute defines the unit by which to measure the duration. Possible values for the unit element are second, minute, or hour.

Within the arrivalphase element is a users element. The arrivalrate attribute of the users element defines the number of arrivals within the timeframe defined by the unit element. Possible values for the unit element are second, minute, or hour. Here we are telling Tsung to start 200 “arrivals” every second for 5 minutes.

The sessions element:

```
<!-- Sessions setup --> <sessions>
... </sessions>
```

The sessions element contains a list of session elements. These each represent user sessions which may be simulated. We can define multiple sessions and each can have its own probability, but the total probability of all sessions must add up to 100. Let's take a look at each session individually.

The session element with the name attribute value of post_get:

```
<session name="post_get" probability="2.5" type="ts_http"> ...
</session>
```

This session element contains a name attribute with the value of post_get. This name will be used in reports to identify the session. The session element's probability attribute indicates the percent probability of this session being used for any given user. Remember, the total probability of all sessions must add up to 100. The session element's type attribute can be either ts_http, ts_jabber, or ts_mysql. Since we're using HTTP, the type is ts_http.

The thinktime element:

```
<thinktime value="10" random="true"/>[source,xml]
```

The thinktime element defines an amount of time to wait, or "think", before continuing. This is helpful when trying to more realistically simulate load. The thinktime element's value attribute is the amount of time, in seconds, to wait. Setting the thinktime ele- ment's random attribute to a value of true tells Tsung to randomize the wait time, using the value attribute's value as the mean.

The setdynvars elements:

```
<setdynvars sourcetype="random_number" start="2008" end="2011"> <var name="yyyy"/>
</setdynvars> <setdynvars sourcetype="random_number" start="10" end="12">
<var name="mm"/> </setdynvars> <setdynvars sourcetype="random_number" start="10" end="28">
<var name="dd"/> </setdynvars>
```

Each of the setdynvars elements sets a dynamic variable. The sourcetype attribute value of random_number tells Tsung to generate a random number. The start and end attributes indicate the starting and ending values, respectively, to use when generating the random number. The nested var element actually instantiates the variable, using the variable name defined in the name attribute. Here we are generating random year, month, and day values which we will use later in the session.

A request element:

```
<request subst="true"> <match do="abort" when="nomatch">201 Created</match> <dyn_variable  ↩
    name="id" jsonpath="$.id"/> <dyn_variable name="rev" jsonpath="$.rev"/> <http
>
method="POST" url="/api" content_type="application/json" contents="{
&quot;date&quot;:[ &quot;%%_yyyy%%&quot;, &quot;%%_mm%%&quot;, &quot;%%_dd%%&quot;
] }"
<http_header name="Accept" value="application/json"/> </http>
</request>
```

A request element defines a request to be made as part of the session. Since we'll be using the dynamic variables defined earlier, we need set the request element's subst attribute's value to true. This tells Tsung to substitute variables for their values, when encountered.

The match element tells Tsung to "match" on a certain condition. The do attribute value of abort tells Tsung to abort the session if the match condition is true. Possible values for the do attribute are continue, log, abort, restart, or loop. The "when" attribute can either be match or nomatch. The text of the match element is the text to match or not match on. In this case, if the text 201 Created is not found in the response (i.e., the document was not created) then we abort the session.

The two dyn_variable elements define dynamic variables that will be based on the server's response. The name attribute defines the name of the variable to use. Tsung allows matching using a limited subset of JSONPath (XPath for JSON), using the jsonpath attribute. These two variables will contain the ID and revision of the created document (once the response has been received).

The http element initiates an HTTP request. The method attribute specifies the HTTP method to use for the request (e.g., GET, POST, PUT, DELETE). The url attribute specifies the URL to which to make the request. This can be relative to the host set up earlier in the servers element, or a full URL. The content_type attribute specifies the value of the Content-Type HTTP header. The contents attribute specifies the contents of a POST or PUT request body. Here we are using a JSON object as the request body. The JSON object contains one field, date, with its value being an array of year, month, and day values (using the random dynamic variables created earlier).

A "for" element:

```
<for from="0" to="9" incr="1" var="x"> <thinktime value="10" random="true"/> <request subst ↩
    ="true">
<match do="abort" when="nomatch">304 Not Modified</match> <http method="GET" url="/api/%% ↩
    _id%%">
```

```
<http_header name="If-None-Match" value="&quot;%%_rev%%&quot;"/>
<http_header name="Accept" value="application/json"/> </http>
</request> </for>
```

A "for" element will tell Tsung to repeat the enclosed directives a specified number of times. Here we are using a from value of 0, a to value of 9, an incr value of 1, and using a var (variable) with a name of x. This means that the variable x will start out with the value of 0, increment by 1 in each iteration, and the loop will stop when x has reached the value of 9.

The contained thinktime, request, match, and http elements should look familiar. Within the http element, we'll see two http_header elements. As we may have guessed, these specify the name and value of HTTP headers to send as part of the request.

The If-None-Match HTTP header allows us to use conditional caching and the Accept header tells CouchDB that our client can handle content of type application/json.

The remaining sessions in the configuration file should be self-explanatory.

5.6.3 Running Tsung

First, we need to create the view that is used in the above configuration file. This is simply a view of dates (as an array of year, month, and day) from our documents:

```
curl -X PUT http://couch-proxy.example.com/api/_design/default -d
'{
"language": "javascript", "views": {
"dates": { "map":
"function(doc) { if (doc.date) {
} }",
emit(doc.date);
}
"reduce": "_count"
}'
}
```

The response:

```
{ }
"ok":true, "id":"_design/default", "rev":"1-edb41165ec8e4839dd7918e88e2125fa"
```

Start Tsung, telling it to use the above configuration file:

```
tsung -f ~/http_distributed_couch_proxy.xml start
```

Note that Tsung will wait for all sessions to complete before finishing, even if it takes longer than the duration of all phases. Tsung will let us know what directory it has logged to, for example:

```
"Log directory is: /home/bradley-holt/.tsung/log/20110221-23:26"
```

Change into the log directory and generate the HTML and graph reports using the tsung_stats.pl script package with Tsung:

```
/usr/lib/tsung/bin/tsung_stats.pl
```

If everything works correctly, a report.html file will be created in this same directory. Open this report and we will see several statistics and graphs. Under the statistics reports, the main statistics table shows the highest 10 second mean, lowest 10 second mean, highest rate, mean, and count for each part of the HTTP connection.

Tsung allows us to group requests into transactions. A transaction might be useful when testing an HTML page as we could group the requests for the HTML and all related assets (e.g., JavaScript, CSS, and images) into one transaction. We have not done this here, so our transactions statistics table will be empty. The network through- put table lets us see the size of the network traffic received and sent.

There are also several graphs reports available. For all graphs, the x-axis represents a progression of time throughout the test. The first graph represents mean transaction response time. The y-axis for this graph represents the mean number of milliseconds that the transaction response took during a given moment in the test.

5.7 Identifying Bottlenecks

Based on the results of the above tests, we can attempt to make a few conclusions. First, some analysis:

- The CPU utilization percentages on the read-only slave nodes, the write-only master node, and the proxy server are all quite low. It appears that none of these nodes are CPU bound.
- The free memory amounts on the read-only slave nodes, the write-only master node, and the proxy server never drops critically low. It looks like none of the nodes ever run out of memory, so excessive swapping should not be an issue.
- The server load averages on the read-only slave nodes and the write-only master node are reasonable.
- The server load average on the proxy server is quite high.

Based on this analysis, we might conclude that the proxy server is a potential bottleneck in our system. If we look back at the counter statistics, we'll see that the maximum number of connections reached was 2797. However, the maximum number of connections allowed to each read-only node was 4. With three read-only nodes, this gives us a total of 12 maximum connections to the backend CouchDB nodes. The write-only node did not have a limit, but our test scenarios were read-heavy. It appears that the proxy server is effectively queuing requests for the backend CouchDB nodes, which could account for the high server load.

Based on the above hypothesis, adding more read-only CouchDB nodes might actually lessen the load on the proxy server.