The Antepenultimate CouchDB Reference Card

{ "compiled-by" : "Jan-Piet Mens", "on" : "2010-05-11", "_rev" : "15-b3d3ec5ec1ceae407a20a7ae4b8f5da6" }

CouchDB

Apache CouchDB (`couchdb.apache.org`) is a document-oriented database that provides a RESTful JSON API which can be access from any client that speaks HTTP. CouchDB can be queried and indexed in a MapReduce fashion using JavaScript, and it offers incremental replication with bi-directional conflict detection and resolution.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format, easy for humans to read and write and easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. (`json.org`). The following file `jj.json` contains an example:

```
{  "_id" : "jane",                                          unique doc id
   "_rev" : "18-2f990995d34d1f1fc65544bc6411420b",          revision
   "surname" : "Jolie"
   "active" : true,
   "hobbies" : [ "dance", "music", "books" ],
   "height" : 168 }
```

CouchDB documents all have an *_id* field which is unique, and a *_rev* field containing a documents last revision. (In the examples below, revision numbers are truncated for brevity.) The *_id* can be any string and is often a UUID, whereas the revision number changes on each update to the document.

Curl

curl, a command-line tool for transferring data (`curl.haxx.se`), is excellent for experimenting with CouchDB. Use *curl*'s `-X` switch to specify one of the methods (`GET`, `PUT`, `DELETE`, `COPY`, `POST` – default is `GET`) to describe the desired action to be performed on a resource. In this document we show a portion of the URL only for brevity. So, when we specify say, `/db/doc`, we typically mean something like this:

```
curl -s http://127.0.0.1:5984/db/doc                        silent
curl -v http://user:password@127.0.0.1:5984/db/doc          verbose
```

curl expects data after the `-d` option, either as single string on the command line (ensuring the shell doesn't interpret its content) or from a file such as `jj.json` above. For example, to create a CouchDB document as in the above example, use either of the following:

```
curl -X PUT http://127.0.0.1:5984/db/jane -d @jj.json                data from file
curl -X PUT http://127.0.0.1:5984/db/jane -d '{"name":"Jane",\       data right here
             "surname":"Jolie","height":168, ...}'
```

Database operations

The commands in this section deal with CouchDB databases as a whole, i.e. creating, querying information, and deleting them.

list databases

```
GET /_all_dbs
["rep1","db","names"]
```

create database

```
PUT /db                                                     lowercase only
```

destroy database

```
DELETE /db
```

database information

```
GET /db
{
   "compact_running" : false,
   "doc_count" : 793,
   "db_name" : "jsconf",
   "purge_seq" : 0,
   "doc_del_count" : 0,
   "disk_format_version" : 5,
   "update_seq" : 793,
   "instance_start_time" : "1271509153975346",
   "disk_size" : 5771364
}
```

change notifications

```
GET /db/_changes
         &style=main_only                                   default
         &style=all_docs                                    more rev info
```

changes since seq

```
GET /db/_changes?since=87
         &timeout=milliseconds                              default 60s
```

changes: long polling

```
GET /db/_changes?feed=longpoll
```

changes: continuous feed

```
GET /db/_changes?feed=continuous
         &heartbeat=milliseconds                            keep alive
```

Document operations

list all documents

```
GET /db/_all_docs
        &descending=true          change sort order
        &limit=3                  limit count
        &include_docs=true        full documents
```

get single document

```
GET /db/doc
        &revs=true                show revisions
        &rev=nnnnn                get revision
```

fetch documents by keys

```
POST /db/_all_docs \              keys are
   -d '{"keys":["jane","jkp"]}'   document ids
```

create document with id

```
PUT /db/88 -d '{"name":"Jane"}'   id part of URI
{"ok":true,"id":"88","rev":"1-c289"}
```

create document

```
POST /db -d '{"name":"Jesse"}'    doc's id generated
{"ok":true,"id":"bd4b48f65792a45224ad406ff00013f4",
       "rev":"1-a3ea"}
```

copy document

```
COPY /db/88 -H "Destination: jane"
{"id":"jane","rev":"1-c2890"}
```

update document

```
PUT /db/doc                       requires _rev
PUT /db/88 -d '{"name":"JanE", "_rev":"1-c289"}'
{"ok":true,"id":"88","rev":"2-a524"}
```

delete document

```
DELETE /db/doc                    requires _rev
DELETE /db/88?rev=2-a524
```

bulk operations

CouchDB provides a bulk insert/update feature. To use this, you make a POST request to the URI `/_bulk_docs`, with the request body being a JSON document containing a list of new documents to be inserted or updated. Consider the following JSON file `bulk.json`:

```
{
  "all_or_nothing":false,
  "docs": [
    {"_id": "annie", "name": "Annie"},
    { "name": "Suzie"},
  ]
}
```

We now POST this data to CouchDB. Documents without an explicit `_id` get one generated by CouchDB.

```
POST /db/_bulk_docs -d @bulk.json
[                                 output reformatted
 { "id":"annie", "rev":"1-581f",}
 { "id":"bd4b...5a","rev":"1-3be6" },   generated id
]
```

To update existing documents, include the `_rev` field to the current document's revision, and to delete documents in bulk include both the revision and a boolean `_deleted` set true. Individual updates may fail (e.g. because they don't pass validation), but you can get an all-or-nothing transaction-like feeling by adding a boolean `"all_or_nothing":true` field alongside the `docs` array.

add attachment

```
PUT /db/doc/image.jpg?rev=1-c289 \      requires _rev
    -H 'Content-Type: image/jpeg' \
    --data-binary @filename.jpg
```

inline attachments

Add any number of inline attachments to the document when `PUT`ting it into the database using the document's *_attachments* field.

```
$ cat person.json
{  "name" : "Jo",
   "_attachments" : {
      "portrait.jpg" : {
         "content_type" : "image/jpeg",
         "data" : "..."               base64, no white
                                      space
      },
      "payroll.txt" : {
         "content_type" : "text/plain",
         "data" : "..."               base64, no white
                                      space
      } } }
```

list attachments

```
GET /db/doc
{
   "_id" : "doc",
   ...
   "_attachments" : {
      "image.jpg" : {
         "revpos" : 2,
         "length" : 14102,
         "content_type" : "image/jpeg",
         "stub" : true
      }
   }
}
```

This is just the metadata (`stub=true`). Get the base64 encoded `data` using the `?attachments=true` option.

retrieve attachments

```
GET /db/doc/image.jpg
```

delete attachment

```
DELETE /db/doc/attachmentname?rev=...   requires _rev
```

Validation

As documents written to disk, they can be validated dynamically by JavaScript functions. When the document passes all the formula validation criteria, the update is allowed to continue, otherwise the update is aborted and the client gets an error response. Both the user's credentials and the updated document are given as inputs to the validation formula, and can be used to implement custom security models by validating a user's permissions to update a document. There is function one per design document, but there can be many design documents, in which case the validation functions are invoked in an undefined order. Add an attribute `validate_doc_update` containing the validation function.

```
{
   "_id": "_design/aa",
   "validate_doc_update": "function(newDoc,oldDoc,userCtx) {...}"
}
```

```
function(newDoc, oldDoc, userCtx) {
  if (!newDoc.type) {
    throw {"forbidden":"Documents need a type."};
  }
}
```

Update handlers

These [1] allow updating a database document without the usual `GET`-modify-`POST` cycle. The handler is passed the server's current version of the document.

```
function(doc, req) {
  var field = req.query.field;
  doc[field] = req.query.value;
  doc.countr = (doc.countr) ? doc.countr + 1 : 1;
  return [doc, "Thanks."];
}
```

```
PUT /db/_design/app/_update/setfield/doc?field=name&value=JP
Thanks.
```

Views

Views are the method of aggregating and reporting on the documents in a CouchDB database, and they are built on-demand. Views are built dynamically and don't affect the underlying documents; you can have as many different view representations of the same data as you like. View definitions are strictly virtual and only display the documents from the current database instance, making them separate from the data they display and compatible with replication. CouchDB views are defined in design documents and can replicate like regular documents.

[1] `http://wiki.apache.org/couchdb/Document_Update_Handlers`

Map/Reduce

A map function takes a single document as input, and returns an array of key/value pairs as output, whereby an empty array is possible. For a given input, the map function must produce the same output, so the result can't vary according to the time of day, or any other factor. Map functions in CouchDB use the `emit()` function to send each of the key-value pairs that make up the array back to the server.
A reduce function takes an array as input, and it returns a single value (which may be a complex type such as an array or hash) as output. Depending on the amount of data CouchDB has to process, the reduce operation could be broken up into smaller chunks by the server. When this happens, the reduce function is invoked with *rereduce* set to *false*. The reduce function's results are then amassed by CouchDB, and the reduce function is finally called with *rereduce* set to *true*, with an array of values in *values*. So, if *rereduce* is *false*, the *keys* and *values* arguments are a list of keys/values for each row emitted by *map* respectively. If, on the other hand, *rereduce* is *true*, *keys* will be *null*, and the *values* argument contains an array of results produced by the previous invocation(s) of the reduce function. Whew. The reduce function must produce the same result if the input array is randomly shuffled.

map

Get a list of all documents of type *card* for a view and issue the name of the card:

```
function (doc) {
    if (doc['type'] == "card") {
        emit(doc.name, 1);
    }
}
```

Output a list of all tags in each document, where tags is a JSON array such as `["music","books","food"]`.

```
function(doc) {
    if (doc.tags && doc.tags.length > 0) {
        for (var i = 0; i < doc.tags.length; i++) {
            emit(doc.tags[i], 1);
        }
    }
}
```

map/reduce

Determine which and how many documents have attachments. (Use this view with `?group=` – see below.)

```
function(doc) {                          map
    if (doc._attachments) {
    emit("with file", 1);
    } else {
    emit("no files", 1);
    }
}
function(keys, values) {                 reduce
```

```
    return sum(values);
}
```

query view definition

```
GET /db/_design/viewname
```

query view information

```
GET /db/_design/viewname/_info
```

query view content

```
GET /db/_design/myv/_view/cards
{"total_rows":2,"offset":0,"rows":[
{"id":"jane","key":"Jane","value":1},
...
]}
```

querying options

```
GET .../cards?key='"Jane"'                    quotes! valid JSON
              ?startkey='"Jo%20Guest"'
              ?endkey=...
              ?limit=nnn
              ?descending=true
              ?skip=4
              ?group=true                     red. to distinct keys
              ?include_docs=true
```

temporary views

One-off queries (use for development only) can be `POST`ed to the special `_temp_view`. For example, to log (to the `couch.log` file) a list of all documents:

```
POST /db/_temp_view \
     -d '{"map":"function(doc){log(doc);}"}'
```

Shows

Use show functions to output data in any way. These functions are stored in your design document, under the *shows* key.

```
{
   "_id": "_design/myv",
   "shows": {
     "hello": "function(doc,req) { return 'hello world';}",
     "name": "function(doc,req) { if (doc) { return doc.surname; }}",
     "rq": "function(doc,req) { return 'NAME=' + req.query.name;}"
   }
}
```

simple show

```
GET /db/_design/myv/_show/hello
hello world
```

show with document id

```
GET /db/_design/myv/_show/name/jane
Jolie
```

show with parameters

```
GET /db/_design/myv/_show/rq
NAME=undefined
GET /db/_design/myv/_show/rq?name=Somebody
NAME=Somebody
```

Lists

List functions are a mechanism for iterating over rows in a view to produce output. CouchDB list functions are typically used to generate alternate formats for output (RSS, XML, HTML, etc.).

```
{
   "_id": "_design/myv",
   "lists": {
       "foo": "function(head,req) {
                  var row;
                  while (row = getRow()) {
                    send('Name='+row.key + '\\012');
                  }
                }"
   }
}
```

list

```
GET /db/_design/myv/_list/listname/viewname
GET /db/_design/myv/_list/foo/cards
Name=Jane
Name=Jo
```

list options

```
GET ... _list/foo/cards?key='"Jane"'    quotes!
                       ?descending=true
                       ?startkey=...
                       ?limit=10
```

Replication

Replication in CouchDB is a one-off operation where you send an HTTP request to CouchDB that includes a *source* and a *target* database that are to be replicated. CouchDB sends changes from the source to the target and replication is thus complete. Both *source* and *target* are either simple database names (e.g. `db`) or URLs to remote CouchDB databases.

one-off replication

```
POST /_replicate -d \
  '{"source":"http://example.com/abook", \
     "target":"mydb"}'
{
 "ok" : true,                              Yeah!
 "history" : [                             truncated
  {
   "docs_read" : 10,
   "doc_write_failures" : 0,
   "start_time" : "Tue, 20 Apr 2010 10:17:01 GMT",
   "docs_written" : 10,
  ...
}
```

continuous replication

Add a boolean element `"continuous":true` to the request shown above to specify that you want CouchDB to initiate continous replication (Note that at the time of this writing continous replication does not pick up after a server restart)

create target

Since version 0.11 CouchDB can automatically create the target database. In order to do so, add a `"create_target":true` to the replication request.

individual documents

Since 0.11 you can specify a list of document ids to be replicated from the source to the target, useful when you want only a subset of documents replicated (e.g. design only). Documents deleted on the source are not replicated.

```
POST /_replicate \
   -d '{"source":"http://example.de:5984/db1",\
        "target":"dbhere",\
        "doc_ids":["u235","wgx","jane"]}'
```

filters

In 0.11 you can specify a filter for the `_changes` feed used by the replicator.

```
POST /_replicate \
   -d '{"source":"http://example.de:5984/db1",\
        "target":"dbhere",\
        "filter":"myv/avatar"}'
```

The filter is a function contained in the specified design document[2]:

```
function(doc, req) {
    if (doc.type == 'avatar') {
            return true;
    }
    return false;
}
```

Status

welcome

```
GET /
{"couchdb":"Welcome","version":"0.11.0"}
```

configuration data

```
GET /_config
```

statistics

```
GET /_stats
```

active tasks

```
GET /_active_tasks
```

UUIDs

```
GET /_uuids?count=1
{"uuids":["bd4b48f65792a45224ad406ff0002647"]}
```

[2]More information at `http://mens.de/:/52`

[3]`http://github.com/couchapp/couchapp/`

Utilities

compact database

```
POST /db/_compact
POST /db/_compact/designname
```

Compaction compresses the database file (or the views) by removing unused sections created during updates. Old revisions of documents are also removed from the database though a small amount of meta data is kept for use in conflict replication during replication.

view cleanup

```
POST /db/_view_cleanup
```

When you change a view, old indexes remain on disk. To clean up all outdated view indexes (files named after the MD5 representation of views, that does not exist anymore) you can trigger a view cleanup.

Rewriting

The `_rewrite` URL endpoint on your design documents lets you rewrite any incoming request to a regular CouchDB API URL. The rewrite target in the *to* attribute is relative to the design document it is in. rewriting root is the design document. Here is an array with two rewriting rules in it:

```
{
   "_id": "_design/myv",
   "rewrites": [
     { "from": "/hola",
       "to":   "/_show/hello" },
     { "from": "show/:id",
       "to":   "_show/name/:id" }
   ]
}
```

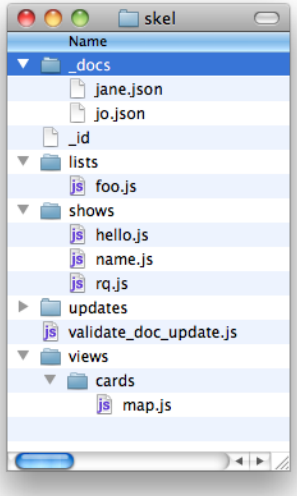
will cause all requests for `/db/_design/myv/_rewrite/asdf` to be directed to `/db/_design/myv/_show/jp`, and requests for `.../_rewrite/show/999` to go to `.../_show/mydoc/999`.

CouchApp

CouchApp[3] is a system for deploying applications into a CouchDB database: "*CouchApp is a set of scripts and a jQuery plugin designed to bring clarity and order to the freedom of CouchDB's document-based approach.* However, it can also be use it to deploy views, shows, lists, etc. into a database. Start off by creating an application using `couchapp generate skel`. Set up a `.couchapprc` file in your `skel` directory containing the URL to your database:

```
{ "env":{ "default":{ "db":"http://example.de:5984/db" } } }
```

You can *push* this structure into your CouchDB by running `couchapp push` from within your `skel` directory.



views, shows, lists Functions are easily edited as *dir*`/`*func*`.js` where *dir* represents a directory named shows, views, etc. and func is the name of a show, view, list, etc.

preload documents You can also "preload" your CouchDB database with documents (inline attachments and all) by creating each as a `.json` file and dropping them into a `_docs` directory in your `skel/`; these are pushed into the database as well.

More relaxation...

The Guide The title says it all. This is the CouchDB book in its entirety, although I do recommend you get the printed version to read on your couch. `http://books.couchdb.org/relax/`

Resty Resty is a tiny script wrapper for curl which provides a simple, concise shell interface for interacting with REST services. It is implemented as functions in your own shell. You can use Resty in pipelines to process data from REST services, and PUT or POST the data right back. You can even pipe the data in and then edit it interactively in your text editor prior to PUT or POST. Highly recommended. `http://github.com/micha/resty` (I've made a small screencat of Resty at `http://mens.de/:/53`)

pudo Pudo reads a directory containing JSON or YAML files and uploads their content as JSON documents to a CouchDB. If a matching *doc*`.attachments` directory is found, all contained files are attached to the document. `http://github.com/jpmens/pudo`