CouchDB Release 1.1 Feature Guide

Abstract

This document provides details on the new features introduced in the CouchDB 1.1 release from the CouchDB 1.0.x release series.

Last document update: 25 Jan 2012 14:44; *Document built*: 21 Feb 2012 20:8.

Table of Contents

1. Upgrading to CouchDB 1.1

You can upgrade your existing CouchDB 1.0.x installation to CouchDB 1.1 without any specific steps or migration. When you run CouchDB 1.1 the existing data and index files will be opened and used as normal.

The first time you run a compaction routine on your database within CouchDB 1.1, the data structure and indexes will be updated to the new version of the CouchDB database format that can only be read by CouchDB 1.1 and later. This step is not reversable. Once the data files have been updated and migrated to the new version the data files will no longer work with a CouchDB 1.0.x release.

Warning

If you want to retain support for openein gthe data files in CouchDB 1.0.x you must back up your data files before performing the upgrade and compaction process.

2. Replicator Database

A database where you `PUT`/`POST` documents to trigger replications and you `DELETE` to cancel ongoing replications. These documents have exactly the same content as the JSON objects we used to `POST` to `_replicate` (fields `source`, `target`, `create_target`, `continuous`, `doc_ids`, `filter`, `query_params`.

Replication documents can have a user defined `_id`. Design documents (and `_local` documents) added to the replicator database are ignored.

The default name of this database is `_replicator`. The name can be changed in the `local.ini` configuration, section `[replicator]`, parameter `db`.

2.1. Basics

Let's say you PUT the following document into _replicator:

```
{
    "_id": "my_rep",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar",
    "create_target":  true
}
```

In the couch log you'll see 2 entries like these:

```
[Thu, 17 Feb 2011 19:43:59 GMT] [info] [<0.291.0>] Document `my_rep` triggered replication `c0ebe9256695ff083347cbf95
[Thu, 17 Feb 2011 19:44:37 GMT] [info] [<0.124.0>] Replication `c0ebe9256695ff083347cbf95f93e280+create_target` finis
```

As soon as the replication is triggered, the document will be updated by CouchDB with 3 new fields:

```
{
    "_id": "my_rep",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar",
    "create_target":  true,
    "_replication_id":  "c0ebe9256695ff083347cbf95f93e280",
    "_replication_state":  "triggered",
    "_replication_state_time":  1297974122
}
```

Special fields set by the replicator start with the prefix `_replication_`.

- `_replication_id`

  The ID internally assigned to the replication. This is also the ID exposed by `/_active_tasks`.

- `_replication_state`

  The current state of the replication.

- `_replication_state_time`

  A Unix timestamp (number of seconds since 1 Jan 1970) that tells us when the current replication state (marked in `_replication_state`) was set.

When the replication finishes, it will update the `_replication_state` field (and `_replication_state_time`) with the value `completed`, so the document will look like:

```
{
    "_id": "my_rep",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar",
    "create_target":  true,
    "_replication_id":  "c0ebe9256695ff083347cbf95f93e280",
    "_replication_state":  "completed",
    "_replication_state_time":  1297974122
}
```

When an error happens during replication, the `_replication_state` field is set to `error` (and `_replication_state` gets updated of course).

When you PUT/POST a document to the `_replicator` database, CouchDB will attempt to start the replication up to 10 times (configurable under `[replicator]`, parameter `max_replication_retry_count`). If it fails on the first attempt, it waits 5 seconds before doing a second attempt. If the second attempt fails, it waits 10 seconds before doing a third attempt. If the third attempt fails, it waits 20 seconds before doing a fourth attempt (each attempt doubles the previous wait period). When an attempt fails, the Couch log will show you something like:

```
[error] [<0.149.0>] Error starting replication `67c1bb92010e7abe35d7d629635f18b6+create_target` (document `my_rep_2`)
```

Note

The `_replication_state` field is only set to `error` when all the attempts were unsuccessful.

There are only 3 possible values for the `_replication_state` field: `triggered`, `completed` and `error`. Continuous replications never get their state set to `completed`.

2.2. Documents describing the same replication

Lets suppose 2 documents are added to the `_replicator` database in the following order:

```
{
    "_id": "doc_A",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar"
}
```

and

```
{
    "_id": "doc_B",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar"
}
```

Both describe exactly the same replication (only their `_ids` differ). In this case document `doc_A` triggers the replication, getting updated by CouchDB with the fields `_replication_state`, `_replication_state_time` and `_replication_id`, just like it was described before. Document `doc_B` however, is only updated with one field, the `_replication_id` so it will look like this:

```
{
    "_id": "doc_B",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar",
    "_replication_id":  "c0ebe9256695ff083347cbf95f93e280"
}
```

While document `doc_A` will look like this:

```
{
    "_id": "doc_A",
    "source":  "http://myserver.com:5984/foo",
    "target":  "bar",
    "_replication_id":  "c0ebe9256695ff083347cbf95f93e280",
    "_replication_state":  "triggered",
    "_replication_state_time":  1297974122
}
```

Note that both document get exactly the same value for the `_replication_id` field. This way you can identify which documents refer to the same replication - you can for example define a view which maps replication IDs to document IDs.

2.3. Canceling replications

To cancel a replication simply `DELETE` the document which triggered the replication. The Couch log will show you an entry like the following:

```
[Thu, 17 Feb 2011 20:16:29 GMT] [info] [<0.125.0>] Stopped replication `c0ebe9256695ff083347cbf95f93e280+continuous+cr
```

Note

You need to `DELETE` the document that triggered the replication. `DELETE`ing another document that describes the same replication but did not trigger it, will not cancel the replication.

2.4. Server restart

When CouchDB is restarted, it checks its `_replicator` database and restarts any replication that is described by a document that either has its `_replication_state` field set to `triggered` or it doesn't have yet the `_replication_state` field set.

Note

Continuous replications always have a `_replication_state` field with the value `triggered`, therefore they're always restarted when CouchDB is restarted.

2.5. Changing the Replicator Database

Imagine your replicator database (default name is _replicator) has the two following documents that represent pull replications from servers A and B:

```
{
    "_id": "rep_from_A",
    "source":  "http://aserver.com:5984/foo",
    "target":  "foo_a",
    "continuous":  true,
    "_replication_id":  "c0ebe9256695ff083347cbf95f93e280",
    "_replication_state":  "triggered",
    "_replication_state_time":  1297971311
}
{
    "_id": "rep_from_B",
    "source":  "http://bserver.com:5984/foo",
    "target":  "foo_b",
    "continuous":  true,
    "_replication_id":  "231bb3cf9d48314eaa8d48a9170570d1",
    "_replication_state":  "triggered",
    "_replication_state_time":  1297974122
}
```

Now without stopping and restarting CouchDB, you change the name of the replicator database to `another_replicator_db`:

```
$ curl -X PUT http://localhost:5984/_config/replicator/db -d '"another_replicator_db"'
"_replicator"
```

As soon as this is done, both pull replications defined before, are stopped. This is explicitly mentioned in CouchDB's log:

```
[Fri, 11 Mar 2011 07:44:20 GMT] [info] [<0.104.0>] Stopping all ongoing replications because the replicator database w
[Fri, 11 Mar 2011 07:44:20 GMT] [info] [<0.127.0>] 127.0.0.1 - - PUT /_config/replicator/db 200
```

Imagine now you add a replication document to the new replicator database named `another_replicator_db`:

```
{
    "_id": "rep_from_X",
    "source":  "http://xserver.com:5984/foo",
    "target":  "foo_x",
    "continuous":  true
}
```

From now own you have a single replication going on in your system: a pull replication pulling from server X. Now you change back the replicator database to the original one `_replicator`:

```
$ curl -X PUT http://localhost:5984/_config/replicator/db -d '"_replicator"'
"another_replicator_db"
```

Immediately after this operation, the replication pulling from server X will be stopped and the replications defined in the _replicator database (pulling from servers A and B) will be resumed.

Changing again the replicator database to `another_replicator_db` will stop the pull replications pulling from servers A and B, and resume the pull replication pulling from server X.

2.6. Replicating the replicator database

Imagine you have in server C a replicator database with the two following pull replication documents in it:

```
{
     "_id": "rep_from_A",
     "source":  "http://aserver.com:5984/foo",
     "target":  "foo_a",
     "continuous":  true,
     "_replication_id":  "c0ebe9256695ff083347cbf95f93e280",
     "_replication_state":  "triggered",
     "_replication_state_time":  1297971311
}
{
     "_id": "rep_from_B",
     "source":  "http://bserver.com:5984/foo",
     "target":  "foo_b",
     "continuous":  true,
     "_replication_id":  "231bb3cf9d48314eaa8d48a9170570d1",
     "_replication_state":  "triggered",
     "_replication_state_time":  1297974122
}
```

Now you would like to have the same pull replications going on in server D, that is, you would like to have server D pull replicating from servers A and B. You have two options:

- Explicitly add two documents to server's D replicator database
- Replicate server's C replicator database into server's D replicator database

Both alternatives accomplish exactly the same goal.

2.7. Delegations

Replication documents can have a custom `user_ctx` property. This property defines the user context under which a replication runs. For the old way of triggering replications (POSTing to `/_replicate/`), this property was not needed (it didn't exist in fact) - this is because at the moment of triggering the replication it has information about the authenticated user. With the replicator database, since it's a regular database, the information about the authenticated user is only present at the moment the replication document is written to the database - the replicator database implementation is like a _changes feed consumer (with `?include_docs=true`) that reacts to what was written to the replicator database - in fact this feature could be implemented with an external script/program. This implementation detail implies that for non admin users, a `user_ctx` property, containing the user's name and a subset of his/her roles, must be defined in the replication document. This is ensured by the document update validation function present in the default design document of the replicator database. This validation function also ensure that a non admin user can set a user name property in the `user_ctx` property that doesn't match his/her own name (same principle applies for the roles).

For admins, the `user_ctx` property is optional, and if it's missing it defaults to a user context with name null and an empty list of roles - this mean design documents will not be written to local targets. If writing design documents to local targets is desired, the a user context with the roles `_admin` must be set explicitly.

Also, for admins the `user_ctx` property can be used to trigger a replication on behalf of another user. This is the user context that will be passed to local target database document validation functions.

Note

The `user_ctx` property only has effect for local endpoints.

Example delegated replication document:

```
{
     "_id": "my_rep",
     "source":  "http://bserver.com:5984/foo",
     "target":  "bar",
     "continuous":  true,
     "user_ctx": {
          "name": "joe",
          "roles": ["erlanger", "researcher"]
     }
}
```

As stated before, for admins the user_ctx property is optional, while for regular (non admin) users it's mandatory. When the roles property of `user_ctx` is missing, it defaults to the empty list `[ ]`.

3. Native SSL Support

CouchDB 1.1 supports SSL natively. All your secure connection needs can now be served without the need set and maintain a separate proxy server that handles SSL.

SSL setup can be tricky, but the configuration in CouchDB was designed to be as easy as possible. All you need is two files; a certificate and a private key. If you bought an official SSL certificate from a certificate authority, both should be in your possession already.

If you just want to try this out and don't want to pay anything upfront, you can create a self-signed certificate. Everything will work the same, but clients will get a warning about an insecure certificate.

You will need the OpenSSL command line tool installed. It probably already is.

```
shell> mkdir cert && cd cert
shell> openssl genrsa > privkey.pem
shell> openssl req -new -x509 -key privkey.pem -out mycert.pem -days 1095
shell> ls
mycert.pem privkey.pem
```

Now, you need to edit CouchDB's configuration, either by editing your `local.ini` file or using the `/_config` API calls or the configuration screen in Futon. Here is what you need to do in `local.ini`, you can infer what needs doing in the other places.

Be sure to make these edits. Under `[daemons]` you should see:

```
; enable SSL support by uncommenting the following line and supply the PEM's below.
; the default ssl port CouchDB listens on is 6984
;httpsd = {couch_httpd, start_link, [https]}
```

Here uncomment the last line:

```
httpsd = {couch_httpd, start_link, [https]}
```

Next, under `[ssl]` you will see:

```
;cert_file = /full/path/to/server_cert.pem
;key_file = /full/path/to/server_key.pem
```

Uncomment and adjust the paths so it matches your system's paths:

```
cert_file = /home/jan/cert/mycert.pem
key_file = /home/jan/cert/privkey.pem
```

For more information please read http://www.openssl.org/docs/HOWTO/certificates.txt.

Now start (or restart) CouchDB. You should be able to connect to it using HTTPS on port 6984:

```
shell> curl https://127.0.0.1:6984/
curl: (60) SSL certificate problem, verify that the CA cert is OK. Details:
error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed
More details here: http://curl.haxx.se/docs/sslcerts.html

curl performs SSL certificate verification by default, using a "bundle"
of Certificate Authority (CA) public keys (CA certs). If the default
bundle file isn't adequate, you can specify an alternate file
using the --cacert option.
If this HTTPS server uses a certificate signed by a CA represented in
the bundle, the certificate verification probably failed due to a
problem with the certificate (it might be expired, or the name might
not match the domain name in the URL).
If you'd like to turn off curl's verification of the certificate, use
the -k (or --insecure) option.
```

Oh no what happened?! — Remember, clients will notify their users that your certificate is self signed. **curl** is the client in this case and it notifies you. Luckily you trust yourself (don't you?) and you can specify the `-k` option as the message reads:

```
shell> curl -k https://127.0.0.1:6984/
{"couchdb":"Welcome","version":"1.1.0"}
```

All done.

4. HTTP Range Requests

HTTP allows you to specify byte ranges for requests. This allows the implementation of resumable downloads and skippable audio and video streams alike. Now this is available for all attachments inside CouchDB.

This is just a real quick run through how this looks under the hood. Usually, you will have larger binary files to serve from CouchDB, like MP3s and videos, but to make things a little more obvious, I use a text file here (Note that I use the `application/octet-stream` Content-Type instead of `text/plain`).

```
shell> cat file.txt
My hovercraft is full of eels!
```

Now lets store this text file as an attachment in CouchDB. First, we create a database:

```
shell> curl -X PUT http://127.0.0.1:5984/test
{"ok":true}
```

Then we create a new document and the file attachment in one go:

```
shell> curl -X PUT http://127.0.0.1:5984/test/doc/file.txt -H "Content-Type: application/octet-stream" -d@file.txt
{"ok":true,"id":"doc","rev":"1-287a28fa680ae0c7fb4729bf0c6e0cf2"}
```

Now we can request the whole file easily:

```
shell> curl -X GET http://127.0.0.1:5984/test/doc/file.txt
My hovercraft is full of eels!
```

But say we only want the first 13 bytes:

```
shell> curl -X GET http://127.0.0.1:5984/test/doc/file.txt -H "Range: bytes=0-12"
My hovercraft
```

HTTP supports many ways to specify single and even multiple byte rangers. Read all about it in RFC 2616.

Note

Databases that have been created with CouchDB 1.0.2 or earlier will support range requests in 1.1.0, but they are using a less-optimal algorithm. If you plan to make heavy use of this feature, make sure

to compact your database with CouchDB 1.1.0 to take advantage of a better algorithm to find byte ranges.

5. HTTP Proxying

The HTTP proxy feature makes it easy to map and redirect different content through your CouchDB URL. The proxy works by mapping a pathname and passing all content after that prefix through to the configured proxy address.

Configuration of the proxy redirect is handled through the `[httpd_global_handlers]` section of the CouchDB configuration file (typically `local.ini`). The format is:

```
[httpd_global_handlers]
PREFIX = {couch_httpd_proxy, handle_proxy_req, <<"DESTINATION">>}
```

Where:

- `PREFIX`

  Is the string that will be matched. The string can be any valid qualifier, although to ensure that existing database names are not overridden by a proxy configuration, you can use an underscore prefix.

- `DESTINATION`

  The fully-qualified URL to which the request should be sent. The destination must include the `http` prefix. The content is used verbatim in the original request, so you can also forward to servers on different ports and to specific paths on the target host.

The proxy process then translates requests of the form:

```
http://couchdb:5984/PREFIX/path
```

To:

```
DESTINATION/path
```

Note

Everything after `PREFIX` including the required forward slash will be appended to the `DESTINATION`.

The response is then communicated back to the original client.

For example, the following configuration:

```
_google = {couch_httpd_proxy, handle_proxy_req, <<"http://www.google.com">>}
```

Would forward all requests for `http://couchdb:5984/_google` to the Google website.

The service can also be used to forward to related CouchDB services, such as Lucene:

```
[httpd_global_handlers]
_fti = {couch_httpd_proxy, handle_proxy_req, <<"http://127.0.0.1:5985">>}
```

Note

The proxy service is basic. If the request is not identified by the `DESTINATION`, or the remainder of the `PATH` specification is incomplete, the original request URL is interpreted as if the `PREFIX` component of that URL does not exist.

For example, requesting `http://couchdb:5984/_intranet/media` when `/media` on the proxy destination does not exist, will cause the request URL to be interpreted as `http://`

> `couchdb:5984/media`. Care should be taken to ensure that both requested URLs and destination URLs are able to cope

6. Added CommonJS support to map functions

We didn't have CommonJS require in map functions because the current CommonJS implementation is scoped to the whole design doc, and giving views access to load code from anywhere in the design doc would mean we'd have to blow away your view index any time you changed anything. Having to rebuild views from scratch just because you changed some CSS or a show function isn't fun, so we avoided the issue by keeping CommonJS require out of map and reduce altogether.

The solution we came up with is to allow CommonJS inside map and reduce funs, but only of libraries that are stored inside the views part of the design doc.

So you could continue to access CommonJS code in design_doc.foo, from your list functions etc, but we'd add the ability to require CommonJS modules within map and reduce, but only from design_doc.views.lib

There's no worry here about namespace collisions, as Couch just plucks `views.*.map` and `views.*.reduce` out of the design doc. So you could have a view called `lib` if you wanted, and still have CommonJS stored in `views.lib.sha1` and `views.lib.stemmer` if you wanted.

We simplified the implementation by enforcing that CommonJS modules to be used in map functions be stored in views.lib.

A sample design doc (taken from the test suite in Futon) is below:

```
{
   "views" : {
      "lib" : {
         "baz" : "exports.baz = 'bam';",
         "foo" : {
            "zoom" : "exports.zoom = 'yeah';",
            "boom" : "exports.boom = 'ok';",
            "foo" : "exports.foo = 'bar';"
         }
      },
      "commonjs" : {
         "map" : "function(doc) { emit(null, require('views/lib/foo/boom').boom)}"
      }
   },
   "_id" : "_design/test"
}
```

The `require()` statement is relative to the design document, but anything loaded form outside of `views/lib` will fail.

7. More granular ETag support for views

ETags have been assigned to a map/reduce group (the collection of views in a single design document). Any change to any of the indexes for those views would generate a new ETag for all view URL's in a single design doc, even if that specific view's results had not changed.

In CouchDB 1.1 each `_view` URL has it's own ETag which only gets updated when changes are made to the database that effect that index. If the index for that specific view does not change, that view keeps the original ETag head (therefore sending back 304 Not Modified more often).

8. Added built-in filters for `_changes`: `_doc_ids` and `_design`.

The `_changes` feed can now be used to watch changes to specific document ID's or the list of `_design` documents in a database. If the `filters` parameter is set to `_doc_ids` a list of doc IDs can be passed in the "doc_ids" as a JSON array.

9. Allow wildcards in vhosts definitions

Similar to the rewrites section of a `_design` document, the new `vhosts` system uses variables in the form of :varname or wildcards in the form of asterisks. The variable results can be output into the resulting path as they are in the rewriter.

10. OS Daemons

CouchDB now supports starting external processes. The support is simple and enables CouchDB to start each configured OS daemon. If the daemon stops at any point, CouchDB will restart it (with protection to ensure regularly failing daemons are not repeatedly restarted).

The daemon starting process is one-to-one; for each each configured daemon in the configuration file, CouchDB will start exactly one instance. If you need to run multiple instances, then you must create separate individual configurations. Daemons are configured within the `[os_daemons]` section of your configuration file (`local.ini`). The format of each configured daemon is:

```
NAME = PATH ARGS
```

Where `NAME` is an arbitrary (and unique) name to identify the daemon; `PATH` is the full path to the daemon to be executed; `ARGS` are any required arguments to the daemon.

For example:

```
[os_daemons]
basic_responder = /usr/local/bin/responsder.js
```

There is no interactivity between CouchDB and the running process, but you can use the OS Daemons service to create new HTTP servers and responders and then use the new proxy service to redirect requests and output to the CouchDB managed service. For more information on proxying, see Section 5, “HTTP Proxying”. For further background on the OS Daemon service, see CouchDB Externals API

11. Stale views and `update_after`

Currently a view request can include the `stale=ok` query argument, which allows the contents of a stale view index to be used to produce the view output. In order to trigger a build of the outdated view index, a second view request must be made.

To simplify this process, the `update_after` value can be supplied to the `stale` query argument. This triggers a rebuild of the view index after the results of the view have been retrieved.

12. Socket Options Configuration Setting

The socket options for the listening socket in CouchDB can now be set within the CouchDB configuration file. The setting should be added to the `[httpd]` section of the file using the option name `socket_options`. The specification is as a list of tuples. For example:

```
[httpd]
socket_options = [{recbuf, 262144}, {sndbuf, 262144}, {nodelay, true}]
```

The options supported are a subset of full options supported by the TCP/IP stack. A list of the supported options are provided in the Erlang inet documentation.

13. Server Options Configuration Setting

Server options for the MochiWeb component of CouchDB can now be added to the configuration file. Settings should be added to the `server_options` option of the `[httpd]` section of `local.ini`. For example:

```
[httpd]
server_options = [{backlog, 128}, {acceptor_pool_size, 16}]
```

14. Improved Error Messages

The errors reported when CouchDB is unable to read a required file have been updated so that explicit information about the files and problem can now be identified from the error message. The errors report file permission access either when reading or writing to configuration and database files.

The error is raised both through the log file and the error message returned through the API call as a JSON error message. For example, when setting configuration values:

```
shell> curl -H 'X-Couch-Persist: true' -X PUT http://couchdb:5984/_config/couchdb/delayed_commits -d '"false"'
{"error":"file_permission_error","reason":"/etc/couchdb/local.ini"}
```

Errors will always be reported using the `file_permission_error` error type.

During startup permissions errors on key files are also reported in the log with a descriptive error message and file location so that permissions can be fixed before restart.

15. Multiple micro-optimizations when reading data.

We found a number of places where CouchDB wouldn't do the absolute optimal thing when reading data and got rid of quite a few inefficiencies. The problem with small optimizations all over the place is that you may not notice them with every use-case, but we sure hope you can see an improvement overall.