CouchDB 1.1 Manual

CouchDB 1.1 Manual

Abstract

This manual documents the CouchDB 1.1 database system, including the installation, functionality, and CouchDB API.

Last document update: 21 Feb 2012 20:09; *Document built*: 21 Feb 2012 20:9.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

There are two interfaces to CouchDB, the built-in Futon web-based interface and the CouchDB API accessed through the HTTP REST interface. The former is the simplest way to view and monitor your CouchDB installation and perform a number of basic database and system operations. More information on using the Futon interface can be found in Section 1.1, “Using Futon”.

The primary way to interact with the CouchDB API is to use a client library or other interface that provides access to the underlying functionality through your chosen language or platform. However, since the API is supported through HTTP REST, you can interact with your CouchDB with any solution that supports the HTTP protocol.

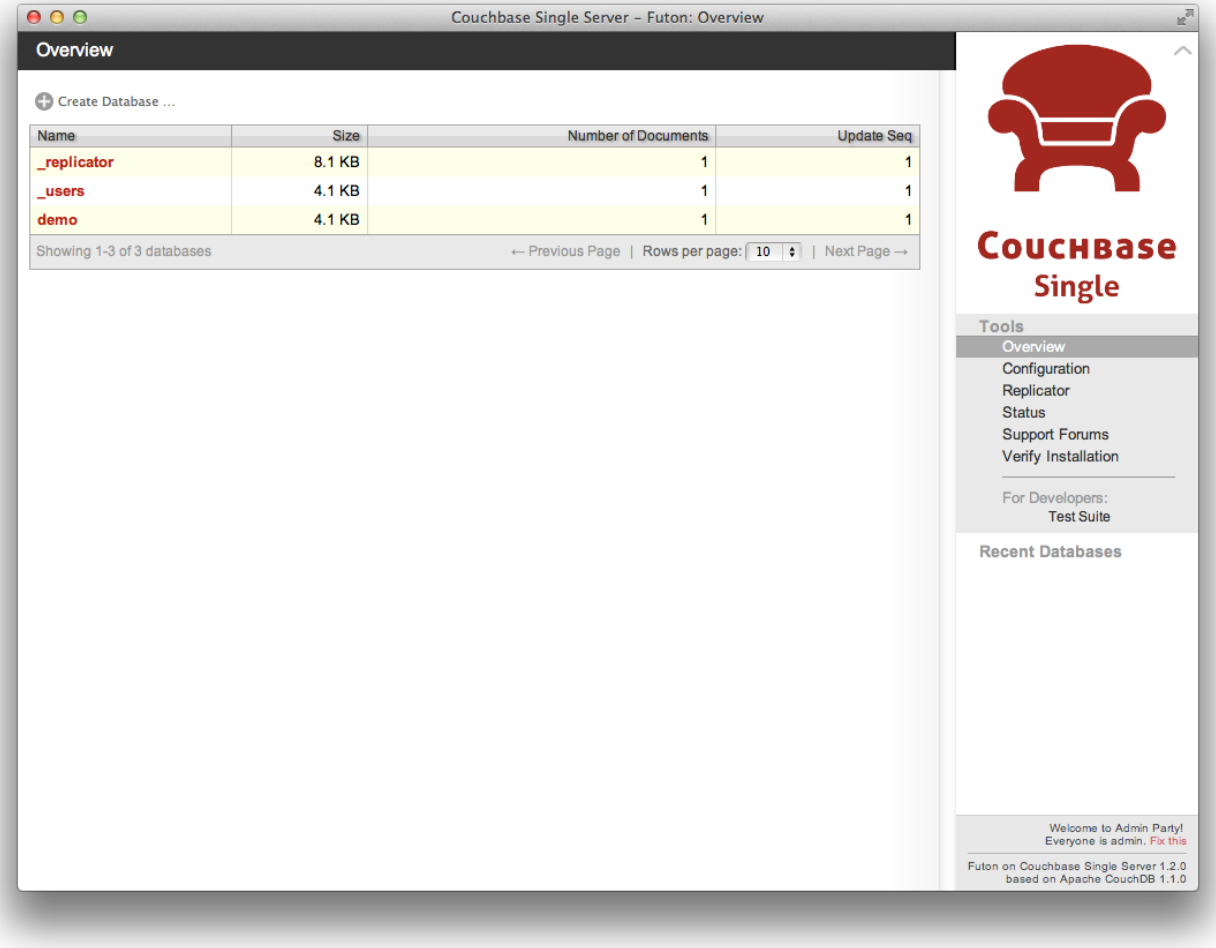
There are a number of different tools that talk the HTTP protocol and allow you to set and configure the necessary information. One tool for this that allows for access from the command-line is **curl**. See Section 1.2, “Using **curl**”.

1.1. Using Futon

Futon is a native web-based interface built into CouchDB. It provides a basic interface to the majority of the functionality, including the ability to create, update, delete and view documents and views, provides access to the configuration parameters, and an interface for initiating replication.

The default view is the Overview page which provides you with a list of the databases. The basic structure of the page is consistent regardless of the section you are in. The main panel on the left provides the main interface to the databases, configuration or replication systems. The side panel on the right provides navigation to the main areas of Futon interface:

Figure 1.1. Futon Overview



The main sections are:

- Overview

  The main overview page, which provides a list of the databases and provides the interface for querying the database and creating and updating documents. See Section 1.1.1, “Managing Databases and Documents”.

- Configuration

  An interface into the configuration of your CouchDB installation. The interface allows you to edit the different configurable parameters. For more details on configuration, see Chapter 12, *Configuring CouchDB*.

- Replicator

  An interface to the replication system, enabling you to initiate replication between local and remote databases. See Section 1.1.2, “Configuring Replication”.

- 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. See Section 9.2, “GET /_active_tasks”.

- Verify Installation

  The Verify Installation allows you to check whether all of the components of your CouchDB installation are correctly installed.

- Test Suite

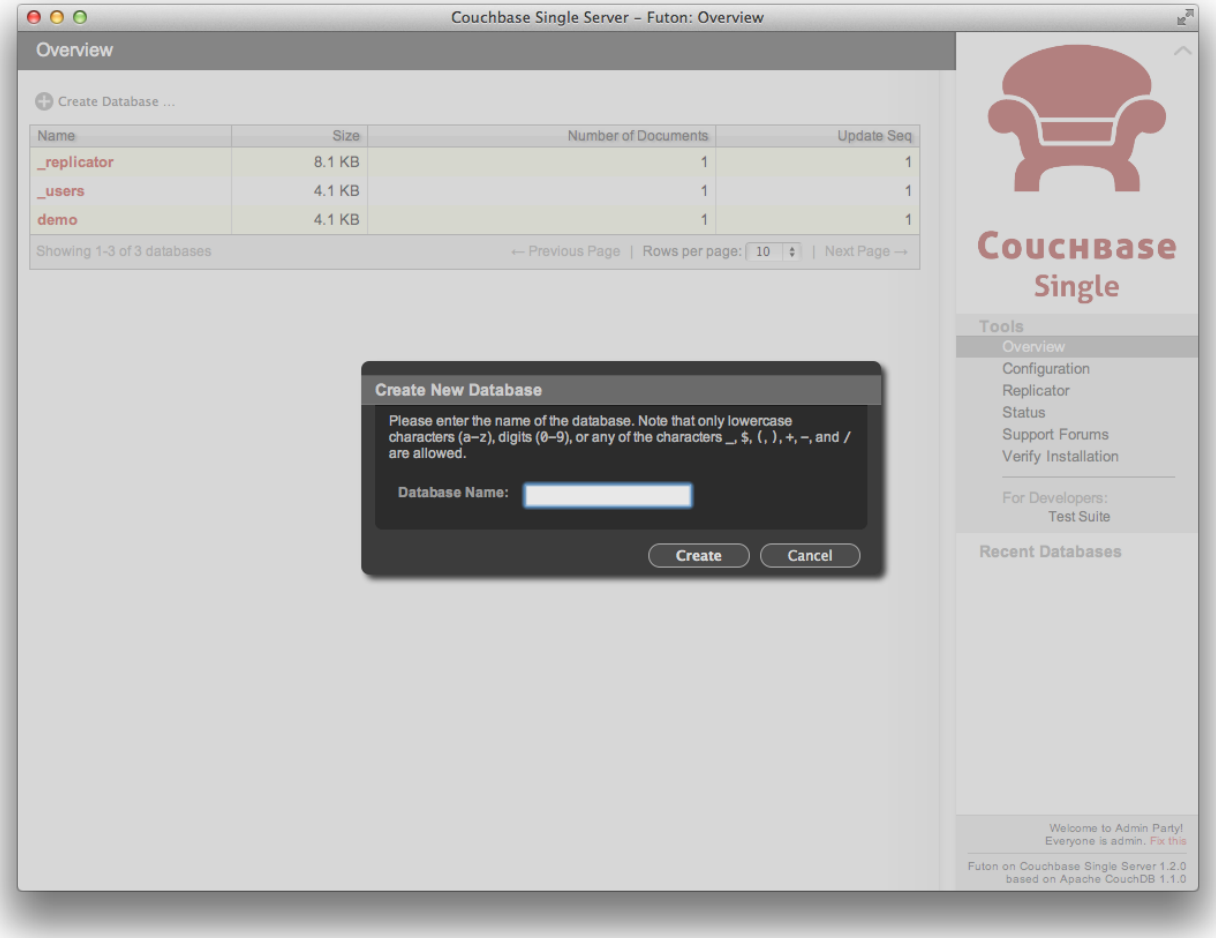
  The Test Suite section allows you to run the built-in test suite. This executes a number of test routines entirely within your browser to test the API and functionality of your CouchDB installation. If you select this page, you can run the tests by using the Run All button. This will execute all the tests, which may take some time.

1.1.1. Managing Databases and Documents

You 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 … button. You will be prompted for the database name, as shown in the figure below.

Figure 1.2. Creating a Database

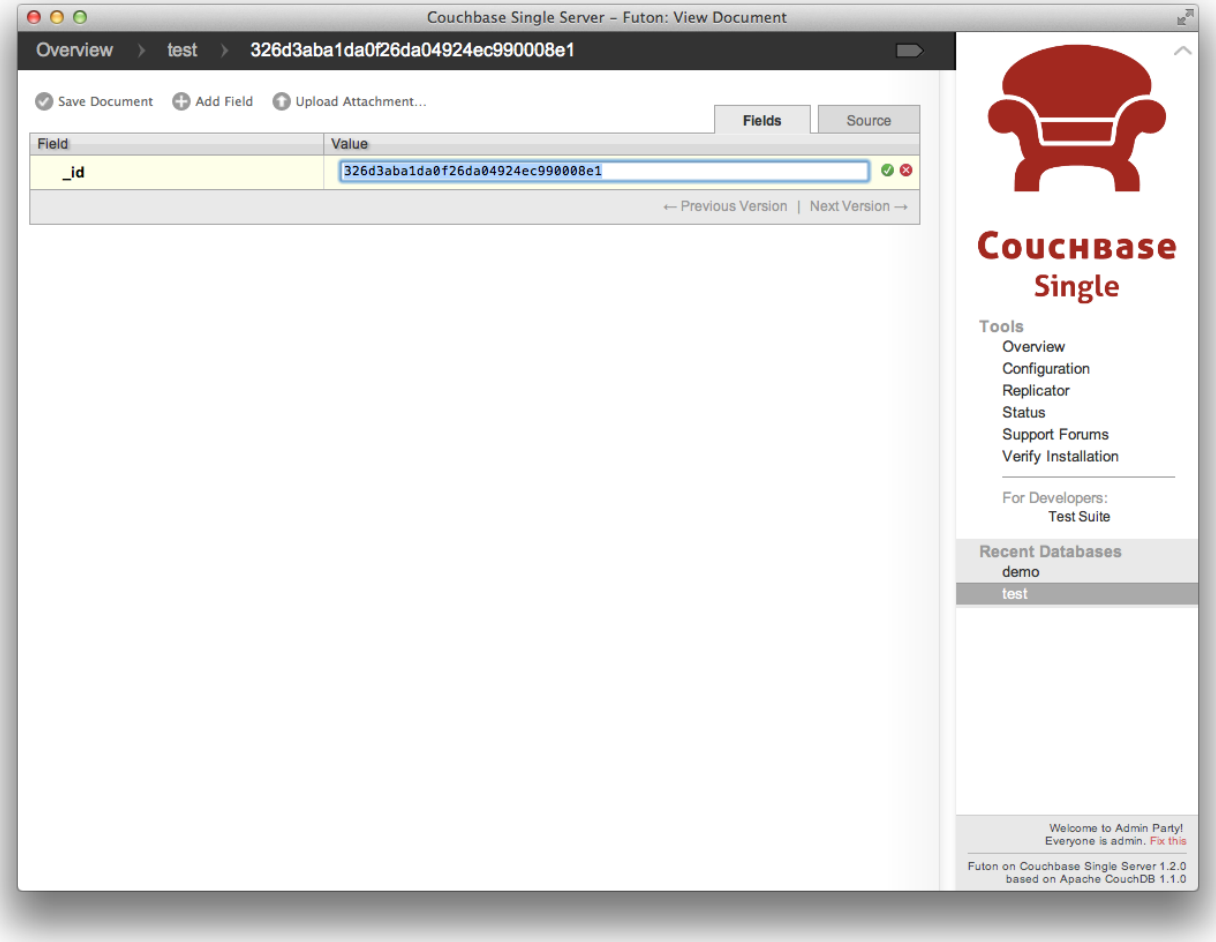


Once you have created the database (or selected an existing one), you will be shown a list of the current documents. If you create a new document, or select an existing document, you will be presented with the edit document display.

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.

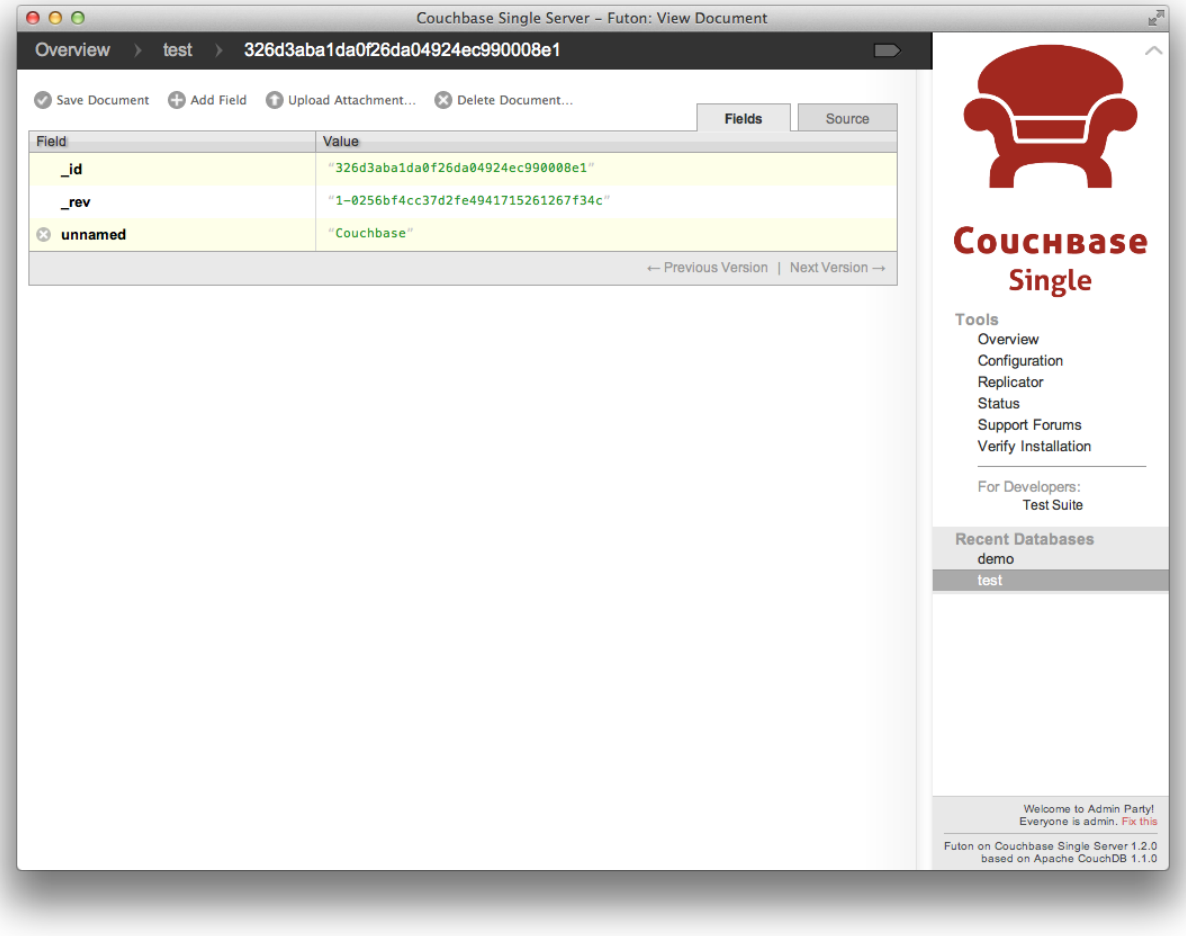
Figure 1.3. Editing a Document



To add a field to the document:

1. Click Add Field.
2. In the fieldname box, enter the name of the field you want to create. For example, “company”.
3. Click the green tick next to the field name to confirm the field name change.
4. Double-click the corresponding Value cell.
5. Enter a company name, for example “Example”.
6. Click the green tick next to the field value to confirm the field value.
7. The document is still not saved as this point. You 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).

Figure 1.4. Edited Document

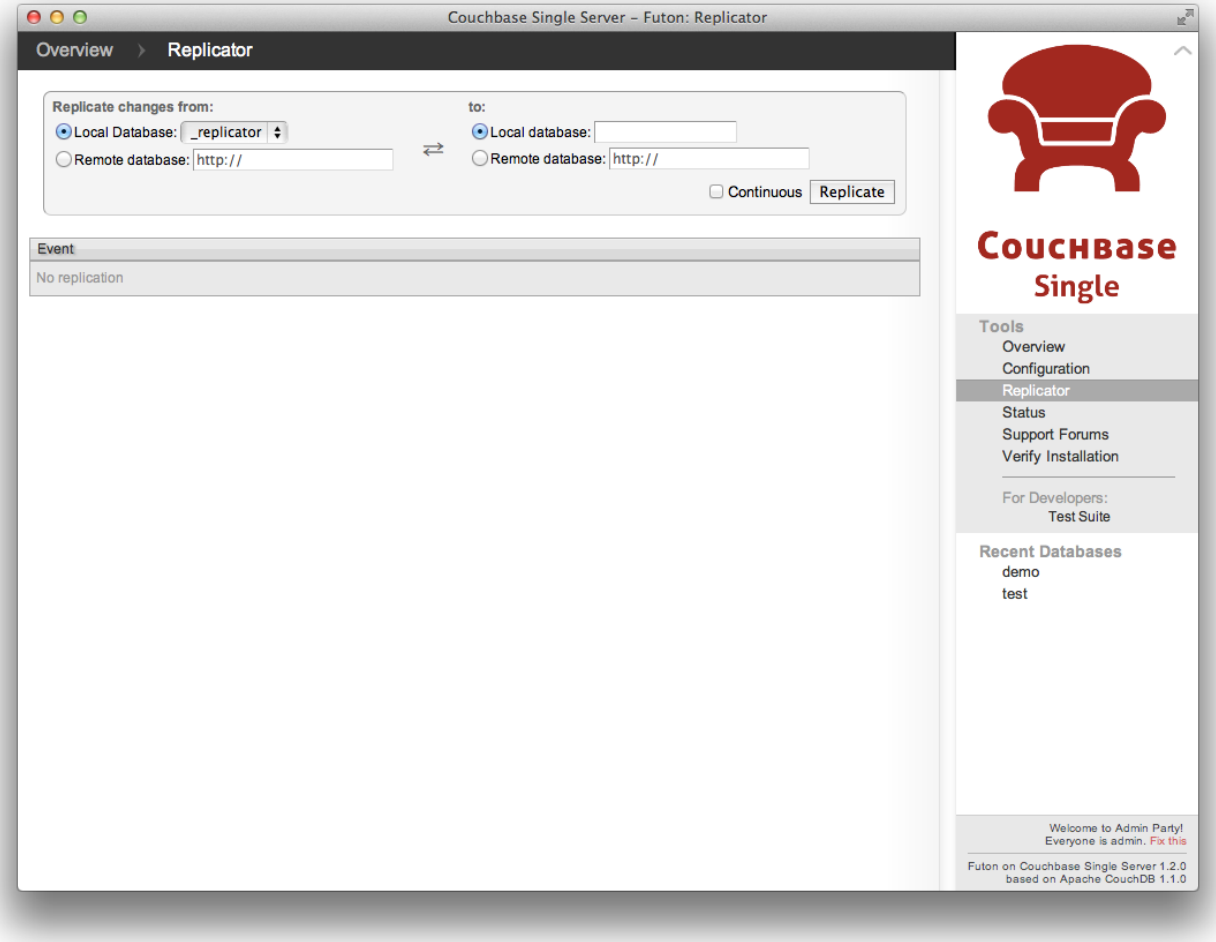


The same basic interface is used for all editng operations within Futon. You *must* rememmbr to save the individual element (fieldname, value) using the green tick button, before then saving the document.

1.1.2. Configuring Replication

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

Figure 1.5. Replication Form



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

If you are specifying a remote database name, you must specify the full URL of the remote database (including the host, port number and database name). If the remote instance requires authentication, you can specify the username and password as part of the URL, for example `http://username:pass@remotehost:5984/demo`.

To enable continuous replication, click the Continuous checkbox.

To start the replication process, click the Replicate button. The replication process should start and will continue in the background. If the replication process will take a long time, you can monitor the status of the replication using the Status option under the Tools menu.

Once replication has been completed, the page will show the information returned when the replication process completes by the API.

The Replicator tool is an interface to the underlying replication API. For more information, see Section 9.5, “`POST /_replicate`”. For more information on replication, see Chapter 3, *Replication*.

1.2. Using curl

The **curl** utility is a command line tool available on Unix, Linux, Mac OS X and Windows and many other platforms. **curl** provides easy access to the HTTP protocol (among others) directly from the command-line and is therefore an ideal way of interacting with CouchDB over the HTTP REST API.

For simple `GET` requests you can supply the URL of the request. For example, to get the database information:

```
shell> curl http://127.0.0.1:5984
```

This returns the database information (formatted in the output below for clarity):

```
{
   "modules" : {
      "geocouch" : "7fd793c10f3aa667a1088a937398bc5b51472b7f"
   },
   "couchdb" : "Welcome",
   "version" : "1.1.0",
}
```

Note

For some URLs, especially those that include special characters such as ampersand, exclamation mark, or question mark, you should quote the URL you are specifying on the command line. For example:

```
shell> curl 'http://couchdb:5984/_uuids?count=5'
```

You can explicitly set the HTTP command using the `-X` command line option. For example, when creating a database, you set the name of the database in the URL you send using a PUT request:

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

But to obtain the database information you use a `GET` request (with the return information formatted for clarity):

```
shell> curl -X GET http://127.0.0.1:5984/demo
{
   "compact_running" : false,
   "doc_count" : 0,
   "db_name" : "demo",
   "purge_seq" : 0,
   "committed_update_seq" : 0,
   "doc_del_count" : 0,
   "disk_format_version" : 5,
   "update_seq" : 0,
   "instance_start_time" : "1306421773496000",
   "disk_size" : 79
}
```

For certain operations, you must specify the content type of request, which you do by specifying the `Content-Type` header using the `-H` command-line option:

```
shell> curl -H 'Content-type: application/json' http://127.0.0.1:5984/_uuids
```

You can also submit 'payload' data, that is, data in the body of the HTTP request using the `-d` option. This is useful if you need to submit JSON structures, for example document data, as part of the request. For example, to submit a simple document to the `demo` database:

```
shell> curl -H 'Content-type: application/json' \
     -X POST http://127.0.0.1:5984/demo \
     -d '{"company": "Example, Inc."}'
{"ok":true,"id":"8843faaf0b831d364278331bc3001bd8",
 "rev":"1-33b9fbce46930280dab37d672bbc8bb9"}
```

In the above example, the argument after the `-d` option is the JSON of the document we want to submit.

The document can be accessed by using the automatically generated document ID that was returned:

```
shell> curl -X GET http://127.0.0.1:5984/demo/8843faaf0b831d364278331bc3001bd8
{"_id":"8843faaf0b831d364278331bc3001bd8",
 "_rev":"1-33b9fbce46930280dab37d672bbc8bb9",
 "company":"Example, Inc."}
```

The API samples in the Chapter 4, *CouchDB API* show the HTTP command, URL and any payload information that needs to be submitted (and the expected return value). All of these examples can be reproduced using **curl** with the command-line examples shown above.

Chapter 2. Features and Functionality

2.1. 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. The following example uses a text file to make the range request process easier.

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

Uploading this as an attachment to a `text` database using **curl**:

```
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"}
```

Requesting the whole file works as normal:

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

But to retrieve only the first 13 bytes using **curl**:

```
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. See 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.

2.2. 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

2.3. CommonJS support for map functions

CommonJS support allows you to use CommonJS notation inside `map` and `reduce` functions, 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.

The implementation is simplified 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.

2.4. Granular ETag support

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).

Chapter 3. Replication

3.1. 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.

3.1.1. 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` finish
```

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`.

3.1.2. 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.

3.1.3. 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+c
```

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.

3.1.4. 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.

3.1.5. 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.

3.1.6. 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.

3.1.7. 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 `[ ]`.

Chapter 4. CouchDB API

The CouchDB API is the primary method of interfacing to a CouchDB instance. Requests are made using HTTP and requests are used to request information from the database, store new data, and perform views and formatting of the information stored within the documents.

Requests to the API can be categorised by the different areas of the CouchDB system that you are accessing, and the HTTP method used to send the request. Different methods imply different operations, for example retrieval of information from the database is typically handled by the `GET` operation, while updates are handled by either a `POST` or `PUT` request. There are some differences between the information that must be supplied for the different methods. For a guide to the basic HTTP methods and request structure, see Section 4.1, “Request Format and Responses”.

For nearly all operations, the submitted data, and the returned data structure, is defined within a JavaScript Object Notation (JSON) object. Basic information on the content and data types for JSON are provided in Section 4.3, “JSON Basics”.

Errors when accessing the CouchDB API are reported using standard HTTP Status Codes. A guide to the generic codes returned by CouchDB are provided in Section 4.4, “HTTP Status Codes”.

When accessing specific areas of the CouchDB API, specific information and examples on the HTTP methods and request, JSON structures, and error codes are provided. For a guide to the different areas of the API, see Section 4.5, “CouchDB API Overview”.

4.1. Request Format and Responses

CouchDB supports the following HTTP request methods:

- `GET`

  Request the specified item. As with normal HTTP requests, the format of the URL defines what is returned. With CouchDB this can include static items, database documents, and configuration and statistical information. In most cases the information is returned in the form of a JSON document.

- `HEAD`

  The `HEAD` method is used to get the HTTP header of a `GET` request without the body of the response.

- `POST`

  Upload data. Within CouchDB `POST` is used to set values, including uploading documents, setting document values, and starting certain administration commands.

- `PUT`

  Used to put a specified resource. In CouchDB `PUT` is used to create new objects, including databases, documents, views and design documents.

- `DELETE`

  Deletes the specified resource, including documents, views, and design documents.

- `COPY`

  A special method that can be used to copy documents and objects.

If you use the an unsupported HTTP request type with a URL that does not support the specified type, a 405 error will be returned, listing the supported HTTP methods. For example:

```
{
    "error":"method_not_allowed",
    "reason":"Only GET,HEAD allowed"
}
```

The CouchDB design document API and the functions when returning HTML (for example as part of a show or list) enables you to include custom HTTP headers through the `headers` block of the return object. For more information, see ???.

4.2. HTTP Headers

Because CouchDB uses HTTP for all communication, you need to ensure that the correct HTTP headers are supplied (and processed on retrieval) so that you get the right format and encoding. Different environments and clients will be more or less strict on the effect of these HTTP headers (especially when not present). Where possible you should be as specific as possible.

4.2.1. Request Headers

- `Content-type`

  Specifies the content type of the information being supplied within the request. The specification uses MIME type specifications. For the majority of requests this will be JSON (`application/json`). For some settings the MIME type will be plain text. When uploading attachments it should be the corresponding MIME type for the attachment or binary (`application/octet-stream`).

  The use of the `Content-type` on a request is highly recommended.

- `Accept`

  Specifies the list of accepted data types to be returned by the server (i.e. that are accepted/understandable by the client). The format should be a list of one or more MIME types, separated by colons.

  For the majority of requests the definition should be for JSON data (`application/json`). For attachments you can either specify the MIME type explicitly, or use `*/*` to specify that all file types are supported. If the `Accept` header is not supplied, then the `*/*` MIME type is assumed (i.e. client accepts all formats).

  The use of `Accept` in queries for CouchDB is not required, but is highly recommended as it helps to ensure that the data returned can be processed by the client.

  If you specify a data type using the `Accept` header, CouchDB will honor the specified type in the `Content-type` header field returned. For example, if you explicitly request `application/json` in the `Accept` of a request, the returned HTTP headers will use the value in the returned `Content-type` field.

  For example, when sending a request without an explicit `Accept` header, or when specifying `*/*`:

```
GET /recipes HTTP/1.1
Host: couchdb:5984
Accept: */*
```

  The returned headers are:

```
Server: CouchDB/1.0.1 (Erlang OTP/R13B)
Date: Thu, 13 Jan 2011 13:39:34 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 227
Cache-Control: must-revalidate
```

  Note that the returned content type is `text/plain` even though the information returned by the request is in JSON format.

Explicitly specifying the `Accept` header:

```
GET /recipes HTTP/1.1
Host: couchdb:5984
Accept: application/json
```

The headers returned include the `application/json` content type:

```
Server: CouchDB/1.0.1 (Erlang OTP/R13B)
Date: Thu, 13 Jan 2011 13:40:11 GMT
Content-Type: application/json
Content-Length: 227
Cache-Control: must-revalidate
```

4.2.2. Response Headers

Response headers are returned by the server when sending back content and include a number of different header fields, many of which are standard HTTP response header and have no significance to CouchDB operation. The list of response headers important to CouchDB are listed below.

- `Content-type`

  Specifies the MIME type of the returned data. For most request, the returned MIME type is `text/plain`. All text is encoded in Unicode (UTF-8), and this is explicitly stated in the returned `Content-type`, as `text/plain;charset=utf-8`.

- `Cache-control`

  The cache control HTTP response header provides a suggestion for client caching mechanisms on how to treat the returned information. CouchDB typically returns the `must-revalidate`, which indicates that the information should be revalidated if possible. This is used to ensure that the dynamic nature of the content is correctly updated.

- `Content-length`

  The length (in bytes) of the returned content.

- `Etag`

  The `Etag` HTTP header field is used to show the revision for a document.

4.3. JSON Basics

The majority of requests and responses to CouchDB use the JavaScript Object Notation (JSON) for formatting the content and structure of the data and responses.

JSON is used because it is the simplest and easiest to use solution for working with data within a web browser, as JSON structures can be evaluated and used as JavaScript objects within the web browser environment. JSON also integrates with the server-side JavaScript used within CouchDB.

JSON supports the same basic types as supported by JavaScript, these are:

- Number (either integer or floating-point).
- String; this should be enclosed by double-quotes and supports Unicode characters and backslash escaping. For example:

  ```
  "A String"
  ```

- Boolean - a `true` or `false` value. You can use these strings directly. For example:

```
{ "value": true}
```

- Array - a list of values enclosed in square brackets. For example:

```
["one", "two", "three"]
```

- Object - a set of key/value pairs (i.e. an associative array, or hash). The key must be a string, but the value can be any of the supported JSON values. For example:

```
{
   "servings" : 4,
   "subtitle" : "Easy to make in advance, and then cook when ready",
   "cooktime" : 60,
   "title" : "Chicken Coriander"
}
```

  In CouchDB, the JSON object is used to represent a variety of structures, including the main CouchDB document.

Parsing JSON into a JavaScript object is supported through the `eval()` function in JavaScript, or through various libraries that will perform the parsing of the content into a JavaScript object for you. Libraries for parsing and generating JSON are available in many languages, including Perl, Python, Ruby, Erlang and others.

Warning

Care should be taken to ensure that your JSON structures are valid, invalid structures will cause CouchDB to return an HTTP status code of 500 (server error). See HTTP Status Code 500 [22] .

4.4. HTTP Status Codes

With the interface to CouchDB working through HTTP, error codes and statuses are reported using a combination of the HTTP status code number, and corresponding data in the body of the response data.

A list of the error codes returned by CouchDB, and generic descriptions of the related errors are provided below. The meaning of different status codes for specific request types are provided in the corresponding API call reference.

- `200 - OK`

  Request completed successfully.

- `201 - Created`

  Document created successfully.

- `202 - Accepted`

  Request has been accepted, but the corresponding operation may not have completed. This is used for background operations, such as database compaction.

- `304 - Not Modified`

  The additional content requested has not been modified. This is used with the ETag system to identify the version of information returned.

- `400 - Bad Request`

  Bad request structure. The error can indicate an error with the request URL, path or headers. Differences in the supplied MD5 hash and content also trigger this error, as this may indicate message corruption.

- `401 - Unauthorized`

The item requested was not available using the supplied authorization, or authorization was not supplied.

- `403 - Forbidden`

  The requested item or operation is forbidden.

- `404 - Not Found`

  The requested content could not be found. The content will include further information, as a JSON object, if available. The structure will contain two keys, `error` and `reason`. For example:

  ```
  {"error":"not_found","reason":"no_db_file"}
  ```

- `405 - Resource Not Allowed`

  A request was made using an invalid HTTP request type for the URL requested. For example, you have requested a `PUT` when a `POST` is required. Errors of this type can also triggered by invalid URL strings.

- `406 - Not Acceptable`

  The requested content type is not supported by the server.

- `409 - Conflict`

  Request resulted in an update conflict.

- `412 - Precondition Failed`

  The request headers from the client and the capabilities of the server do not match.

- `415 - Bad Content Type`

  The content types supported, and the content type of the information being requested or submitted indicate that the content type is not supported.

- `416 - Requested Range Not Satisfiable`

  The range specified in the request header cannot be satisfied by the server.

- `417 - Expectation Failed`

  When sending documents in bulk, the bulk load operation failed.

- `500 - Internal Server Error`

  The request was invalid, either because the supplied JSON was invalid, or invalid information was supplied as part of the request.

4.5. CouchDB API Overview

The components of the API URL path help determine the part of the CouchDB server that is being accessed. The result is the structure of the URL request both identifies and effectively describes the area of the database you are accessing.

As with all URLs, the individual components are separated by a forward slash.

As a general rule, URL components and JSON fields starting with the `_` (underscore) character represent a special component or entity within the server or returned object. For example, the URL fragment `/_all_dbs` gets a list of all of the databases in a CouchDB instance.

The remainder of the URL API structure can be divided up according to the URL structure. The different sections are divided as follows:

- `/db`

  Database methods, related to adding, updating or deleting databases, and setting database parameters and operations. For more detailed information, see Chapter 5, *CouchDB API Server Database Methods* .

- `/db/doc`

  Document methods, those that create, store, update or delete CouchDB documents and their attachments. For more information, see Chapter 6, *CouchDB API Server Document Methods*.

- `/db/_local/local-doc`

  Document methods, those that create, store, update or delete CouchDB documents only within the local database. Local documents are not synchronized with other databases. For more information, see Chapter 7, *CouchDB API Server Local (non-replicating) Document Methods*.

- `/db/_design/design-doc`

  Design documents provide the methods and structure for recovering information from a CouchDB database in the form of views, shows and lists. For more information, see Chapter 8, *CouchDB API Server Design Document Methods*.

- `/_special`

  Special methods that obtain or set information about the CouchDB instance, including methods for configuring replication, accessing the logs, and generate Universally Unique IDs (UUIDs). For more information, see Chapter 9, *CouchDB API Server Miscellaneous Methods*.

- `/_config`

  Methods for getting, and settings, CouchDB configuration parameters. For more information, see Chapter 9, *CouchDB API Server Miscellaneous Methods*.

Chapter 5. CouchDB API Server Database Methods

The Database methods provide an interface to an entire database withing CouchDB. These are database, rather than document, level requests.

A list of the available methods and URL paths are provided below:

Table 5.1. Database API Calls

Method	Path	Description
GET	/db	Returns database information
PUT	/db	Create a new database
DELETE	/db	Delete an existing database
GET	/db/_all_docs	Returns a built-in view of all documents in this database
POST	/db/_all_docs	Returns certain rows from the built-in view of all documents
POST	/db/_bulk_docs	Insert multiple documents in to the database in a single request
GET	/db/_changes	Returns changes for the given database
POST	/db/_compact	Starts a compaction for the database
POST	/db/_compact/design-doc	Starts a compaction for all the views in the selected design document
POST	/db/_ensure_full_commit	Makes sure all uncommitted changes are written and synchronized to the disk
POST	/db/_missing_revs	Given a list of document revisions, returns the document revisions that do not exist in the database
POST	/db/_purge	Purge some historical documents entirely from database history
POST	/db/_revs_diff	Given a list of document revisions, returns differences between the given revisions and ones that are in the database
GET	/db/_revs_limit	Gets the limit of historical revisions to store for a single document in the database
PUT	/db/_revs_limit	Sets the limit of historical revisions to store for a single document in the database
GET	/db/_security	Returns the special security object for the database
PUT	/db/_security	Sets the special security object for the database
POST	/db/_temp_view	Execute a given view function for all documents and return the result

Method	Path	Description
POST	/db/_view_cleanup	Removes view files that are not used by any design document

For all the database methods, the database name within the URL path should be the database name that you wish to perform the operation on. For example, to obtain the meta information for the database `recipes`, you would use the HTTP request:

```
GET /recipes
```

For clarity, the form below is used in the URL paths:

```
GET /db
```

Where `db` is the name of any database.

5.1. GET /db

Method	GET /db
Request	None
Response	Information about the database in JSON format
Admin Privileges Required	no
Return Codes	
404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Gets information about the specified database. For example, to retrieve the information for the database `recipe`:

```
GET http://couchdb:5984/recipes
Accept: application/json
```

The JSON response contains meta information about the database. A sample of the JSON returned for an empty database is provided below:

```
{
   "compact_running" : false,
   "committed_update_seq" : 375048,
   "disk_format_version" : 5,
   "disk_size" : 33153123,
   "doc_count" : 18386,
   "doc_del_count" : 0,
   "db_name" : "recipes",
   "instance_start_time" : "1290700340925570",
   "purge_seq" : 10,
   "update_seq" : 375048
}
```

The elements of the returned structure are shown in the table below:

Table 5.2. CouchDB database information object

Field	Description
committed_update_seq	The number of committed update.
compact_running	Set to true if the database compaction routine is operating on this database.
db_name	The name of the database.
disk_format_version	The version of the physical format used for the data when it is stored on disk.

`disk_size`	Size in bytes of the data as stored on the disk. Views indexes are not included in the calculation.
`doc_count`	A count of the documents in the specified database.
`doc_del_count`	Number of deleted documents
`instance_start_time`	Timestamp of when the database was created, expressed in milliseconds since the epoch.
`purge_seq`	The number of purge operations on the database.
`update_seq`	The current number of updates to the database.

5.2. `PUT /db`

Method	`PUT /db`
Request	None
Response	JSON success statement
Admin Privileges Required	no
Return Codes	
400	Invalid database name
412	Database already exists

Creates a new database. The database name must be composed of one or more of the following characters:

- Lowercase characters (`a-z`)
- Name must begin with a lowercase letter
- Digits (`0-9`)
- Any of the characters `_`, `$`, `(`, `)`, `+`, `-`, and `/`.

Trying to create a database that does not meet these requirements will return an error quoting these restrictions.

To create the database `recipes`:

```
PUT http://couchdb:5984/recipes
Content-Type: application/json
```

The returned content contains the JSON status:

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

Anything should be treated as an error, and the problem should be taken form the HTTP response code.

5.3. `DELETE /db`

Method	`DELETE /db`
Request	None
Response	JSON success statement
Admin Privileges Required	no
Return Codes	
200	Database has been deleted

404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Deletes the specified database, and all the documents and attachments contained within it.

To delete the database `recipes` you would send the request:

```
DELETE http://couchdb:5984/recipes
Content-Type: application/json
```

If successful, the returned JSON will indicate success

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

5.4. `GET /db/_changes`

Method	`GET /db/_changes`	
Request	None	
Response	JSON of the changes to the database	
Admin Privileges Required	no	
Query Arguments	**Argument**	`doc_ids`
	Description	Specify the list of documents IDs to be filtered
	Optional	yes
	Type	json
	Default	none
	Argument	`feed`
	Description	Type of feed
	Optional	yes
	Type	string
	Default	normal
	Supported Values	
	`continuous`	Continuous (non-polling) mode
	`longpoll`	Long polling mode
	`normal`	Normal mode
	Argument	`filter`
	Description	Filter function from a design document to get updates
	Optional	yes
	Type	string
	Default	none
	Supported Values	

	Argument	`heartbeat`
	Description	Period after which an empty line is sent during longpoll or continuous
	Optional	yes
	Type	numeric
	Default	60000
	Quantity	milliseconds
	Argument	`include_docs`
	Description	Include the document with the result
	Optional	yes
	Type	boolean
	Default	false
	Argument	`limit`
	Description	Maximum number of rows rows to return
	Optional	yes
	Type	numeric
	Default	none
	Argument	`since`
	Description	Start the results from changes immediately after the specified sequence number
	Optional	yes
	Type	numeric
	Default	0
	Argument	`timeout`
	Description	Maximum period to wait before the response is sent
	Optional	yes
	Type	numeric
	Default	60000
	Quantity	milliseconds

Obtains a list of the changes made to the database. This can be used to monitor for update and modifications to the database for post processing or synchronization. There are three different types of supported changes feeds, poll, longpoll, and continuous. All requests are poll requests by default. You can select any feed type explicitly using the `feed` query argument.

- **Poll**

  With polling you can request the changes that have occured since a specific sequence number. This returns the JSON structure containing the changed document information. When you perform a poll change request, only the changes since the specific sequence number are returned. For example, the query

```
DELETE http://couchdb:5984/recipes/_changes
Content-Type: application/json
```

  Will get all of the changes in the database. You can request a starting point using the `since` query argument and specifying the sequence number. You will need to record the latest sequence number in your client and then use this when making another request as the new value to the `since` parameter.

- **Longpoll**

  With long polling the request to the server will remain open until a change is made on the database, when the changes will be reported, and then the connection will close. The long poll is useful when you want to monitor for changes for a specific purpose without wanting to monitoring continuously for changes.

  Because the wait for a change can be significant you can set a timeout before the connection is automatically closed (the `timeout` argument). You can also set a heartbeat interval (using the `heartbeat` query argument), which sends a newline to keep the connection open.

- **Continuous**

  Continuous sends all new changes back to the client immediately, without closing the connection. In continuous mode the format of the changes is slightly different to accommodate the continuous nature while ensuring that the JSON output is still valid for each change notification.

  As with the longpoll feed type you can set both the timeout and heartbeat intervals to ensure that the connection is kept open for new changesand updates.

The return structure for `normal` and `longpoll` modes is a JSON array of changes objects, and the last update sequence number. The structure is described in the following table.

Table 5.3. Changes information for a database

Field	Description
`last_seq`	Last change sequence number
`results [array]`	Changes made to a database
`changes [array]`	List of changes, field-by-field, for this document
`id`	Document ID
`seq`	Update sequence number

The return format for `continuous` mode the server sends a CRLF (carriage-return, linefeed) delimited line for each change. Each line contains the JSON object.

You can also request the full contents of each document change (instead of just the change notification) by using the `include_docs` parameter.

5.4.1. Filtering

You can filter the contents of the changes feed in a number of ways. The most basic way is to specify one or more document IDs to the query. This causes the returned structure value to only contain changes for the specified IDs. Note that the value of this query argument should be a JSON formatted array.

You can also filter the `_changes` feed by defining a filter function within a design document. The specification for the filter is the same as for replication filters. You specify the name of the filter function to the `filter` parameter, specifying the design document name and filter name. For example:

```
GET /db/_changes?filter=design_doc/filtername
```

The `_changes` feed can 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` parameter as a JSON array.

For more information, see ???.

5.5. `POST /db/_compact`

Method	`POST /db/_compact`
Request	None
Response	JSON success statement
Admin Privileges Required	no
Return Codes	
202	Compaction request has been accepted
404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Request compaction of the specified database. Compaction compresses the disk database file by performing the following operations:

- Writes a new version of the database file, removing any unused sections from the new version during write. Because a new file is temporary created for this purpose, you will need twice the current storage space of the specified database in order for the compaction routine to complete.
- Removes old revisions of documents from the database, up to the per-database limit specified by the `_revs_limit` database parameter. See Section 5.1, “`GET /db`” .

Compaction can only be requested on an individual database; you cannot compact all the databases for a CouchDB instance. The compaction process runs as a background process.

You can determine if the compaction process is operating on a database by obtaining the database meta information, the `compact_running` value of the returned database structure will be set to true. See Section 5.1, “`GET /db`” .

You can also obtain a list of running processes to determine whether compaction is currently running. See Section 9.2, “`GET /_active_tasks`”.

5.6. `POST /db/_compact/design-doc`

Method	`POST /db/_compact/design-doc`
Request	None
Response	JSON success statement
Admin Privileges Required	yes
Return Codes	
202	Compaction request has been accepted
404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Compacts the view indexes associated with the specified design document. You can use this in place of the full database compaction if you know a specific set of view indexes have been affected by a recent database change.

For example, to compact the views associated with the `recipes` design document:

```
POST http://couchdb:5984/recipes/_compact/recipes
Content-Type: application/json
```

CouchDB will immediately return with a status indicating that the compaction request has been received (HTTP status code 202):

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

5.7. POST /db/_view_cleanup

Method	`POST /db/_view_cleanup`
Request	None
Response	JSON success statement
Admin Privileges Required	yes

Cleans up the cached view output on disk for a given view. For example:

```
POST http://couchdb:5984/recipes/_view_cleanup
Content-Type: application/json
```

If the request is successful, a basic status message us returned:

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

5.8. POST /db/_ensure_full_commit

Method	`POST /db/_ensure_full_commit`
Request	None
Response	JSON success statement
Admin Privileges Required	no
Return Codes	
200	Commit completed successfully
404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Commits any recent changes to the specified database to disk. You should call this if you want to ensure that recent changes have been written. For example, to commit all the changes to disk for the database `recipes` you would use:

```
POST http://couchdb:5984/recipes/_ensure_full_commit
Content-Type: application/json
```

This returns a status message, containing the success message and the timestamp for when the CouchDB instance was started:

```
{
  "ok" : true,
  "instance_start_time" : "1288186189373361"
}
```

5.9. POST /db/_bulk_docs

Method	POST /db/_bulk_docs
Request	JSON of the docs and updates to be applied
Response	JSON of updated documents
Admin Privileges Required	no
Return Codes	
201	Document(s) have been created or updated

The bulk document API allows you to create and update multiple documents at the same time within a single request. The basic operation is similar to creating or updating a single document, except that you batch the document structure and information and . When creating new documents the document ID is optional. For updating existing documents, you must provide the document ID, revision information, and new document values.

For both inserts and updates the basic structure of the JSON is the same:

Table 5.4. Bulk Documents

Field	Description
all_or_nothing (optional)	Sets the database commit mode to use all-or-nothing semantics
docs [array]	Bulk Documents Document
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_deleted (optional)	Whether the document should be deleted

5.9.1. Inserting Documents in Bulk

To insert documents in bulk into a database you need to supply a JSON structure with the array of documents that you want to add to the database. Using this method you can either include a document ID, or allow the document ID to be automatically generated.

For example, the following inserts three new documents, two with the supplied document IDs, and one which will have a document ID generated:

```
{
   "docs" : [
      {
         "_id" : "FishStew",
         "servings" : 4,
         "subtitle" : "Delicious with fresh bread",
         "title" : "Fish Stew"
      },
      {
         "_id" : "LambStew",
         "servings" : 6,
         "subtitle" : "Delicious with scone topping",
         "title" : "Lamb Stew"
      },
      {
         "servings" : 8,
         "subtitle" : "Delicious with suet dumplings",
         "title" : "Beef Stew"
      },
   ]
}
```

The return type from a bulk insertion will be 201, with the content of the returned structure indicating specific success or otherwise messages on a per-document basis.

The return structure from the example above contains a list of the documents created, here with the combination and their revision IDs:

```
POST http://couchdb:5984/recipes/_bulk_docs
Content-Type: application/json

[
   {
      "id" : "FishStew",
      "rev" : "1-9c65296036141e575d32ba9c034dd3ee",
   },
   {
      "id" : "LambStew",
      "rev" : "1-34c318924a8f327223eed702ddfdc66d",
   },
   {
      "id" : "7f7638c86173eb440b8890839ff35433",
      "rev" : "1-857c7cbeb6c8dd1dd34a0c73e8da3c44",
   }
]
```

The content and structure of the returned JSON will depend on the transaction semantics being used for the bulk update; see Section 5.9.3, “Bulk Documents Transaction Semantics” for more information. Conflicts and validation errors when updating documents in bulk must be handled separately; see Section 5.9.4, “Bulk Document Validation and Conflict Errors”.

5.9.2. Updating Documents in Bulk

The bulk document update procedure is similar to the insertion procedure, except that you must specify the document ID and current revision for every document in the bulk update JSON string.

For example, you could send the following request:

```
POST http://couchdb:5984/recipes/_bulk_docs
Content-Type: application/json

{
   "docs" : [
      {
         "_id" : "FishStew",
         "_rev" : "1-9c65296036141e575d32ba9c034dd3ee",
         "servings" : 4,
         "subtitle" : "Delicious with freshly baked bread",
         "title" : "Fish Stew"
      },
      {
         "_id" : "LambStew",
         "_rev" : "1-34c318924a8f327223eed702ddfdc66d",
         "servings" : 6,
         "subtitle" : "Serve with a wholemeal scone topping",
         "title" : "Lamb Stew"
      },
      {
         "_id" : "7f7638c86173eb440b8890839ff35433"
         "_rev" : "1-857c7cbeb6c8dd1dd34a0c73e8da3c44",
         "servings" : 8,
         "subtitle" : "Hand-made dumplings make a great accompaniment",
         "title" : "Beef Stew"
      }
   ]
}
```

The return structure is the JSON of the updated documents, with the new revision and ID information:

```
[
   {
      "id" : "FishStew",
      "rev" : "2-e7af4c4e9981d960ecf78605d79b06d1"
   },
   {
      "id" : "LambStew",
      "rev" : "2-0786321986194c92dd3b57dfbfc741ce"
   },
   {
      "id" : "7f7638c86173eb440b8890839ff35433",
      "rev" : "2-bdd3bf3563bee516b96885a66c743f8e"
   }
]
```

You can optionally delete documents during a bulk update by adding the `_deleted` field with a value of `true` to each docment ID/revision combination within the submitted JSON structure.

The return type from a bulk insertion will be 201, with the content of the returned structure indicating specific success or otherwise messages on a per-document basis.

The content and structure of the returned JSON will depend on the transaction semantics being used for the bulk update; see Section 5.9.3, “Bulk Documents Transaction Semantics” for more information. Conflicts and validation errors when updating documents in bulk must be handled separately; see Section 5.9.4, “Bulk Document Validation and Conflict Errors”.

5.9.3. Bulk Documents Transaction Semantics

CouchDB supports two different modes for updating (or inserting) documents using the bulk documentation system. Each mode affects both the state of the documents in the event of system failure, and the level of conflict checking performed on each document. The two modes are:

- `non-atomic`

  The default mode is non-atomic, that is, CouchDB will only guarantee that some of the documents will be saved when you send the request. The response will contain the list of documents successfully inserted or updated during the process. In the event of a crash, some of the documents may have been successfully saved, and some will have been lost.

  In this mode, the response structure will indicate whether the document was updated by supplying the new `_rev` parameter indicating a new document revision was created. If the update failed, then you will get an `error` of type `conflict`. For example:

  ```
  [
     {
        "id" : "FishStew",
        "error" : "conflict",
        "reason" : "Document update conflict."
     },
     {
        "id" : "LambStew",
        "error" : "conflict",
        "reason" : "Document update conflict."
     },
     {
        "id" : "7f7638c86173eb440b8890839ff35433",
        "error" : "conflict",
        "reason" : "Document update conflict."
     }
  ]
  ```

  In this case no new revision has been created and you will need to submit the document update, with the correct revision tag, to update the document.

- `all-or-nothing`

In all-or-nothing mode, either all documents are written to the database, or no documents are written to the database, in the event of a system failure during commit.

In addition, the per-document conflict checking is not performed. Instead a new revision of the document is created, even if the new revision is in conflict with the current revision in the database. The returned structure contains the list of documents with new revisions:

```
[
   {
      "id" : "FishStew",
      "rev" : "2-e7af4c4e9981d960ecf78605d79b06d1"
   },
   {
      "id" : "LambStew",
      "rev" : "2-0786321986194c92dd3b57dfbfc741ce"
   },
   {
      "id" : "7f7638c86173eb440b8890839ff35433",
      "rev" : "2-bdd3bf3563bee516b96885a66c743f8e"
   }
]
```

When updating documents using this mode the revision of a document included in views will be arbitrary. You can check the conflict status for a document by using the `conflicts=true` query argument when accessing the view. Conflicts should be handled individually to ensure the consistency of your database.

To use this mode, you must include the `all_or_nothing` field (set to true) within the main body of the JSON of the request.

The effects of different database operations on the different modes are summarized in the table below:

Table 5.5. Conflicts on Bulk Inserts

Transaction Mode	Transaction	Cause	Resolution
Non-atomic	Insert	Requested document ID already exists	Resubmit with different document ID, or update the existing document
Non-atomic	Update	Revision missing or incorrect	Resubmit with correct revision
All-or-nothing	Insert	Additional revision inserted	Resolve conflicted revisions
All-or-nothing	Update	Additional revision inserted	Resolve conflicted revisions

Replication of documents is independent of the type of insert or update. The documents and revisions created during a bulk insert or update are replicated in the same way as any other document. This can mean that if you make use of the all-or-nothing mode the exact list of documents, revisions (and their conflict state) may or may not be replicated to other databases correctly.

5.9.4. Bulk Document Validation and Conflict Errors

The JSON returned by the `_bulk_docs` operation consists of an array of JSON structures, one for each document in the original submission. The returned JSON structure should be examined to ensure that all of the documents submitted in the original request were successfully added to the database.

The exact structure of the returned information is shown in Table 5.6, “ Bulk Document Response ”.

Table 5.6. Bulk Document Response

Field	Description

docs [array]	Bulk Docs Returned Documents
error	Error type
id	Document ID
reason	Error string with extended reason

When a document (or document revision) is not correctly comitted to the database because of an error, you should check the `error` field to determine error type and course of action. Errors will be one of the following type:

- `conflict`

  The document as submitted is in conflict. If you used the default bulk transaction mode then the new revision will not have been created and you will need to re-submit the document to the database. If you used `all-or-nothing` mode then you will need to manually resolve the conflicted revisions of the document.

  Conflict resolution of documents added using the bulk docs interface is identical to the resolution procedures used when resolving conflict errors during replication.

- `forbidden`

  Entries with this error type indicate that the validation routine applied to the document during submission has returned an error.

  For example, if your validation routine includes the following:

```
throw({forbidden: 'invalid recipe ingredient'});
```

  The error returned will be:

```
{
   "id" : "7f7638c86173eb440b8890839ff35433",
   "error" : "forbidden",
   "reason" : "invalid recipe ingredient"
}
```

  For more information, see ???.

5.10. `POST /db/_temp_view`

Method	`POST /db/_temp_view`
Request	JSON with the temporary view definition
Response	Temporary view result set
Admin Privileges Required	yes

Creates (and executes) a temporary view based on the view function supplied in the JSON request. For example:

```
POST http://couchdb:5984/recipes/_temp_view
Content-Type: application/json

{
   "map" : "function(doc) { if (doc.value > 9995) { emit(null, doc.value); } }"
}
```

The resulting JSON response is the result from the execution of the temporary view:

```
{
   "total_rows" : 3,
   "rows" : [
      {
         "value" : 9998.41913029012,
         "id" : "05361cc6aa42033878acc1bacb1f39c2",
         "key" : null
      },
      {
         "value" : 9998.94149934853,
         "id" : "1f443f471e5929dd7b252417625ed170",
         "key" : null
      },
      {
         "value" : 9998.01511339154,
         "id" : "1f443f471e5929dd7b252417629c102b",
         "key" : null
      }
   ],
   "offset" : 0
}
```

The arguments also available to standard view requests also apply to temporary views, but the execution of the view may take some time as it relies on being executed at the time of the request. In addition to the time taken, they are also computationally very expensive to produce. You should use a defined view if you want to achieve the best performance.

For more information, see ???.

5.11. `POST /db/_purge`

Method	`POST /db/_purge`
Request	JSON of the document IDs/revisions to be purged
Response	JSON structure with purged documents and purge sequence
Admin Privileges Required	no

A database purge permanently removes the references to deleted documents from the database. Deleting a document within CouchDB does not actually remove the documen from the database, instead, the document is marked as a deleted (and a new revision is created). This is to ensure that deleted documents are replicated to other databases as having been deleted. This also means that you can check the status of a document and identify that the document has been deleted.

The purge operation removes the refernces to the deleted documents from the database. The purging of old documents is not replicated to other databases. If you are replicating between databases and have deleted a large number of documents you should run purge on each database.

> **Note**
>
> Purging documents does not remove the space used by them on disk. To reclaim disk space, you should run a database compact (see Section 5.5, “`POST /db/_compact`”, and compact views (see Section 5.6, “`POST /db/_compact/design-doc`”).

To perform a purge operation you must send a request including the JSON of the document IDs that you want to purge. For example:

```
POST http://couchdb:5984/recipes/_purge
Content-Type: application/json

{
  "FishStew" : [
    "17-b3eb5ac6fbaef4428d712e66483dcb79"
    ]
}
```

The format of the request must include the document ID and one or more revisions that must be purged.

The response will contain the purge sequence number, and a list of the document IDs and revisions successfully purged.

```
{
   "purged" : {
      "FishStew" : [
         "17-b3eb5ac6fbaef4428d712e66483dcb79"
      ]
   },
   "purge_seq" : 11
}
```

5.11.1. Updating Indexes

The number of purges on a database is tracked using a purge sequence. This is used by the view indexer to optimize the updating of views that contain the purged documents.

When the indexer identifies that the purge sequence on a database has changed, it compares the purge sequence of the database with that stored in the view index. If the difference between the stored sequence and database is sequence is only 1, then the indexer uses a cached list of the most recently purged documents, and then removes these documents from the index individually. This prevents completely rebuilding the index from scratch.

If the difference between the stored sequence number and current database sequence is greater than 1, then the view index is entirely rebuilt. This is an expensive operation as every document in the database must be examined.

5.12. `GET /db/_all_docs`

Method	`GET /db/_all_docs`	
Request	None	
Response	JSON object containing document information, ordered by the document ID	
Admin Privileges Required	no	
Query Arguments	**Argument**	`descending`
	Description	Return the documents in descending by key order
	Optional	yes
	Type	boolean
	Default	false
	Argument	`endkey`
	Description	Stop returning records when the specified key is reached
	Optional	yes
	Type	string
	Argument	`endkey_docid`
	Description	Stop returning records when the specified document ID is reached
	Optional	yes
	Type	string
	Argument	`group`

	Description	Group the results using the reduce function to a group or single row
	Optional	yes
	Type	boolean
	Default	false
	Argument	`group_level`
	Description	Specify the group level to be used
	Optional	yes
	Type	numeric
	Argument	`include_docs`
	Description	Include the full content of the documents in the return
	Optional	yes
	Type	boolean
	Default	false
	Argument	`inclusive_end`
	Description	Specifies whether the specified end key should be included in the result
	Optional	yes
	Type	boolean
	Default	true
	Argument	`key`
	Description	Return only documents that match the specified key
	Optional	yes
	Type	string
	Argument	`limit`
	Description	Limit the number of the returned documents to the specified number
	Optional	yes
	Type	numeric
	Argument	`reduce`
	Description	Use the reduction function
	Optional	yes

	Type	boolean
	Default	true
	Argument	`skip`
	Description	Skip this number of records before starting to return the results
	Optional	yes
	Type	numeric
	Default	0
	Argument	`stale`
	Description	Allow the results from a stale view to be used
	Optional	yes
	Type	string
	Default	
	Supported Values	
	`ok`	Allow stale views
	Argument	`startkey`
	Description	Return records starting with the specified key
	Optional	yes
	Type	string
	Argument	`startkey_docid`
	Description	Return records starting with the specified document ID
	Optional	yes
	Type	string
	Argument	`update_seq`
	Description	Include the update sequence in the generated results
	Optional	yes
	Type	boolean
	Default	false

Returns a JSON structure of all of the documents in a given database. The information is returned as a JSON structure containing meta information about the return structure, and the list documents and basic contents, consisting the ID, revision and key. The key is generated from the document ID.

Table 5.7. All Database Documents

Field	Description
`offset`	Offset where the document list started
`rows [array]`	Array of document object
`total_rows`	Number of documents in the database/view
`update_seq` (optional)	Current update sequence for the database

By default the information returned contains only the document ID and revision. For example, the request:

```
GET http://couchdb:5984/recipes/_all_docs
Accept: application/json
```

Returns the following structure:

```
{
   "total_rows" : 18386,
   "rows" : [
      {
         "value" : {
            "rev" : "1-bc0d5aed1e339b1cc1f29578f3220a45"
         },
         "id" : "Aberffrawcake",
         "key" : "Aberffrawcake"
      },
      {
         "value" : {
            "rev" : "3-68a20c89a5e70357c20148f8e82ca331"
         },
         "id" : "Adukiandorangecasserole-microwave",
         "key" : "Adukiandorangecasserole-microwave"
      },
      {
         "value" : {
            "rev" : "3-9b2851ed9b6f655cc4eb087808406c60"
         },
         "id" : "Aioli-garlicmayonnaise",
         "key" : "Aioli-garlicmayonnaise"
      },
      ...
         ],
   "offset" : 0
}
```

The information is returned in the form of a temporary view of all the database documents, with the returned key consisting of the ID of the document. The remainder of the interface is therefore identical to the View query arguments and their behavior.

5.13. `POST /db/_all_docs`

Method	`POST /db/_all_docs`
Request	JSON of the document IDs you want included
Response	JSON of the returned view
Admin Privileges Required	no

The `POST` to `_all_docs` allows to specify multiple keys to be selected from the database. This enables you to request multiple documents in a single request, in place of multiple Section 6.2, “`GET /db/doc`” requests.

The request body should contain a list of the keys to be returned as an array to a `keys` object. For example:

```
POST http://couchdb:5984/recipes/_all_docs
User-Agent: MyApp/0.1 libwww-perl/5.837

{
   "keys" : [
      "Zingylemontart",
      "Yogurtraita"
   ]
}
```

The return JSON is the all documents structure, but with only the selected keys in the output:

```
{
   "total_rows" : 2666,
   "rows" : [
      {
         "value" : {
            "rev" : "1-a3544d296de19e6f5b932ea77d886942"
         },
         "id" : "Zingylemontart",
         "key" : "Zingylemontart"
      },
      {
         "value" : {
            "rev" : "1-91635098bfe7d40197a1b98d7ee085fc"
         },
         "id" : "Yogurtraita",
         "key" : "Yogurtraita"
      }
   ],
   "offset" : 0
}
```

5.14. `POST /db/_missing_revs`

Method	`POST /db/_missing_revs`
Request	JSON list of document revisions
Response	JSON of missing revisions
Admin Privileges Required	no

5.15. `POST /db/_revs_diff`

Method	`POST /db/_revs_diff`
Request	JSON list of document and revisions
Response	JSON list of differences from supplied document/revision list
Admin Privileges Required	no

5.16. `GET /db/_security`

Method	`GET /db/_security`
Request	None
Response	JSON of the security object
Admin Privileges Required	no

Gets the current secrity object from the specified database. The security object consists of two compulsory elements, `admins` and `readers`, which are used to specify the list of users and/or roles that have admin and reader rights to the database respectively. Any additional fields in the security object are optional. The entire security object is made available to validation and other internal functions so that the database can control and limit functionality.

To get the existing security object you would send the following request:

```
{
   "admins" : {
      "roles" : [],
      "names" : [
         "mc",
         "slp"
      ]
   },
   "readers" : {
      "roles" : [],
      "names" : [
         "tim",
         "brian"
      ]
   }
}
```

Table 5.8. Security Object

Field	Description
`admins`	Roles/Users with admin privileges
`roles [array]`	List of roles with parent privilege
`users [array]`	List of users with parent privilege
`readers`	Roles/Users with reader privileges
`roles [array]`	List of roles with parent privilege
`users [array]`	List of users with parent privilege

Note

If the security object for a database has never beent set, then the value returned will be empty.

5.17. `PUT /db/_security`

Method	`PUT /db/_security`
Request	JSON specifying the admin and user security for the database
Response	JSON status message
Admin Privileges Required	no

Sets the security object for the given database.For example, to set the security object for the `recipes` database:

```
PUT http://couchdb:5984/recipes/_security
Content-Type: application/json

{
   "admins" : {
      "roles" : [],
      "names" : [
         "mc",
         "slp"
      ]
   },
   "readers" : {
      "roles" : [],
      "names" : [
         "tim",
         "brian"
      ]
   }
}
```

If the setting was successful, a JSON status object will be returned:

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

5.18. GET /db/_revs_limit

Method	GET /db/_revs_limit
Request	None
Response	The current revision limit setting
Admin Privileges Required	no

Gets the current revs_limit (revision limit) setting.

For example to get the current limit:

```
GET http://couchdb:5984/recipes/_revs_limit
Content-Type: application/json
```

The returned information is the current setting as a numerical scalar:

```
1000
```

5.19. PUT /db/_revs_limit

Method	PUT /db/_revs_limit
Request	A scalar integer of the revision limit setting
Response	Confirmation of setting of the revision limit
Admin Privileges Required	no

Sets the maximum number of document revisions that will be tracked by CouchDB, even after compaction has occurred. You can set the revision limit on a database by using PUT with a scalar integer of the limit that you want to set as the request body.

For example to set the revs limit to 100 for the recipes database:

```
PUT http://couchdb:5984/recipes/_revs_limit
Content-Type: application/json

100
```

If the setting was successful, a JSON status object will be returned:

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

Chapter 6. CouchDB API Server Document Methods

The CouchDB API Server Document methods detail how to create, read, update and delete documents within a database.

A list of the available methods and URL paths are provided below:

Table 6.1. Document API Calls

Method	Path	Description
`POST`	`/db`	Create a new document
`GET`	`/db/doc`	Returns the latest revision of the document
`HEAD`	`/db/doc`	Returns bare information in the HTTP Headers for the document
`PUT`	`/db/doc`	Inserts a new document, or new version of an existing document
`DELETE`	`/db/doc`	Deletes the document
`COPY`	`/db/doc`	Copies the document
`GET`	`/db/doc/attachment`	Gets the attachment of a document
`PUT`	`/db/doc/attachment`	Adds an attachment of a document
`DELETE`	`/db/doc/attachment`	Deletes an attachment of a document

6.1. `POST /db`

Method	`POST /db`	
Request	JSON of the new document	
Response	JSON with the committed document information	
Admin Privileges Required	no	
Query Arguments	**Argument**	`batch`
	Description	Allow document store request to be batched with others
	Optional	yes
	Type	string
	Supported Values	
	`ok`	Enable
Return Codes		
201	Document has been created successfully	
409	Conflict - a document with the specified document ID already exists	

Create a new document in the specified database, using the supplied JSON document structure. If the JSON structure includes the `_id` field, then the document will be created with the specified document ID. If the `_id` field is not specified, a new unique ID will be generated.

For example, you can generate a new document with a generated UUID using the following request:

```
POST http://couchdb:5984/recipes/
Content-Type: application/json

{
   "servings" : 4,
   "subtitle" : "Delicious with fresh bread",
   "title" : "Fish Stew"
}
```

The return JSON will specify the automatically enerated ID and revision information:

```
{
   "id" : "64575eef70ab90a2b8d55fc09e00440d",
   "ok" : true,
   "rev" : "1-9c65296036141e575d32ba9c034dd3ee"
}
```

6.1.1. Specifying the Document ID

The document ID can be specified by including the `_id` field in the JSON of the submitted record. The following request will create the same document with the ID `FishStew`:

```
POST http://couchdb:5984/recipes/
Content-Type: application/json

{
   "_id" : "FishStew",
   "servings" : 4,
   "subtitle" : "Delicious with fresh bread",
   "title" : "Fish Stew"
}
```

The structure of the submitted document is as shown in the table below:

Table 6.2. CouchDB Document

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)

In either case, the returned JSON will specify the document ID, revision ID, and status message:

```
{
   "id" : "FishStew",
   "ok" : true,
   "rev" : "1-9c65296036141e575d32ba9c034dd3ee"
}
```

6.1.2. Batch Mode Writes

You can write documents to the database at a higher rate by using the batch option. This collects document writes together in memory (on a user-by-user basis) before they are committed to disk. This increases the risk of the documents not being stored in the event of a failure, since the documents are not written to disk immediately.

To use the batched mode, append the `batch=ok` query argument to the URL of the `PUT` or `POST` request. The CouchDB server will respond with a 202 HTTP response code immediately.

6.1.3. Including Attachments

You can include one or more attachments with a given document by incorporating the attachment information within the JSON of the document. This provides a simpler alternative to loading documents with attachments than making a separate call (see Section 6.8, “`PUT /db/doc/attachment`”).

Table 6.3. Document with Attachments

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_attachments` (optional)	Document Attachment
`filename`	Attachment information
`content_type`	MIME Content type string
`data`	File attachment content, Base64 encoded

The `filename` will be the attachment name. For example, when sending the JSON structure below:

```
{
   "_id" : "FishStew",
   "servings" : 4,
   "subtitle" : "Delicious with fresh bread",
   "title" : "Fish Stew"
   "_attachments" : {
      "styling.css" : {
         "content-type" : "text/css",
         "data" : "cCB7IGZvbnQtc2l6ZTogMTJwdDsgfQo=",
         },
   },
}
```

The attachment `styling.css` can be accessed using `/recipes/FishStew/styling.css`. For more information on attachments, see Section 6.7, "`GET /db/doc/attachment`".

The document data embedded in to the structure must be encoded using base64.

6.2. `GET /db/doc`

Method	`GET /db/doc`	
Request	None	
Response	Returns the JSON for the document	
Admin Privileges Required	no	
Query Arguments	**Argument**	`conflicts`
	Description	Returns the conflict tree for the document.
	Optional	yes
	Type	boolean
	Default	false
	Supported Values	
	`true`	Includes the revisions
	Argument	`rev`
	Description	Specify the revision to return
	Optional	yes
	Type	string

	Supported Values	
	`true`	Includes the revisions
	Argument	`revs`
	Description	Return a list of the revisions for the document
	Optional	yes
	Type	boolean
	Argument	`revs_info`
	Description	Return a list of detailed revision information for the document
	Optional	yes
	Type	boolean
	Supported Values	
	`true`	Includes the revisions
Return Codes		
201	Document created	
400	The format of the request or revision was invalid	
404	The specified document or revision cannot be found, or has been deleted	
409	Conflict - a document with the specified document ID already exists	

Returns the specified `doc` from the specified `db`. For example, to retrieve the document with the id `FishStew` you would send the following request:

```
GET http://couchdb:5984/recipes/FishStew
Content-Type: application/json
Accept: application/json
```

The returned JSON is the JSON of the document, including the document ID and revision number:

```
{
   "_id" : "FishStew",
   "_rev" : "3-a1a9b39ee3cc39181b796a69cb48521c",
   "servings" : 4,
   "subtitle" : "Delicious with a green salad",
   "title" : "Irish Fish Stew"
}
```

Unless you request a specific revision, the latest revision of the document will always be returned.

6.2.1. Attachments

If the document includes attachments, then the returned structure will contain a summary of the attachments associatd with the document, but not the attachment data itself.

The JSON for the returned document will include the `_attachments` field, with one or more attachment definitions. For example:

```
{
   "_id" : "FishStew",
   "servings" : 4,
   "subtitle" : "Delicious with fresh bread",
   "title" : "Fish Stew"
   "_attachments" : {
      "styling.css" : {
         "stub" : true,
         "content-type" : "text/css",
         "length" : 783426,
         },
   },
}
```

The format of the returned JSON is shown in the table below:

Table 6.4. Returned Document with Attachments

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_attachments` (optional)	Document Attachment
`filename`	Attachment
`content_type`	MIME Content type string
`length`	Length (bytes) of the attachment data
`revpos`	Revision where this attachment exists
`stub`	Indicates whether the attachment is a stub

6.2.2. Getting a List of Revisions

You can obtain a list of the revisions for a given document by adding the `revs=true` parameter to the request URL. For example:

```
GET http://couchdb:5984/recipes/FishStew?revs=true
Accept: application/json
```

The returned JSON structure includes the original document, including a `_revisions` structure that includes the revision information:

```
{
   "servings" : 4,
   "subtitle" : "Delicious with a green salad",
   "_id" : "FishStew",
   "title" : "Irish Fish Stew",
   "_revisions" : {
      "ids" : [
         "a1a9b39ee3cc39181b796a69cb48521c",
         "7c4740b4dcf26683e941d6641c00c39d",
         "9c65296036141e575d32ba9c034dd3ee"
      ],
      "start" : 3
   },
   "_rev" : "3-a1a9b39ee3cc39181b796a69cb48521c"
}
```

Table 6.5. Returned CouchDB Document with Revision Info

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_revisions`	CouchDB Document Revisions

ids [array]	Array of valid revision IDs, in reverse order (latest first)
start	Prefix number for the latest revision

6.2.3. Obtaining an Extended Revision History

You can get additional information about the revisions for a given document by supplying the `revs_info` argument to the query:

```
GET http://couchdb:5984/recipes/FishStew?revs_info=true
Accept: application/json
```

This returns extended revision information, including the availability and status of each revision:

```
{
   "servings" : 4,
   "subtitle" : "Delicious with a green salad",
   "_id" : "FishStew",
   "_revs_info" : [
      {
         "status" : "available",
         "rev" : "3-a1a9b39ee3cc39181b796a69cb48521c"
      },
      {
         "status" : "available",
         "rev" : "2-7c4740b4dcf26683e941d6641c00c39d"
      },
      {
         "status" : "available",
         "rev" : "1-9c65296036141e575d32ba9c034dd3ee"
      }
   ],
   "title" : "Irish Fish Stew",
   "_rev" : "3-a1a9b39ee3cc39181b796a69cb48521c"
}
```

Table 6.6. Returned CouchDB Document with Detailed Revision Info

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_revs_info` [array]	CouchDB Document Extended Revision Info
rev	Full revision string
status	Status of the revision

6.2.4. Obtaining a Specific Revision

To get a specific revision, use the `rev` argument to the request, and specify the full revision number:

```
GET http://couchdb:5984/recipes/FishStew?rev=2-7c4740b4dcf26683e941d6641c00c39d
Accept: application/json
```

The specified revision of the document will be returned, including a `_rev` field specifying the revision that was requested:

```
{
   "_id" : "FishStew",
   "_rev" : "2-7c4740b4dcf26683e941d6641c00c39d",
   "servings" : 4,
   "subtitle" : "Delicious with a green salad",
   "title" : "Fish Stew"
}
```

6.3. HEAD /db/doc

Method	HEAD /db/doc

Request	None	
Response	None	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Specify the revision to return
	Optional	yes
	Type	string
	Argument	`revs`
	Description	Return a list of the revisions for the document
	Optional	yes
	Type	boolean
	Argument	`revs_info`
	Description	Return a list of detailed revision information for the document
	Optional	yes
	Type	boolean
Return Codes		
404	The specified document or revision cannot be found, or has been deleted	

Returns the HTTP Headers containing a minimal amount of information about the specified document. The method supports the same query arguments as the `GET` method, but only the header information (including document size, and the revision as an ETag), is returned. For example, a simple `HEAD` request:

```
HEAD http://couchdb:5984/recipes/FishStew
Content-Type: application/json
```

Returns the following HTTP Headers:

```
HTTP/1.1 200 OK
Server: CouchDB/1.0.1 (Erlang OTP/R13B)
Etag: "7-a19a1a5ecd946dad70e85233ba039ab2"
Date: Fri, 05 Nov 2010 14:54:43 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 136
Cache-Control: must-revalidate
```

The `Etag` header shows the current revision for the requested document, and the `Content-Length` specifies the length of the data, if the document were requested in full.

Adding any of the query arguments (as supported by `GET` method), then the resulting HTTP Headers will correspond to what would be returned. Note that the current revision is not returned when the `refs_info` argument is used. For example:

```
HTTP/1.1 200 OK
Server: CouchDB/1.0.1 (Erlang OTP/R13B)
Date: Fri, 05 Nov 2010 14:57:16 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 609
Cache-Control: must-revalidate
```

6.4. PUT /db/doc

Method	PUT /db/doc	
Request	JSON of the new document, or updated version of the existed document	
Response	JSON of the document ID and revision	
Admin Privileges Required	no	
Query Arguments	**Argument**	batch
	Description	Allow document store request to be batched with others
	Optional	yes
	Type	string
	Supported Values	
	ok	Enable
HTTP Headers	**Header**	If-Match
	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
201	Document has been created successfully	
202	Document accepted for writing (batch mode)	

The PUT method creates a new named document, or creates a new revision of the existing document. Unlike the POST method, you must specify the document ID in the request URL.

For example, to create the docment FishStew, you would send the following request:

```
PUT http://couchdb:5984/recipes/FishStew
Content-Type: application/json

{
  "servings" : 4,
  "subtitle" : "Delicious with fresh bread",
  "title" : "Fish Stew"
}
```

The return type is JSON of the status, document ID,and revision number:

```
{
   "id" : "FishStew",
   "ok" : true,
   "rev" : "1-9c65296036141e575d32ba9c034dd3ee"
}
```

6.4.1. Updating an Existing Document

To update an existing document you must specify the current revision number within the _rev parameter. For example:

```
PUT http://couchdb:5984/recipes/FishStew
Content-Type: application/json

{
  "_rev" : "1-9c65296036141e575d32ba9c034dd3ee",
  "servings" : 4,
  "subtitle" : "Delicious with fresh salad",
  "title" : "Fish Stew"
}
```

Alternatively, you can supply the current revision number in the `If-Match` HTTP header of the request. For example:

```
PUT http://couchdb:5984/recipes/FishStew
If-Match: 2-d953b18035b76f2a5b1d1d93f25d3aea
Content-Type: application/json

{
   "servings" : 4,
   "subtitle" : "Delicious with fresh salad",
   "title" : "Fish Stew"
}
```

The JSON returned will include the updated revision number:

```
{
   "id" : "FishStew99",
   "ok" : true,
   "rev" : "2-d953b18035b76f2a5b1d1d93f25d3aea"
}
```

For information on batched writes, which can provide improved performance, see Section 6.1.2, “Batch Mode Writes”.

6.5. `DELETE /db/doc`

Method	`DELETE /db/doc`	
Request	None	
Response	JSON of the deleted revision	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes
	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
409	Revision is missing, invalid or not the latest	

Deletes the specified document from the database. You must supply the current (latest) revision, either by using the `rev` parameter to specify the revision:

```
DELETE http://couchdb:5984/recipes/FishStew?rev=3-a1a9b39ee3cc39181b796a69cb48521c
Content-Type: application/json
```

Alternatively, you can use ETags with the `If-Match` field:

```
DELETE http://couchdb:5984/recipes/FishStew
If-Match: 3-a1a9b39ee3cc39181b796a69cb48521c
Content-Type: application/json
```

The returned JSON contains the document ID, revision and status:

```
{
   "id" : "FishStew",
   "ok" : true,
   "rev" : "4-2719fd41187c60762ff584761b714cfb"
}
```

Note

Note that deletion of a record increments the revision number. The use of a revision for deletion of the record allows replication of the database to correctly track the deletion in synchronized copies.

6.6. `COPY /db/doc`

Method	`COPY /db/doc`	
Request	None	
Response	JSON of the new document and revision	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Revision to copy from
	Optional	yes
	Type	string
HTTP Headers	**Header**	`Destination`
	Description	Destination document (and optional revision)
	Optional	no
Return Codes		
201	Document has been copied and created successfully	
409	Conflict (target document already exists)	

The `COPY` command (which is non-standard HTTP) copies an existing document to a new or existing document.

The source document is specified on the request line, with the `Destination` HTTP Header of the request specifying the target document.

6.6.1. Copying a Document

You can copy the latest version of a document to a new document by specifying the current document and target document:

```
COPY http://couchdb:5984/recipes/FishStew
Content-Type: application/json
Destination: IrishFishStew
```

The above request copies the document `FishStew` to the new document `IrishFishStew`. The response is the ID and revision of the new document.

```
{
   "id" : "IrishFishStew",
   "rev" : "1-9c65296036141e575d32ba9c034dd3ee"
}
```

6.6.2. Copying from a Specific Revision

To copy *from* a specific version, use the `rev` argument to the query string:

```
COPY http://couchdb:5984/recipes/FishStew?rev=5-acfd32d233f07cea4b4f37daaacc0082
Content-Type: application/json
Destination: IrishFishStew
```

The new document will be created using the information in the specified revision of the source document.

6.6.3. Copying to an Existing Document

To copy to an existing document, you must specify the current revision string for the target document, using the `rev` parameter to the `Destination` HTTP Header string. For example:

```
COPY http://couchdb:5984/recipes/FishStew
Content-Type: application/json
Destination: IrishFishStew?rev=1-9c65296036141e575d32ba9c034dd3ee
```

The return value will be the new revision of the copied document:

```
{
   "id" : "IrishFishStew",
   "rev" : "2-55b6a1b251902a2c249b667dab1c6692"
}
```

6.7. `GET /db/doc/attachment`

Method	`GET /db/doc/attachment`
Request	None
Response	Returns the document data
Admin Privileges Required	no

Returns the file attachment `attachment` associated with the document `doc`. The raw data of the associated attachment is returned (just as if you were accessing a static file. The returned HTTP `Content-type` will be the same as the content type set when the document attachment was submitted into the database.

6.8. `PUT /db/doc/attachment`

Method	`PUT /db/doc/attachment`	
Request	Raw document data	
Response	JSON document status	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current document revision
	Optional	no
	Type	string
HTTP Headers	**Header**	`Content-Length`

	Description	Length (bytes) of the attachment being uploaded
	Optional	no
	Header	`Content-Type`
	Description	MIME type for the uploaded attachment
	Optional	no
	Header	`If-Match`
	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
201	Attachment has been accepted	

Upload the supplied content as an attachment to the specified document (`doc`). The `attachment` name provided must be a URL encoded string. You must also supply either the `rev` query argument or the `If-Match` HTTP header for validation, and the HTTP headers (to set the attacment content type). The content type is used when the attachment is requested as the corresponding content-type in the returned document header.

For example, you could upload a simple text document using the following request:

```
PUT http://couchdb:5984/recipes/FishStew/basic?rev=8-a94cb7e50ded1e06f943be5bfbddf8ca
Content-Length: 10
Content-Type: text/plain

Roast it
```

Or by using the `If-Match` HTTP header:

```
PUT http://couchdb:5984/recipes/FishStew/basic
If-Match: 8-a94cb7e50ded1e06f943be5bfbddf8ca
Content-Length: 10
Content-Type: text/plain

Roast it
```

The returned JSON contains the new document information:

```
{
   "id" : "FishStew",
   "ok" : true,
   "rev" : "9-247bb19a41bfd9bfdaf5ee6e2e05be74"
}
```

Note

Uploading an attachment updates the corresponding document revision. Revisions are tracked for the parent document, not individual attachments.

6.8.1. Updating an Existing Attachment

Uploading an attachment using an existing attachment name will update the corresponding stored content of the database. Since you must supply the revision information to add an attachment to a document, this serves as validation to update the existing attachment.

6.9. DELETE /db/doc/attachment

Method	`DELETE /db/doc/attachment`	
Request	None	
Response	JSON status	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Revision of the document to be deleted
	Optional	no
	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
200	Attachment deleted successfully	
409	Supplied revision is incorrect or missing	

Deletes the attachment `attachment` to the specified `doc`. You must supply the `rev` argument with the current revision to delete the attachment.

For example to delete the attachment `basic` from the recipe `FishStew`:

```
DELETE http://couchdb:5984/recipes/FishStew/basic?rev=9-247bb19a41bfd9bfdaf5ee6e2e05be74
Content-Type: application/json
```

The returned JSON contains the updated revision information:

```
{
   "id" : "FishStew",
   "ok" : true,
   "rev" : "10-561bf6b1e27615cee83d1f48fa65dd3e"
}
```

Chapter 7. CouchDB API Server Local (non-replicating) Document Methods

The Local (non-replicating) document interface allows you to create local documents that are not replicated to other databases. These documents can be used to hold configuration or other information that is required specifically on the local CouchDB instance.

Local documents have the following limitations:

- Local documents are not replicated to other databases.
- The ID of the local document must be known for the document to accessed. You cannot obtain a list of local documents from the database.
- Local documents are not output by views, or the `_all_docs` view.

Local documents can be used when you want to store configuration or other information for the curent (local) instance of a given database.

A list of the available methods and URL paths are provided below:

Table 7.1. Local (non-replicating) Document API Calls

Method	Path	Description
`GET`	`/db/_local/local-doc`	Returns the latest revision of the non-replicated document
`PUT`	`/db/_local/local-doc`	Inserts a new version of the non-replicated document
`DELETE`	`/db/_local/local-doc`	Deletes the non-replicated document
`COPY`	`/db/_local/local-doc`	Copies the non-replicated document

7.1. `GET /db/_local/local-doc`

Method	`GET /db/_local/local-doc`	
Request	None	
Response	JSON of the returned document	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Specify the revision to return
	Optional	yes
	Type	string
	Supported Values	
	`true`	Includes the revisions
	Argument	`revs`
	Description	Return a list of the revisions for the document

	Optional	yes
	Type	boolean
	Argument	`revs_info`
	Description	Return a list of detailed revision information for the document
	Optional	yes
	Type	boolean
	Supported Values	
	`true`	Includes the revisions
Return Codes		
400	The format of the request or revision was invalid	
404	The specified document or revision cannot be found, or has been deleted	

Gets the specified local document. The semantics are identical to accessing a standard document in the specified database, except that the document is not replicated. See Section 6.2, "`GET /db/doc`".

7.2. `PUT /db/_local/local-doc`

Method	`PUT /db/_local/local-doc`
Request	JSON of the document
Response	JSON with the committed document information
Admin Privileges Required	no
Return Codes	
201	Document has been created successfully

Stores the specified local document. The semantics are identical to storing a standard document in the specified database, except that the document is not replicated. See Section 6.4, "`PUT /db/doc`".

7.3. `DELETE /db/_local/local-doc`

Method	`DELETE /db/_local/local-doc`	
Request	None	
Response	JSON with the deleted document information	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes
	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation

	Optional	yes
Return Codes		
409	Supplied revision is incorrect or missing	

Deletes the specified local document. The semantics are identical to deleting a standard document in the specified database, except that the document is not replicated. See Section 6.5, "`DELETE /db/doc`".

7.4. `COPY /db/_local/local-doc`

Method	`COPY /db/_local/local-doc`	
Request	None	
Response	JSON of the copied document	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Revision to copy from
	Optional	yes
	Type	string
HTTP Headers	**Header**	`Destination`
	Description	Destination document (and optional revision)
	Optional	no

Copies the specified local document. The semantics are identical to copying a standard document in the specified database, except that the document is not replicated. See Section 6.6, "`COPY /db/doc`".

Chapter 8. CouchDB API Server Design Document Methods

In CouchDB, design documents provide the main interface for building a CouchDB application. The design document defines the views used to extract information from CouchDB through one or more views. Design documents are created within your CouchDB instance in the same way as you create database documents, but the content and definition of the documents is different. Design Documents are named using an ID defined with the design document URL path, and this URL can then be used to access the database contents.

Views and lists operate together to provide automated (and formatted) output from your database.

A list of the available methods and URL paths are provided below:

Table 8.1. Design Document API Calls

Method	Path	Description
GET	/db/_design/design-doc	Returns the latest revision of the design document
PUT	/db/_design/design-doc	Creates or updates a design document
DELETE	/db/_design/design-doc	Deletes the design document
COPY	/db/_design/design-doc	Copies the design document
GET	/db/_design/design-doc/_info	Returns information about the design document
GET	/db/_design/design-doc/_list/list-name/other-design-doc/view-name	Invokes the list handler to translate the given view results
POST	/db/_design/design-doc/_list/list-name/other-design-doc/view-name	Invokes the list handler to translate the given view results for certain documents
GET	/db/_design/design-doc/_list/list-name/view-name	Invokes the list handler to translate the given view results
POST	/db/_design/design-doc/_list/list-name/view-name	Invokes the list handler to translate the given view results for certain documents
ALL	/db/_design/design-doc/_rewrite/rewrite-name/anything	Invokes the URL rewrite handler and processes the request after rewriting
GET	/db/_design/design-doc/_show/show-name	Invokes the show handler without a document
GET	/db/_design/design-doc/_show/show-name/doc	Invokes the show handler for the given document
POST	/db/_design/design-doc/_update/update-name	Invokes the update handler without a document ID
PUT	/db/_design/design-doc/_update/update-name/doc	Invokes the update handler with a specific document ID
GET	/db/_design/design-doc/_view/view-name	Returns results of the view
POST	/db/_design/design-doc/_view/view-name	Returns certain rows from the view

Method	Path	Description
GET	/db/_design/design-doc/attachment	Gets an attachment of the design document
PUT	/db/_design/design-doc/attachment	Inserts an attachment to the design document
DELETE	/db/_design/design-doc/attachment	Deletes an attachment from the design document

8.1. GET /db/_design/design-doc

Method	GET /db/_design/design-doc	
Request	None	
Response	JSON of the existing design document	
Admin Privileges Required	no	
Query Arguments	Argument	rev
	Description	Specify the revision to return
	Optional	yes
	Type	string
	Argument	revs
	Description	Return a list of the revisions for the document
	Optional	yes
	Type	boolean
	Supported Values	
	true	Includes the revisions
	Argument	revs_info
	Description	Return a list of detailed revision information for the document
	Optional	yes
	Type	boolean
	Supported Values	
	true	Includes the revisions

Returns the specified design document, design-doc from the specified db. For example, to retrieve the design document recipes you would send the following request:

```
GET http://couchdb:5984/recipes/_design/recipes
Content-Type: application/json
```

The returned string will be the JSON of the design document:

```
{
   "_id" : "_design/recipes",
   "_rev" : "5-39f56a392b86bbee57e2138921346406"
   "language" : "javascript",
   "views" : {
      "by_recipe" : {
         "map" : "function(doc) { if (doc.title != null) emit(doc.title, doc) }"
      },
   },
}
```

A list of the revisions can be obtained by using the `revs` query argument, or an extended list of revisions using the `revs_info` query argument. This operates in the same way as for other documents. Fur further examples, see Section 6.2, “GET /db/doc”.

8.2. PUT /db/_design/design-doc

Method	`PUT /db/_design/design-doc`
Request	JSON of the design document
Response	JSON status
Admin Privileges Required	no

Upload the specified design document, `design-doc`, to the specified database. The design document should follow the definition of a design document, as summarised in the following table.

Table 8.2. Design Document

Field	Description
`_id`	Design Document ID
`_rev`	Design Document Revision
`views`	View
`viewname`	View Definition
`map`	Map Function for View
`reduce` (optional)	Reduce Function for View

For more information on writing views, see ???.

8.3. DELETE /db/_design/design-doc

Method	`DELETE /db/_design/design-doc`	
Request	None	
Response	JSON of deleted design document	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes
	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation

	Optional	yes
Return Codes		
409	Supplied revision is incorrect or missing	

Delete an existing design document. Deleting a design document also deletes all of the associated view indexes, and recovers the corresponding space on disk for the indexes in question.

To delete, you must specify the current revision of the design document using the `rev` query argument.

For example:

```
DELETE http://couchdb:5984/recipes/_design/recipes?rev=2-ac58d589b37d01c00f45a4418c5a15a8
Content-Type: application/json
```

The response contains the delete document ID and revision:

```
{
   "id" : "recipe/_design/recipes"
   "ok" : true,
   "rev" : "3-7a05370bff53186cb5d403f861aca154",
}
```

8.4. `COPY /db/_design/design-doc`

Method	`COPY /db/_design/design-doc`	
Request	None	
Response	JSON of the copied document and revision	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Revision to copy from
	Optional	yes
	Type	string
HTTP Headers	**Header**	`Destination`
	Description	Destination document (and optional revision)
	Optional	no

The `COPY` command (non-standard HTTP) copies an existing design document to a new or existing document.

The source design document is specified on the request line, with the `Destination` HTTP Header of the request specifying the target document.

8.4.1. Copying a Design Document

To copy the latest version of a design document to a new document you specify the base document and target document:

```
COPY http://couchdb:5984/recipes/_design/recipes
Content-Type: application/json
Destination: /recipes/_design/recipelist
```

The above request copies the design document `recipes` to the new design document `recipelist`. The response is the ID and revision of the new document.

```
{
   "id" : "recipes/_design/recipelist"
   "rev" : "1-9c65296036141e575d32ba9c034dd3ee",
}
```

Note

Copying a design document does automatically reconstruct the view indexes. These will be recreated, as with other views, the first time the new view is accessed.

8.4.2. Copying from a Specific Revision

To copy *from* a specific version, use the `rev` argument to the query string:

```
COPY http://couchdb:5984/recipes/_design/recipes?rev=1-e23b9e942c19e9fb10ff1fde2e50e0f5
Content-Type: application/json
Destination: recipes/_design/recipelist
```

The new design document will be created using the specified revision of the source document.

8.4.3. Copying to an Existing Design Document

To copy to an existing document, you must specify the current revision string for the target document, using the `rev` parameter to the `Destination` HTTP Header string. For example:

```
COPY http://couchdb:5984/recipes/_design/recipes
Content-Type: application/json
Destination: recipes/_design/recipelist?rev=1-9c65296036141e575d32ba9c034dd3ee
```

The return value will be the new revision of the copied document:

```
{
   "id" : "recipes/_design/recipes"
   "rev" : "2-55b6a1b251902a2c249b667dab1c6692",
}
```

8.5. `GET /db/_design/design-doc/attachment`

Method	`GET /db/_design/design-doc/attachment`
Request	None
Response	Document content
Admin Privileges Required	no

Returns the file attachment `attachment` associated with the design document `/_design_/design-doc`. The raw data of the associated attachment is returned (just as if you were accessing a static file. The returned HTTP `Content-type` will be the same as the content type set when the document attachment was submitted into the database.

8.6. `PUT /db/_design/design-doc/attachment`

Method	`PUT /db/_design/design-doc/attachment`	
Request	JSON of the design document	
Response	JSON status statement	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes

	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation
	Optional	yes

Upload the supplied content as an attachment to the specified design document (`/_design/design-doc`). The `attachment` name provided must be a URL encoded string. You must also supply either the `rev` query argument or the `If-Match` HTTP header for validation, and the HTTP headers (to set the attacment content type). The content type is used when the attachment is requested as the corresponding content-type in the returned document header.

For example, you could upload a simple text document using the following request:

```
PUT http://couchdb:5984/recipes/_design/recipes/view.css?rev=7-f7114d4d81124b223283f3e89eee043e
Content-Length: 39
Content-Type: text/plain

div.recipetitle {
font-weight: bold;
}
```

Or by using the `If-Match` HTTP header:

```
PUT http://couchdb:5984/recipes/FishStew/basic
If-Match: 7-f7114d4d81124b223283f3e89eee043e
Content-Length: 39
Content-Type: text/plain

div.recipetitle {
font-weight: bold;
}
```

The returned JSON contains the new document information:

```
{
   "id" : "_design/recipes"
   "ok" : true,
   "rev" : "8-cb2b7d94eeac76782a02396ba70dfbf5",
}
```

Note

Uploading an attachment updates the corresponding document revision. Revisions are tracked for the parent document, not individual attachments.

8.7. `DELETE /db/_design/design-doc/attachment`

Method	`DELETE /db/_design/design-doc/attachment`	
Request	None	
Response	JSON of the deleted revision	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes
	Type	string
HTTP Headers	**Header**	`If-Match`

	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
409	Supplied revision is incorrect or missing	

Deletes the attachment `attachment` to the specified `_design/design-doc`. You must supply the `rev` argument with the current revision to delete the attachment.

For example to delete the attachment `view.css` from the design document `recipes`:

```
DELETE http://couchdb:5984/recipes/_design/recipes/view.css?rev=9-3db559f13a845c7751d407404cdeaa4a
```

The returned JSON contains the updated revision information for the parent document:

```
{
   "id" : "_design/recipes"
   "ok" : true,
   "rev" : "10-f3b15bb408961f8dcc3d86c7d3b54c4c",
}
```

8.8. `GET /db/_design/design-doc/_info`

Method	`GET /db/_design/design-doc/_info`
Request	None
Response	JSON of the design document information
Admin Privileges Required	no

Obtains information about a given design document, including the index, index size and current status of the design document and associated index information.

For example, to get the information for the `recipes` design document:

```
GET http://couchdb:5984/recipes/_design/recipes/_info
Content-Type: application/json
```

This returns the following JSON structure:

```
{
   "name" : "recipes"
   "view_index" : {
      "compact_running" : false,
      "updater_running" : false,
      "language" : "javascript",
      "purge_seq" : 10,
      "waiting_commit" : false,
      "waiting_clients" : 0,
      "signature" : "fc65594ee76087a3b8c726caf5b40687",
      "update_seq" : 375031,
      "disk_size" : 16491
   },
}
```

The individual fields in the returned JSON structure are detailed in Table 8.3, “Design Document Info JSON Contents”.

Table 8.3. Design Document Info JSON Contents

Field	Description
`name`	Name/ID of Design Document
`view_index`	View Index

`compact_running`	Indicates whether a compaction routine is currently running on the view
`disk_size`	Size in bytes of the view as stored on disk
`language`	Language for the defined views
`purge_seq`	The purge sequence that has been processed
`signature`	MD5 signature of the views for the design document
`update_seq`	The update sequence of the corresponding database that has been indexed
`updater_running`	Indicates if the view is currently being updated
`waiting_clients`	Number of clients waiting on views from this design document
`waiting_commit`	Indicates if there are outstanding commits to the underlying database that need to processed

8.9. `GET /db/_design/design-doc/_view/view-name`

Method	`GET /db/_design/design-doc/_view/view-name`	
Request	None	
Response	JSON of the documents returned by the view	
Admin Privileges Required	no	
Query Arguments	**Argument**	`descending`
	Description	Return the documents in descending by key order
	Optional	yes
	Type	boolean
	Default	false
	Argument	`endkey`
	Description	Stop returning records when the specified key is reached
	Optional	yes
	Type	string
	Argument	`endkey_docid`
	Description	Stop returning records when the specified document ID is reached
	Optional	yes
	Type	string
	Argument	`group`
	Description	Group the results using the reduce function to a group or single row
	Optional	yes
	Type	boolean

	Default	false
	Argument	`group_level`
	Description	Specify the group level to be used
	Optional	yes
	Type	numeric
	Argument	`include_docs`
	Description	Include the full content of the documents in the return
	Optional	yes
	Type	boolean
	Default	false
	Argument	`inclusive_end`
	Description	Specifies whether the specified end key should be included in the result
	Optional	yes
	Type	boolean
	Default	true
	Argument	`key`
	Description	Return only documents that match the specified key
	Optional	yes
	Type	string
	Argument	`limit`
	Description	Limit the number of the returned documents to the specified number
	Optional	yes
	Type	numeric
	Argument	`reduce`
	Description	Use the reduction function
	Optional	yes
	Type	boolean
	Default	true
	Argument	`skip`

	Description	Skip this number of records before starting to return the results
	Optional	yes
	Type	numeric
	Default	0
	Argument	`stale`
	Description	Allow the results from a stale view to be used
	Optional	yes
	Type	string
	Default	
	Supported Values	
	`ok`	Allow stale views
	Argument	`startkey`
	Description	Return records starting with the specified key
	Optional	yes
	Type	string
	Argument	`startkey_docid`
	Description	Return records starting with the specified document ID
	Optional	yes
	Type	string
	Argument	`update_seq`
	Description	Include the update sequence in the generated results
	Optional	yes
	Type	boolean
	Default	false

Executes the specified `view-name` from the specified `design-doc` design document.

8.9.1. Querying Views and Indexes

The definition of a view within a design document also creates an index based on the key information defined within each view. The production and use of the index significantly increases the speed of access and searching or selecting documents from the view.

However, the index is not updated when new documents are added or modified in the database. Instead, the index is generated or updated, either when the view is first accessed, or when the view is accessed after a document has been updated. In each case, the index is updated before the view query is executed against the database.

View indexes are updated incrementally in the following situations:

- A new document has been added to the database.
- A document has been deleted from the database.
- A document in the database has been updated.

View indexes are rebuilt entirely when the view definition changes. To achieve this, a 'fingerprint' of the view definition is created when the design document is updated. If the fingerprint changes, then the view indexes are entirely rebuilt. This ensures that changes to the view definitions are reflected in the view indexes.

> **Note**
>
> View index rebuilds occur when one view from the same the view group (i.e. all the views defined within a single a design document) has been determined as needing a rebuild. For example, if if you have a design document with different views, and you update the database, all three view indexes within the design document will be updated.

Because the view is updated when it has been queried, it can result in a delay in returned information when the view is accessed, especially if there are a large number of documents in the database and the view index does not exist. There are a number of ways to mitigate, but not completely eliminate, these issues. These include:

- Create the view definition (and associated design documents) on your database before allowing insertion or updates to the documents. If this is allowed while the view is being accessed, the index can be updated incrementally.
- Manually force a view request from the database. You can do this either before users are allowed to use the view, or you can access the view manually after documents are added or updated.
- Use the `/db/_changes` method to monitor for changes to the database and then access the view to force the corresponding view index to be updated. See Section 5.4, “`GET /db/_changes`” for more information.
- Use a monitor with the `update_notification` section of the CouchDB configuration file to monitor for changes to your database, and trigger a view query to force the view to be updated. For more information, see Section 12.5, “Update Notifications”.

None of these can completely eliminate the need for the indexes to be rebuilt or updated when the view is accessed, but they may lessen the effects on end-users of the index update affecting the user experience.

Another alternative is to allow users to access a 'stale' version of the view index, rather than forcing the index to be updated and displaying the updated results. Using a stale view may not return the latest information, but will return the results of the view query using an existing version of the index.

For example, to access the existing stale view `by_recipe` in the `recipes` design document:

```
http://couchdb:5984/recipes/_design/recipes/_view/by_recipe?stale=ok
```

Accessing a stale view:

- Does not trigger a rebuild of the view indexes, even if there have been changes since the last access.
- Returns the current version of the view index, if a current version exists.

- Returns an empty result set if the given view index does exist.

As an alternative, you use the `update_after` value to the `stale` paramater. This causes the view to be returned as a stale view, but for the update process to be triggered after the view information has been returned to the client.

In addition to using stale views, you can also make use of the `update_seq` query argument. Using this query argument generates the view information including the update sequence of the database from which the view was generated. The returned value can be compared this to the current update sequence exposed in the database information (returned by Section 5.1, “`GET /db`”).

8.9.2. Sorting Returned Rows

Each element within the returned array is sorted using native UTF-8 sorting according to the contents of the key portion of the emitted content. The basic order of output is as follows:

- `null`
- `false`
- `true`
- Numbers
- Text (case sensitive, lowercase first)
- Arrays (according to the values of each element, in order)
- Objects (according to the values of keys, in key order)

You can reverse the order of the returned view information by using the `descending` query value set to true. For example, Retrieving the list of recipes using the `by_title` (limited to 5 records) view:

```
{
   "offset" : 0,
   "rows" : [
      {
         "id" : "3-tiersalmonspinachandavocadoterrine",
         "key" : "3-tier salmon, spinach and avocado terrine",
         "value" : [
            null,
            "3-tier salmon, spinach and avocado terrine"
         ]
      },
      {
         "id" : "Aberffrawcake",
         "key" : "Aberffraw cake",
         "value" : [
            null,
            "Aberffraw cake"
         ]
      },
      {
         "id" : "Adukiandorangecasserole-microwave",
         "key" : "Aduki and orange casserole - microwave",
         "value" : [
            null,
            "Aduki and orange casserole - microwave"
         ]
      },
      {
         "id" : "Aioli-garlicmayonnaise",
         "key" : "Aioli - garlic mayonnaise",
         "value" : [
            null,
            "Aioli - garlic mayonnaise"
         ]
      },
      {
         "id" : "Alabamapeanutchicken",
         "key" : "Alabama peanut chicken",
         "value" : [
            null,
            "Alabama peanut chicken"
         ]
      }
   ],
   "total_rows" : 2667
}
```


Requesting the same in descending order will reverse the entire view content. For example the request

```
GET http://couchdb:5984/recipes/_design/recipes/_view/by_title?limit=5&descending=true
Accept: application/json
Content-Type: application/json
```

Returns the last 5 records from the view:

```
{
   "offset" : 0,
   "rows" : [
      {
         "id" : "Zucchiniinagrodolcesweet-sourcourgettes",
         "key" : "Zucchini in agrodolce (sweet-sour courgettes)",
         "value" : [
            null,
            "Zucchini in agrodolce (sweet-sour courgettes)"
         ]
      },
      {
         "id" : "Zingylemontart",
         "key" : "Zingy lemon tart",
         "value" : [
            null,
            "Zingy lemon tart"
         ]
      },
      {
         "id" : "Zestyseafoodavocado",
         "key" : "Zesty seafood avocado",
         "value" : [
            null,
            "Zesty seafood avocado"
         ]
      },
      {
         "id" : "Zabaglione",
         "key" : "Zabaglione",
         "value" : [
            null,
            "Zabaglione"
         ]
      },
      {
         "id" : "Yogurtraita",
         "key" : "Yogurt raita",
         "value" : [
            null,
            "Yogurt raita"
         ]
      }
   ],
   "total_rows" : 2667
}
```

The sorting direction is applied before the filtering applied using the `startkey` and `endkey` query arguments. For example the following query:

```
GET http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient?startkey=%22carrots%22&endkey=%22egg%22
Accept: application/json
Content-Type: application/json
```

Will operate correctly when listing all the matching entries between “carrots” and `egg`. If the order of output is reversed with the `descending` query argument, the view request will return no entries:

```
GET http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient?descending=true&startkey=%22carrots%22&endkey=%22
Accept: application/json
Content-Type: application/json
```

The returned result is empty:

```
{
   "total_rows" : 26453,
   "rows" : [],
   "offset" : 21882
}
```

The results will be empty because the entries in the view are reversed before the key filter is applied, and therefore the `endkey` of “egg” will be seen before the `startkey` of “carrots”, resulting in an empty list.

Instead, you should reverse the values supplied to the `startkey` and `endkey` parameters to match the descending sorting applied to the keys. Changing the previous example to:

```
GET http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient?descending=true&startkey=%22egg%22&endkey=%22carr
Accept: application/json
Content-Type: application/json
```

8.9.3. Specifying Start and End Values

The `startkey` and `endkey` query arguments can be used to specify the range of values to be displayed when querying the view.

Note

The values

8.9.4. Using Limits and Skipping Rows

TBC

8.9.5. View Reduction and Grouping

TBC

8.10. `POST /db/_design/design-doc/_view/view-name`

Method	`POST /db/_design/design-doc/_view/view-name`	
Request	List of keys to be returned from specified view	
Response	JSON of the documents returned by the view	
Admin Privileges Required	no	
Query Arguments	**Argument**	`descending`
	Description	Return the documents in descending by key order
	Optional	yes
	Type	boolean
	Default	false
	Argument	`endkey`
	Description	Stop returning records when the specified key is reached
	Optional	yes
	Type	string
	Argument	`endkey_docid`
	Description	Stop returning records when the specified document ID is reached
	Optional	yes
	Type	string

	Argument	`group`
	Description	Group the results using the reduce function to a group or single row
	Optional	yes
	Type	boolean
	Default	false
	Argument	`group_level`
	Description	Specify the group level to be used
	Optional	yes
	Type	numeric
	Argument	`include_docs`
	Description	Include the full content of the documents in the return
	Optional	yes
	Type	boolean
	Default	false
	Argument	`inclusive_end`
	Description	Specifies whether the specified end key should be included in the result
	Optional	yes
	Type	boolean
	Default	true
	Argument	`key`
	Description	Return only documents that match the specified key
	Optional	yes
	Type	string
	Argument	`limit`
	Description	Limit the number of the returned documents to the specified number
	Optional	yes
	Type	numeric
	Argument	`reduce`
	Description	Use the reduction function

	Optional	yes
	Type	boolean
	Default	true
	Argument	`skip`
	Description	Skip this number of records before starting to return the results
	Optional	yes
	Type	numeric
	Default	0
	Argument	`stale`
	Description	Allow the results from a stale view to be used
	Optional	yes
	Type	string
	Default	
	Supported Values	
	`ok`	Allow stale views
	Argument	`startkey`
	Description	Return records starting with the specified key
	Optional	yes
	Type	string
	Argument	`startkey_docid`
	Description	Return records starting with the specified document ID
	Optional	yes
	Type	string
	Argument	`update_seq`
	Description	Include the update sequence in the generated results
	Optional	yes
	Type	boolean
	Default	false

Executes the specified `view-name` from the specified `design-doc` design document. Unlike the `GET` method for accessing views, the `POST` method supports the specification of explicit keys to be retrieved from the view results.

The remainder of the `POST` view functionality is identical to the Section 8.9, "`GET /db/_design/design-doc/_view/view-name`" fun

For example, the request below will return all the recipes where the key for the view matches either "claret" or "clear apple cider" :

```
POST http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient
Content-Type: application/json

{
   "keys" : [
      "claret",
      "clear apple juice"
   ]
}
```

The returned view data contains the standard view information, but only where the keys match.

```
{
   "total_rows" : 26484,
   "rows" : [
      {
         "value" : [
            "Scotch collops"
         ],
         "id" : "Scotchcollops",
         "key" : "claret"
      },
      {
         "value" : [
            "Stand pie"
         ],
         "id" : "Standpie",
         "key" : "clear apple juice"
      }
   ],
   "offset" : 6324
}
```

8.10.1. Multi-document Fetching

By combining the `POST` method to a given view with the `include_docs=true` query argument you can obtain multiple documents from a database. The result is more efficient than using multiple Section 6.2, "`GET /db/doc`" requests.

For example, sending the following request for ingredients matching "claret" and "clear apple juice":

```
POST http://couchdb:5984/recipes/_design/recipes/_view/by_ingredient?include_docs=true
Content-Type: application/json

{
   "keys" : [
      "claret",
      "clear apple juice"
   ]
}
```

Returns the full document for each recipe:

```
{
   "offset" : 6324,
   "rows" : [
      {
         "doc" : {
            "_id" : "Scotchcollops",
            "_rev" : "1-bcbdf724f8544c89697a1cbc4b9f0178",
            "cooktime" : "8",
            "ingredients" : [
               {
                  "ingredient" : "onion",
                  "ingredtext" : "onion, peeled and chopped",
                  "meastext" : "1"
               },
            ...
            ],
            "keywords" : [
               "cook method.hob, oven, grill@hob",
               "diet@wheat-free",
               "diet@peanut-free",
               "special collections@classic recipe",
               "cuisine@british traditional",
               "diet@corn-free",
               "diet@citrus-free",
               "special collections@very easy",
               "diet@shellfish-free",
               "main ingredient@meat",
               "occasion@christmas",
               "meal type@main",
               "diet@egg-free",
               "diet@gluten-free"
            ],
            "preptime" : "10",
            "servings" : "4",
            "subtitle" : "This recipe comes from an old recipe book of 1683 called 'The Gentlewoman's Kitchen'. This i
            "title" : "Scotch collops",
            "totaltime" : "18"
         },
         "id" : "Scotchcollops",
         "key" : "claret",
         "value" : [
            "Scotch collops"
         ]
      },
      {
         "doc" : {
            "_id" : "Standpie",
            "_rev" : "1-bff6edf3ca2474a243023f2dad432a5a",
            "cooktime" : "92",
            "ingredients" : [
...            ],
            "keywords" : [
               "diet@dairy-free",
               "diet@peanut-free",
               "special collections@classic recipe",
               "cuisine@british traditional",
               "diet@corn-free",
               "diet@citrus-free",
               "occasion@buffet party",
               "diet@shellfish-free",
               "occasion@picnic",
               "special collections@lunchbox",
               "main ingredient@meat",
               "convenience@serve with salad for complete meal",
               "meal type@main",
               "cook method.hob, oven, grill@hob / oven",
               "diet@cow dairy-free"
            ],
            "preptime" : "30",
            "servings" : "6",
            "subtitle" : "Serve this pie with pickled vegetables and potato salad.",
            "title" : "Stand pie",
            "totaltime" : "437"
         },
         "id" : "Standpie",
         "key" : "clear apple juice",
         "value" : [
            "Stand pie"
         ]
      }
   ],
   "total_rows" : 26484
}
```

8.11. POST /db/_design/design-doc/_show/show-name

Method	`GET /db/_design/design-doc/_show/show-name`	
Request	None	
Response	Returns the result of the show	
Admin Privileges Required	no	
Query Arguments	**Argument**	`details`
	Description	Indicates whether details should be included
	Optional	yes
	Type	string
	Argument	`format`
	Description	Format of the returned information
	Optional	yes
	Type	string

8.12. POST /db/_design/design-doc/_show/show-name/doc

Method	`GET /db/_design/design-doc/_show/show-name/doc`
Request	None
Response	Returns the show for the given document
Admin Privileges Required	no

8.13. GET /db/_design/design-doc/_list/list-name/other-design-doc/view-name

Method	`GET /db/_design/design-doc/_list/list-name/other-design-doc/view-name`
Request	TBC
Response	TBC
Admin Privileges Required	no

8.14. POST /db/_design/design-doc/_list/list-name/other-design-doc/view-name

Method	`POST /db/_design/design-doc/_list/list-name/other-design-doc/view-name`
Request	TBC
Response	TBC
Admin Privileges Required	no

8.15. GET /db/_design/design-doc/_list/list-name/view-name

Method	`GET /db/_design/design-doc/_list/list-name/view-name`
Request	TBC
Response	TBC
Admin Privileges Required	no

8.16. POST /db/_design/design-doc/_list/list-name/view-name

Method	`POST /db/_design/design-doc/_list/list-name/view-name`
Request	TBC
Response	TBC
Admin Privileges Required	no

8.17. PUT /db/_design/design-doc/_update/updatename/doc

Method	`PUT /db/_design/design-doc/_update/update-name/doc`
Request	Document update information
Response	Updated document
Admin Privileges Required	no

8.18. POST /db/_design/design-doc/_update/updatename

Method	`POST /db/_design/design-doc/_update/update-name`
Request	Document update information
Response	Updated document
Admin Privileges Required	no

8.19. ALL /db/_design/design-doc/_rewrite/rewrite-name/anything

Method	`ALL /db/_design/design-doc/_rewrite/rewrite-name/anything`
Request	TBC
Response	TBC
Admin Privileges Required	no

Chapter 9. CouchDB API Server Miscellaneous Methods

The CouchDB Miscellaneous interface provides the basic interface to a CouchDB server for obtaining CouchDB information and getting and setting configuration information.

A list of the available methods and URL paths are provided below:

Table 9.1. Miscellaneous API Calls

Method	Path	Description
GET	/	Get the welcome message and version information
GET	/_active_tasks	Obtain a list of the tasks running in the server
GET	/_all_dbs	Get a list of all the DBs
GET	/_log	Return the server log file
POST	/_replicate	Set or cancel replication
POST	/_restart	Restart the server
GET	/_stats	Return server statistics
GET	/_utils	CouchDB administration interface (Futon)
GET	/_uuids	Get generated UUIDs from the server
GET	/favicon.ico	Get the site icon

9.1. GET /

Method	GET /
Request	None
Response	Welcome message and version
Admin Privileges Required	no
Return Codes	
200	Request completed successfully.

Accessing the root of a CouchDB instance returns meta information about the instance. The response is a JSON structure containing information about the server, including a welcome message and the version of the server.

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

9.2. GET /_active_tasks

Method	GET /_active_tasks
Request	None
Response	List of running tasks, including the task type, name, status and process ID

Admin Privileges Required	no
Return Codes	
200	Request completed successfully.

You can obtain a list of active tasks by using the `/_active_tasks` URL. The result is a JSON array of the currently running tasks, with each task being described with a single object. For example:

```
[
   {
    "pid" : "<0.11599.0>",
    "status" : "Copied 0 of 18369 changes (0%)",
    "task" : "recipes",
    "type" : "Database Compaction"
    }
]
```

The returned structure includes the following fields for each task:

Table 9.2. List of Active Tasks

Field	Description
`tasks [array]`	Active Task
`pid`	Process ID
`status`	Task status message
`task`	Task name
`type`	Operation Type

For operation type, valid values include:

- `Database Compaction`
- `Replication`
- `View Group Compaction`
- `View Group Indexer`

9.3. `GET /_all_dbs`

Method	`GET /_all_dbs`
Request	None
Response	JSON list of DBs
Admin Privileges Required	no
Return Codes	
200	Request completed successfully.

Returns a list of all the databases in the CouchDB instance. For example:

```
GET http://couchdb:5984/_all_dbs
Accept: application/json
```

The return is a JSON array:

```
[
   "_users",
   "contacts",
   "docs",
   "invoices",
   "locations"
]
```

9.4. GET /_log

Method	GET /_log	
Request	None	
Response	Log content	
Admin Privileges Required	no	
Query Arguments	**Argument**	bytes
	Description	Bytes to be returned
	Optional	yes
	Type	numeric
	Default	1000
	Argument	offset
	Description	Offset in bytes where the log tail should be started
	Optional	yes
	Type	numeric
	Default	0
Return Codes		
200	Request completed successfully.	

Gets the CouchDB log, equivalent to accessing the local log file of the corresponding CouchDB instance.

When you request the log, the response is returned as plain (UTF-8) text, with an HTTP `Content-type` header as `text/plain`.

For example, the request:

```
GET http://couchdb:5984/_log
Accept: */*
```

The raw text is returned:

```
[Wed, 27 Oct 2010 10:49:42 GMT] [info] [<0.23338.2>] 192.168.0.2 - - 'PUT' /authdb 401

[Wed, 27 Oct 2010 11:02:19 GMT] [info] [<0.23428.2>] 192.168.0.116 - - 'GET' /recipes/FishStew 200

[Wed, 27 Oct 2010 11:02:19 GMT] [info] [<0.23428.2>] 192.168.0.116 - - 'GET' /_session 200

[Wed, 27 Oct 2010 11:02:19 GMT] [info] [<0.24199.2>] 192.168.0.116 - - 'GET' / 200

[Wed, 27 Oct 2010 13:03:38 GMT] [info] [<0.24207.2>] 192.168.0.116 - - 'GET' /_log?offset=5 200
```

If you want to pick out specific parts of the log information you can use the `bytes` argument, which specifies the number of bytes to be returned, and `offset`, which specifies where the reading of the log should start, counted back from the end. For example, if you use the following request:

```
GET /_log?bytes=500&offset=2000
```

Reading of the log will start at 2000 bytes from the end of the log, and 500 bytes will be shown.

9.5. `POST /_replicate`

Method	`POST /_replicate`
Request	Replication specification
Response	Welcome message and version
Admin Privileges Required	no
Return Codes	
200	Replication request successfully completed
202	Continuous replication request has been accepted
404	Either the source or target DB is not found
500	JSON specification was invalid

Request, configure, or stop, a replication operation.

The specification of the replication request is controlled through the JSON content of the request. The JSON should be an object with the fields defining the source, target and other options. The fields of the JSON request are shown in the table below:

Table 9.3. Replication Settings

Field	**Description**
`cancel` (optional)	Cancels the replication
`continuous` (optional)	Configure the replication to be continuous
`create_target` (optional)	Creates the target database
`doc_ids` (optional)	Array of document IDs to be synchronized
`proxy` (optional)	Address of a proxy server through which replication should occur
`source`	Source database name or URL
`target`	Target database name or URL

9.5.1. Replication Operation

The aim of the replication is that at the end of the process, all active documents on the source database are also in the destination database and all documents that were deleted in the source databases are also deleted (if they exist) on the destination database.

Replication can be described as either push or pull replication:

- *Pull replication* is where the `source` is the remote CouchDB instance, and the `destination` is the local database.

  Pull replication is the most useful solution to use if your source database has a permanent IP address, and your destination (local) database may have a dynamically assigned IP address (for example, through DHCP). This is particularly important if you are replicating to a mobile or other device from a central server.

- *Push replication* is where the `source` is a local database, and `destination` is a remote database.

9.5.2. Specifying the Source and Target Database

You must use the URL specification of the CouchDB database if you want to perform replication in either of the following two situations:

- Replication with a remote database (i.e. another instance of CouchDB on the same host, or a different host)
- Replication with a database that requires authentication

For example, to request replication between a database local to the CouchDB instance to which you send the request, and a remote database you might use the following request:

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
   "source" : "recipes",
   "target" : "http://coucdb-remote:5984/recipes",
}
```

In all cases, the requested databases in the `source` and `target` specification must exist. If they do not, an error will be returned within the JSON object:

```
{
   "error" : "db_not_found"
   "reason" : "could not open http://couchdb-remote:5984/ol1ka/",
}
```

You can create the target database (providing your user credentials allow it) by adding the `create_target` field to the request object:

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
   "create_target" : true
   "source" : "recipes",
   "target" : "http://couchdb-remote:5984/recipes",
}
```

The `create_target` field is not destructive. If the database already exists, the replication proceeds as normal.

9.5.3. Single Replication

You can request replication of a database so that the two databases can be synchronized. By default, the replication process occurs one time and synchronizes the two databases together. For example, you can request a single synchronization between two databases by supplying the `source` and `target` fields within the request JSON content.

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
   "source" : "recipes",
   "target" : "recipes-snapshot",
}
```

In the above example, the databases `recipes` and `recipes-snapshot` will be synchronized. These databases are local to the CouchDB instance where the request was made. The response will be a JSON structure containing the success (or failure) of the synchronization process, and statistics about the process:

```
{
   "ok" : true,
   "history" : [
      {
         "docs_read" : 1000,
         "session_id" : "52c2370f5027043d286daca4de247db0",
         "recorded_seq" : 1000,
         "end_last_seq" : 1000,
         "doc_write_failures" : 0,
         "start_time" : "Thu, 28 Oct 2010 10:24:13 GMT",
         "start_last_seq" : 0,
         "end_time" : "Thu, 28 Oct 2010 10:24:14 GMT",
         "missing_checked" : 0,
         "docs_written" : 1000,
         "missing_found" : 1000
      }
   ],
   "session_id" : "52c2370f5027043d286daca4de247db0",
   "source_last_seq" : 1000
}
```

The structure defines the replication status, as described in the table below:

Table 9.4. Replication Status

Field	Description
`history [array]`	Replication History
`doc_write_failures`	Number of document write failures
`docs_read`	Number of documents read
`docs_written`	Number of documents written to target
`end_last_seq`	Last sequence number in changes stream
`end_time`	Date/Time replication operation completed
`missing_checked`	Number of missing documents checked
`missing_found`	Number of missing documents found
`recorded_seq`	Last recorded sequence number
`session_id`	Session ID for this replication operation
`start_last_seq`	First sequence number in changes stream
`start_time`	Date/Time replication operation started
`ok`	Replication status
`session_id`	Unique session ID
`source_last_seq`	Last sequence number read from source database

9.5.4. Continuous Replication

Synchronization of a database with the previously noted methods happens only once, at the time the replicate request is made. To have the target database permanently replicated from the source, you must set the `continuous` field of the JSON object within the request to true.

With continuous replication changes in the source database are replicated to the target database in perpetuity until you specifically request that replication ceases.

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
   "continuous" : true
   "source" : "recipes",
   "target" : "http://couchdb-remote:5984/recipes",
}
```

Changes will be replicated between the two databases as long as a network connection is available between the two instances.

Note

Two keep two databases synchronized with each other, you need to set replication in both directions; that is, you must replicate from `databasea` to `databaseb`, and separately from `databaseb` to `databasea`.

9.5.5. Canceling Continuous Replication

You can cancel continuous replication by adding the `cancel` field to the JSON request object and setting the value to true. Note that the structure of the request must be identical to the original for the cancelation request to be honoured. For example, if you requested continuous replication, the cancellation request must also contain the `continuous` field.

For example, the replication request:

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
   "source" : "recipes",
   "target" : "http://couchdb-remote:5984/recipes",
   "create_target" : true,
   "continuous" : true
}
```

Must be canceled using the request:

```
POST http://couchdb:5984/_replicate
Content-Type: application/json
Accept: application/json

{
    "cancel" : true,
    "continuous" : true
    "create_target" : true,
    "source" : "recipes",
    "target" : "http://couchdb-remote:5984/recipes",
}
```

Requesting cancellation of a replication that does not exist results in a 404 error.

9.6. `POST /_restart`

Method	`POST /_restart`
Request	None
Response	JSON status message
Admin Privileges Required	yes
Return Codes	
201	Document created successfully.

Restarts the CouchDB instance. You must be authenticated as a user with administration privileges for this to work.

For example:

```
POST http://admin:password@couchdb:5984/_restart
```

The return value (if the server has not already restarted) is a JSON status object indicating that the request has been received:

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

If the server has already restarted, the header may be returned, but no actual data is contained in the response.

9.7. GET /_stats

Method	GET /_stats
Request	None
Response	Server statistics
Admin Privileges Required	no
Return Codes	
200	Request completed successfully.

The _stats method returns a JSON object containting the statistics for the running server. The object is structured with top-level sections collating the statistics for a range of entries, with each individual statistic being easily identified, and the content of each statistic is self-describing. For example, the request time statistics, within the couchdb section are structured as follows:

```
{
   "couchdb" : {
...
      "request_time" : {
         "stddev" : "27.509",
         "min" : "0.333333333333333",
         "max" : "152",
         "current" : "400.976",
         "mean" : "10.837",
         "sum" : "400.976",
         "description" : "length of a request inside CouchDB without MochiWeb"
      },
...
    }
}
```

The fields provide the current, minimum and maximum, and a collection of statistical means and quantities. The quantity in each case is not defined, but the descriptions below provide

The statistics are divided into the following top-level sections:

- couchdb

  Describes statistics specific to the internals of CouchDB.

  Table 9.5. couchdb statistics

Statistic ID	Description	Unit
auth_cache_hits	Number of authentication cache hits	number
auth_cache_misses	Number of authentication cache misses	number

Statistic ID	Description	Unit
database_reads	Number of times a document was read from a database	number
database_writes	Number of times a database was changed	number
open_databases	Number of open databases	number
open_os_files	Number of file descriptors CouchDB has open	number
request_time	Length of a request inside CouchDB without MochiWeb	milliseconds

- httpd_request_methods

Table 9.6. httpd_request_methods statistics

Statistic ID	Description	Unit
COPY	Number of HTTP COPY requests	number
DELETE	Number of HTTP DELETE requests	number
GET	Number of HTTP GET requests	number
HEAD	Number of HTTP HEAD requests	number
POST	Number of HTTP POST requests	number
PUT	Number of HTTP PUT requests	number

- httpd_status_codes

Table 9.7. httpd_status_codes statistics

Statistic ID	Description	Unit
200	Number of HTTP 200 OK responses	number
201	Number of HTTP 201 Created responses	number
202	Number of HTTP 202 Accepted responses	number
301	Number of HTTP 301 Moved Permanently responses	number
304	Number of HTTP 304 Not Modified responses	number
400	Number of HTTP 400 Bad Request responses	number
401	Number of HTTP 401 Unauthorized responses	number
403	Number of HTTP 403 Forbidden responses	number
404	Number of HTTP 404 Not Found responses	number
405	Number of HTTP 405 Method Not Allowed responses	number

Statistic ID	Description	Unit
409	Number of HTTP 409 Conflict responses	number
412	Number of HTTP 412 Precondition Failed responses	number
500	Number of HTTP 500 Internal Server Error responses	number

- httpd

Table 9.8. httpd statistics

Statistic ID	Description	Unit
bulk_requests	Number of bulk requests	number
clients_requesting_changes	Number of clients for continuous _changes	number
requests	Number of HTTP requests	number
temporary_view_reads	Number of temporary view reads	number
view_reads	Number of view reads	number

You can also access individual statistics by quoting the statistics sections and statistic ID as part of the URL path. For example, to get the request_time statistics, you can use:

```
GET /_stats/couchdb/request_time
```

This returns an entire statistics object, as with the full request, but containining only the request individual statistic. Hence, the returned structure is as follows:

```
{
   "couchdb" : {
      "request_time" : {
         "stddev" : 7454.305,
         "min" : 1,
         "max" : 34185,
         "current" : 34697.803,
         "mean" : 1652.276,
         "sum" : 34697.803,
         "description" : "length of a request inside CouchDB without MochiWeb"
      }
   }
}
```

9.8. GET /_utils

Method	GET /_utils
Request	None
Response	Administration interface
Admin Privileges Required	no

Accesses the built-in Futon administration interface for CouchDB.

9.9. GET /_uuids

Method	GET /_uuids

Request	None	
Response	List of UUIDs	
Admin Privileges Required	no	
Query Arguments	**Argument**	`count`
	Description	Number of UUIDs to return
	Optional	yes
	Type	numeric
Return Codes		
200	Request completed successfully.	

Requests one or more Universally Unique Identifiers (UUIDs) from the CouchDB instance. The response is a JSON object providing a list of UUIDs. For example:

```
{
   "uuids" : [
      "7e4b5a14b22ec1cf8e58b9cdd0000da3"
   ]
}
```

You can use the `count` argument to specify the number of UUIDs to be returned. For example:

```
GET http://couchdb:5984/_uuids?count=5
```

Returns:

```
{
   "uuids" : [
      "c9df0cdf4442f993fc5570225b405a80",
      "c9df0cdf4442f993fc5570225b405bd2",
      "c9df0cdf4442f993fc5570225b405e42",
      "c9df0cdf4442f993fc5570225b4061a0",
      "c9df0cdf4442f993fc5570225b406a20"
   ]
}
```

The UUID type is determined by the UUID type setting in the CouchDB configuration. See Section 10.4, “`PUT /_config/section/key`”.

For example, changing the UUID type to `random`:

```
PUT http://couchdb:5984/_config/uuids/algorithm
Content-Type: application/json
Accept: */*

"random"
```

When obtaining a list of UUIDs:

```
{
   "uuids" : [
      "031aad7b469956cf2826fcb2a9260492",
      "6ec875e15e6b385120938df18ee8e496",
      "cff9e881516483911aa2f0e98949092d",
      "b89d37509d39dd712546f9510d4a9271",
      "2e0dbf7f6c4ad716f21938a016e4e59f"
   ]
}
```

9.10. `GET /favicon.ico`

Method	`GET /favicon.ico`

Request	None
Response	Binary content for the favicon.ico site icon
Admin Privileges Required	no
Return Codes	
200	Request completed successfully.
404	The requested content could not be found. The returned content will include further information, as a JSON object, if available.

Returns the site icon. The return `Content-type` header is `image/x-icon`, and the content stream is the image data.

Chapter 10. CouchDB API Server Configuration Methods

The CouchDB API Server Configuration Methods provide an interface to query and update the various configuration values within a running CouchDB instance.

A list of the available methods and URL paths are provided below:

Table 10.1. Configuration API Calls

Method	Path	Description
`GET`	`/_config`	Obtain a list of the entire server configuration
`GET`	`/_config/section`	Get all the configuration values for the specified section
`GET`	`/_config/section/key`	Get a specific section/configuration value
`PUT`	`/_config/section/key`	Set the specified configuration value
`DELETE`	`/_config/section/key`	Delete the current setting

10.1. `GET /_config`

Method	`GET /_config`
Request	None
Response	Returns a structure configuration name and value pairs, organized by section
Admin Privileges Required	no

Returns the entire CouchDB server configuration as a JSON structure. The structure is organized by different configuration sections, with individual values.

For example, to get the configuration for a server:

```
GET http://couchdb:5984/_config
Accept: application/json
```

The response is the JSON structure:

```
{
   "query_server_config" : {
      "reduce_limit" : "true"
   },
   "couchdb" : {
      "os_process_timeout" : "5000",
      "max_attachment_chunk_size" : "4294967296",
      "max_document_size" : "4294967296",
      "uri_file" : "/var/lib/couchdb/couch.uri",
      "max_dbs_open" : "100",
      "view_index_dir" : "/var/lib/couchdb",
      "util_driver_dir" : "/usr/lib64/couchdb/erlang/lib/couch-1.0.1/priv/lib",
      "database_dir" : "/var/lib/couchdb",
      "delayed_commits" : "true"
   },
   "attachments" : {
      "compressible_types" : "text/*, application/javascript, application/json,  application/xml",
      "compression_level" : "8"
   },
   "uuids" : {
      "algorithm" : "utc_random"
   },
   "daemons" : {
      "view_manager" : "{couch_view, start_link, []}",
      "auth_cache" : "{couch_auth_cache, start_link, []}",
      "uuids" : "{couch_uuids, start, []}",
      "stats_aggregator" : "{couch_stats_aggregator, start, []}",
      "query_servers" : "{couch_query_servers, start_link, []}",
      "httpd" : "{couch_httpd, start_link, []}",
      "stats_collector" : "{couch_stats_collector, start, []}",
      "db_update_notifier" : "{couch_db_update_notifier_sup, start_link, []}",
      "external_manager" : "{couch_external_manager, start_link, []}"
   },
   "stats" : {
      "samples" : "[0, 60, 300, 900]",
      "rate" : "1000"
   },
   "httpd" : {
      "vhost_global_handlers" : "_utils, _uuids, _session, _oauth, _users",
      "secure_rewrites" : "true",
      "authentication_handlers" : "{couch_httpd_oauth, oauth_authentication_handler},
                                   {couch_httpd_auth, cookie_authentication_handler},
                                   {couch_httpd_auth, default_authentication_handler}",
      "port" : "5984",
      "default_handler" : "{couch_httpd_db, handle_request}",
      "allow_jsonp" : "false",
      "bind_address" : "192.168.0.2",
      "max_connections" : "2048"
   },
   "query_servers" : {
      "javascript" : "/usr/bin/couchjs /usr/share/couchdb/server/main.js"
   },
   "couch_httpd_auth" : {
      "authentication_db" : "_users",
      "require_valid_user" : "false",
      "authentication_redirect" : "/_utils/session.html",
      "timeout" : "600",
      "auth_cache_size" : "50"
   },
   "httpd_db_handlers" : {
      "_design" : "{couch_httpd_db, handle_design_req}",
      "_compact" : "{couch_httpd_db, handle_compact_req}",
      "_view_cleanup" : "{couch_httpd_db, handle_view_cleanup_req}",
      "_temp_view" : "{couch_httpd_view, handle_temp_view_req}",
      "_changes" : "{couch_httpd_db, handle_changes_req}"
   },
   "replicator" : {
      "max_http_sessions" : "10",
      "max_http_pipeline_size" : "10"
   },
   "log" : {
      "include_sasl" : "true",
      "level" : "info",
      "file" : "/var/log/couchdb/couch.log"
   },
   "httpd_design_handlers" : {
      "_update" : "{couch_httpd_show, handle_doc_update_req}",
      "_show" : "{couch_httpd_show, handle_doc_show_req}",
      "_info" : "{couch_httpd_db,   handle_design_info_req}",
      "_list" : "{couch_httpd_show, handle_view_list_req}",
      "_view" : "{couch_httpd_view, handle_view_req}",
      "_rewrite" : "{couch_httpd_rewrite, handle_rewrite_req}"
   },
   "httpd_global_handlers" : {
      "_replicate" : "{couch_httpd_misc_handlers, handle_replicate_req}",
      "/" : "{couch_httpd_misc_handlers, handle_welcome_req, <<\"Welcome\">>}",
```

10.2. GET /_config/section

Method	GET /_config/section
Request	None
Response	All the configuration values within a specified section
Admin Privileges Required	no

Gets the configuration structure for a single section. For example, to retrieve the CouchDB configuration section values:

```
GET http://couchdb:5984/_config/couchdb
Accept: application/json
```

The returned JSON contains just the configuration values for this section:

```
{
   "os_process_timeout" : "5000",
   "max_attachment_chunk_size" : "4294967296",
   "max_document_size" : "4294967296",
   "uri_file" : "/var/lib/couchdb/couch.uri",
   "max_dbs_open" : "100",
   "view_index_dir" : "/var/lib/couchdb",
   "util_driver_dir" : "/usr/lib64/couchdb/erlang/lib/couch-1.0.1/priv/lib",
   "database_dir" : "/var/lib/couchdb",
   "delayed_commits" : "true"
}
```

10.3. GET /_config/section/key

Method	GET /_config/section/key
Request	None
Response	Value of the specified key/section
Admin Privileges Required	no

Gets a single configuration value from within a specific configuration section. For example, to obtain the current log level:

```
GET http://couchdb:5984/_config/log/level
Accept: application/json
```

Returns the string of the log level:

```
"info"
```

Note

The returned value will be the JSON of the value, which may be a string or numeric value, or an array or object. Some client environments may not parse simple strings or numeric values as valid JSON.

10.4. PUT /_config/section/key

Method	PUT /_config/section/key
Request	Value structure
Response	Previous value
Admin Privileges Required	no
Return Codes	

200	Configuration option updated successfully
500	Error setting configuration

Updates a configuration value. The new value should be supplied in the request body in the corresponding JSON format. For example, if you are setting a string value, you must supply a valid JSON string.

For example, to set the function used to generate UUIDs by the `GET /_uuids` API call to use the `utc_random` generator:

```
PUT http://couchdb:5984/_config/uuids/algorithm
Content-Type: application/json

"utc_random"
```

The return value will be empty, with the response code indicating the success or failure of the configuration setting.

10.5. `DELETE /_config/section/key`

Method	`DELETE /_config/section/key`	
Request	None	
Response	Previous value	
Admin Privileges Required	no	
Query Arguments	**Argument**	`rev`
	Description	Current revision of the document for validation
	Optional	yes
	Type	string
HTTP Headers	**Header**	`If-Match`
	Description	Current revision of the document for validation
	Optional	yes
Return Codes		
409	Supplied revision is incorrect or missing	

Deletes a configuration value. The returned JSON will be the value of the configuration parameter before it was deleted. For example, to delete the UUID parameter:

```
DELETE http://couchdb:5984/_config/uuids/algorithm
Content-Type: application/json
```

The returned value is the last configured UUID function:

```
"random"
```

Chapter 11. CouchDB API Server Authentication Methods

The CouchDB Authentication methods provide an interface for obtaining session and authorization data.

A list of the available methods and URL paths are provided below:

Table 11.1. Authentication API Calls

Method	Path	Description
GET	/_oauth/access_token	TBC
GET	/_oauth/authorize	TBC
POST	/_oauth/authorize	TBC
GET	/_oauth/request_token	TBC
GET	/_session	Returns cookie based login user information
POST	/_session	Do cookie based user login
DELETE	/_session	Logout cookie based user

Chapter 12. Configuring CouchDB

12.1. CouchDB Configuration Files

12.2. Configuration File Locations

CouchDB reads files from the following locations, in the following order.

1. `PREFIX/default.ini`
2. `PREFIX/default.d/*`
3. `PREFIX/local.ini`
4. `PREFIX/local.d/*`

Settings in successive documents override the settings in earlier entries. For example, setting the `bind_address` parameter in `local.ini` would override any setting in `default.ini`.

> **Warning**
>
> The `default.ini` file may be overwritten during an upgrade or re-installation, so localised changes should be made to the `local.ini` file or files within the `local.d` directory.

12.3. MochiWeb Server Options

Server options for the MochiWeb component of CouchDB can be added to the configuration files. 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}]
```

12.4. 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 2.2, “HTTP Proxying”. For further background on the OS Daemon service, see CouchDB Externals API

12.5. Update Notifications

12.6. 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.

12.7. `vhosts` definitions

Similar to the rewrites section of a `_design` document, the `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.

12.8. Configuring SSL Network Sockets

SSL 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"}
```

12.9. CouchDB Configuration Options

Table 12.1. Configuration Groups

Section	Description
`attachments`	Attachment options
`couchdb`	CouchDB specific options
`daemons`	Daemons and background processes
`httpd_db_handlers`	Database Operation handlers
`couch_httpd_auth`	HTTPD Authentication options
`httpd`	HTTPD Server options
`httpd_design_handlers`	Handlers for design document operations
`httpd_global_handlers`	Handlers for global operations
`log`	Logging options
`query_servers`	Query Server options
`query_server_config`	Query server options
`replicator`	Replicator Options
`ssl`	SSL (Secure Sockets Layer) Options
`stats`	Statistics options

Section	Description
uuids	UUID generation options

12.9.1. attachments Configuration Options

Table 12.2. Configuration Groups

Option	Description
compressible_types	compressible_types
compression_level	compression_level

12.9.2. couchdb Configuration Options

Table 12.3. Configuration Groups

Option	Description
database_dir	database_dir
delayed_commits	delayed_commits
max_attachment_chunk_size	max_attachment_chunk_size
max_dbs_open	max_dbs_open
max_document_size	max_document_size
os_process_timeout	os_process_timeout
uri_file	uri_file
util_driver_dir	util_driver_dir
view_index_dir	view_index_dir

12.9.3. daemons Configuration Options

Table 12.4. Configuration Groups

Option	Description
httpsd	Enabled HTTPS service
auth_cache	auth_cache
db_update_notifier	db_update_notifier
external_manager	external_manager
httpd	httpd
query_servers	query_servers
stats_aggregator	stats_aggregator
stats_collector	stats_collector
uuids	uuids
view_manager	view_manager

12.9.4. `httpd_db_handlers` Configuration Options

Table 12.5. Configuration Groups

Option	Description
`_changes`	_changes
`_compact`	_compact
`_design`	_design
`_temp_view`	_temp_view
`_view_cleanup`	_view_cleanup

12.9.5. `couch_httpd_auth` Configuration Options

Table 12.6. Configuration Groups

Option	Description
`auth_cache_size`	auth_cache_size
`authentication_db`	authentication_db
`authentication_redirect`	authentication_redirect
`require_valid_user`	require_valid_user
`timeout`	timeout

12.9.6. `httpd` Configuration Options

Table 12.7. Configuration Groups

Option	Description
`nodelay`	Enable TCP_NODELAY
`allow_jsonp`	allow_jsonp
`authentication_handlers`	authentication_handlers
`bind_address`	bind_address
`default_handler`	default_handler
`max_connections`	max_connections
`port`	port
`secure_rewrites`	secure_rewrites
`vhost_global_handlers`	vhost_global_handlers

12.9.7. `httpd_design_handlers` Configuration Options

Table 12.8. Configuration Groups

Option	Description
`_info`	_info
`_list`	_list
`_rewrite`	_rewrite
`_show`	_show
`_update`	_update
`_view`	_view

12.9.8. `httpd_global_handlers` Configuration Options

Table 12.9. Configuration Groups

Option	Description
`/`	/
`_active_tasks`	_active_tasks
`_all_dbs`	_all_dbs
`_config`	_config
`_log`	_log
`_oauth`	_oauth
`_replicate`	_replicate
`_restart`	_restart
`_session`	_session
`_stats`	_stats
`_utils`	_utils
`_uuids`	_uuids
`favicon.ico`	favicon.ico

12.9.9. `log` Configuration Options

Table 12.10. Configuration Groups

Option	Description
`file`	file
`include_sasl`	include_sasl
`level`	level

12.9.10. `query_servers` Configuration Options

Table 12.11. Configuration Groups

Option	Description
javascript	javascript

12.9.11. query_server_config Configuration Options

Table 12.12. Configuration Groups

Option	Description
reduce_limit	reduce_limit

12.9.12. replicator Configuration Options

Table 12.13. Configuration Groups

Option	Description
max_http_pipeline_size	max_http_pipeline_size
max_http_sessions	max_http_sessions

12.9.13. stats Configuration Options

Table 12.14. Configuration Groups

Option	Description
rate	rate
samples	samples

12.9.14. uuids Configuration Options

Table 12.15. Configuration Groups

Option	Description
algorithm	algorithm

Appendix A. JSON Structure Reference

The following appendix provides a quick reference to all the JSON structures that you can supply to CouchDB, or get in return to requests.

Table A.1. JSON Structures

Description
All Database Documents
Bulk Document Response
Bulk Documents
Changes information for a database
CouchDB Document
CouchDB Error Status
CouchDB database information object
Design Document
Design Document Information
Design Document spatial index Information
Document with Attachments
List of Active Tasks
Replication Settings
Replication Status
Returned CouchDB Document with Detailed Revision Info
Returned CouchDB Document with Revision Info
Returned Document with Attachments
Security Object

Table A.2. All Database Documents

Field	Description
`offset`	Offset where the document list started
`rows [array]`	Array of document object
`total_rows`	Number of documents in the database/view
`update_seq` (optional)	Current update sequence for the database

Table A.3. Bulk Document Response

Field	Description
`docs [array]`	Bulk Docs Returned Documents
`error`	Error type
`id`	Document ID
`reason`	Error string with extended reason

Table A.4. Bulk Documents

Field	Description

all_or_nothing (optional)	Sets the database commit mode to use all-or-nothing semantics
docs [array]	Bulk Documents Document
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)
_deleted (optional)	Whether the document should be deleted

Table A.5. Changes information for a database

Field	Description
last_seq	Last change sequence number
results [array]	Changes made to a database
changes [array]	List of changes, field-by-field, for this document
id	Document ID
seq	Update sequence number

Table A.6. CouchDB Document

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)

Table A.7. CouchDB Error Status

Field	Description
error	Error type
id	Document ID
reason	Error string with extended reason

Table A.8. CouchDB database information object

Field	Description
committed_update_seq	The number of committed update.
compact_running	Set to true if the database compaction routine is operating on this database.
db_name	The name of the database.
disk_format_version	The version of the physical format used for the data when it is stored on disk.
disk_size	Size in bytes of the data as stored on the disk. Views indexes are not included in the calculation.
doc_count	A count of the documents in the specified database.
doc_del_count	Number of deleted documents
instance_start_time	Timestamp of when the database was created, expressed in milliseconds since the epoch.
purge_seq	The number of purge operations on the database.
update_seq	The current number of updates to the database.

Table A.9. Design Document

Field	Description

_id	Design Document ID
_rev	Design Document Revision
views	View
viewname	View Definition
map	Map Function for View
reduce (optional)	Reduce Function for View

Table A.10. Design Document Information

Field	Description
name	Name/ID of Design Document
view_index	View Index
compact_running	Indicates whether a compaction routine is currently running on the view
disk_size	Size in bytes of the view as stored on disk
language	Language for the defined views
purge_seq	The purge sequence that has been processed
signature	MD5 signature of the views for the design document
update_seq	The update sequence of the corresponding database that has been indexed
updater_running	Indicates if the view is currently being updated
waiting_clients	Number of clients waiting on views from this design document
waiting_commit	Indicates if there are outstanding commits to the underlying database that need to processed

Table A.11. Design Document spatial index Information

Field	Description
name	Name/ID of Design Document
spatial_index	View Index
compact_running	Indicates whether a compaction routine is currently running on the view
disk_size	Size in bytes of the view as stored on disk
language	Language for the defined views
purge_seq	The purge sequence that has been processed
signature	MD5 signature of the views for the design document
update_seq	The update sequence of the corresponding database that has been indexed
updater_running	Indicates if the view is currently being updated
waiting_clients	Number of clients waiting on views from this design document
waiting_commit	Indicates if there are outstanding commits to the underlying database that need to processed

Table A.12. Document with Attachments

Field	Description
_id (optional)	Document ID
_rev (optional)	Revision ID (when updating an existing document)

`_attachments` (optional)	Document Attachment
`filename`	Attachment information
`content_type`	MIME Content type string
`data`	File attachment content, Base64 encoded

Table A.13. List of Active Tasks

Field	Description
`tasks [array]`	Active Task
`pid`	Process ID
`status`	Task status message
`task`	Task name
`type`	Operation Type

Table A.14. Replication Settings

Field	Description
`cancel` (optional)	Cancels the replication
`continuous` (optional)	Configure the replication to be continuous
`create_target` (optional)	Creates the target database
`doc_ids` (optional)	Array of document IDs to be synchronized
`proxy` (optional)	Address of a proxy server through which replication should occur
`source`	Source database name or URL
`target`	Target database name or URL

Table A.15. Replication Status

Field	Description
`history [array]`	Replication History
`doc_write_failures`	Number of document write failures
`docs_read`	Number of documents read
`docs_written`	Number of documents written to target
`end_last_seq`	Last sequence number in changes stream
`end_time`	Date/Time replication operation completed
`missing_checked`	Number of missing documents checked
`missing_found`	Number of missing documents found
`recorded_seq`	Last recorded sequence number
`session_id`	Session ID for this replication operation
`start_last_seq`	First sequence number in changes stream
`start_time`	Date/Time replication operation started
`ok`	Replication status
`session_id`	Unique session ID
`source_last_seq`	Last sequence number read from source database

Table A.16. Returned CouchDB Document with Detailed Revision Info

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_revs_info [array]`	CouchDB Document Extended Revision Info
`rev`	Full revision string
`status`	Status of the revision

Table A.17. Returned CouchDB Document with Revision Info

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_revisions`	CouchDB Document Revisions
`ids [array]`	Array of valid revision IDs, in reverse order (latest first)
`start`	Prefix number for the latest revision

Table A.18. Returned Document with Attachments

Field	Description
`_id` (optional)	Document ID
`_rev` (optional)	Revision ID (when updating an existing document)
`_attachments` (optional)	Document Attachment
`filename`	Attachment
`content_type`	MIME Content type string
`length`	Length (bytes) of the attachment data
`revpos`	Revision where this attachment exists
`stub`	Indicates whether the attachment is a stub

Table A.19. Security Object

Field	Description
`admins`	Roles/Users with admin privileges
`roles [array]`	List of roles with parent privilege
`users [array]`	List of users with parent privilege
`readers`	Roles/Users with reader privileges
`roles [array]`	List of roles with parent privilege
`users [array]`	List of users with parent privilege