CouchDB
The Definitive Guide
(../index.html)

SEARCH

This edition of the book is a work in progress.
Please create a pull request (https://github.com/oreilly/couchdb-guide/pulls) or report an issue (http://github.com/oreilly/couchdb-guide/issues) for any corrections or suggestions you may have.

Security

(#security)
We mentioned earlier that CouchDB is still in development and that features may have been added since the publication of this book. This is especially true for the security mechanisms in CouchDB. There is rudimentary support in the currently released versions (0.10.0), but as we're writing these lines, additions are being discussed.

In this chapter, we'll look at the basic security mechanisms in CouchDB: the *Admin Party*, *Basic Authentication*, *Cookie Authentication*, and *OAuth*.

The Admin Party

(#party)
When you start out fresh, CouchDB allows any request to be made by anyone. Create a database? No problem, here you go. Delete some documents? Same deal. CouchDB calls this the *Admin Party*. Everybody has privileges to do anything. Neat.

While it is incredibly easy to get started with CouchDB that way, it should be obvious that putting a default installation into the wild is adventurous. Any rogue client could come along and delete a database.

A note of relief: by default, CouchDB will listen only on your loopback network interface (`127.0.0.1` or `localhost`) and thus only you will be able to make requests to CouchDB, nobody else. But when you start to open up your CouchDB to the public (that is, by telling it to bind to your machine's public IP address), you will

Home (../index.html)
Draft edition (index.html)
Previous Page (cookbook.html)
Next Page (performance.html)

Home

- Security (#security)
  - The Admin Party (#party)
    - Creating New Admin Users (#users)
    - Hashing Passwords (#hashing)
  - Basic Authentication (#authentication)
    - Update Validations Again (#validation)
  - Cookie Authentication (#cookies)
  - Network Server Security (#network)

want to think about restricting access so that the next bad guy doesn't ruin your admin party.

In our previous discussions, we dropped some keywords about how things without the admin party work. First, there's *admin* itself, which implies some sort of super user. Then there are *privileges*. Let's explore these terms a little more.

CouchDB has the idea of an *admin user* (e.g. an administrator, a super user, or root) that is allowed to do anything to a CouchDB installation. By default, everybody is an admin. If you don't like that, you can create specific admin users with a username and password as their credentials.

CouchDB also defines a set of requests that only admin users are allowed to do. If you have defined one or more specific admin users, CouchDB will ask for identification for certain requests:

- Creating a database (`PUT /database`)
- Deleting a database (`DELETE /database`)
- Creating a design document (`PUT /database/_design/app`)
- Updating a design document (`PUT /database/_design/app?rev=1-4E2`)
- Deleting a design document (`DELETE /database/_design/app?rev=1-6A7`)
- Triggering compaction (`POST /_compact`)
- Reading the task status list (`GET /_active_tasks`)
- Restarting the server (`POST /_restart`)
- Reading the active configuration (`GET /_config`)
- Updating the active configuration (`PUT /_config`)

Creating New Admin Users

(#users)

Let's do another walk through the API using `curl` to see how CouchDB behaves when you add admin users.

```
> HOST="http://127.0.0.1:5984"
> curl -X PUT $HOST/database
{"ok":true}
```

When starting out fresh, we can add a database. Nothing unexpected. Now let's create an admin user. We'll call her `anna`, and her password is `secret`. Note the double quotes in the following code; they are needed to denote a string value for the configuration API (as we learned earlier):

```
curl -X PUT $HOST/_config/admins/anna -d '"secret"'
""
```

As per the `_config` API's behavior, we're getting the previous value for the config item we just wrote. Since our admin user didn't exist, we get an empty string.

When we now sneak over to the CouchDB log file, we find these two entries:

```
[debug] [<0.43.0>] saving to file
'/Users/jan/Work/couchdb-
git/etc/couchdb/local_dev.ini', Config:
'{{"admins","anna"},"secret"}'

[debug] [<0.43.0>] saving to file
'/Users/jan/Work/couchdb-
git/etc/couchdb/local_dev.ini',
Config:'{{"admins","anna"}, "-hashed-
6a1cc3760b4d09c150d44edf302ff40606221526,a69a9e4f0047b
e899ebfe09a40b2f52c"}'
```

The first is our initial request. You see that our admin user gets written to the CouchDB configuration files. We set our CouchDB log level to `debug` to see exactly what is going on. We first see the request coming in with a plain-text password and then again with a hashed password.

Hashing Passwords

(#hashing)

Seeing the plain-text password is scary, isn't it? No worries; in normal operation when the log level is not set to `debug`, the plain-text password doesn't show up anywhere. It gets hashed right away. The hash is that big, ugly, long string that starts out with `-hashed-`. How does that work?

1. Creates a new 128-bit UUID. This is our *salt*.
2. Creates a sha1 hash of the concatenation of the bytes of the plain-text password and the salt (`sha1(password + salt)`).
3. Prefixes the result with `-hashed-` and appends `,salt`.

To compare a plain-text password during authentication with the stored hash, the same procedure is run and the resulting hash is compared to the stored hash. The probability of two identical hashes for different passwords is too insignificant to mention (c.f.

Bruce Schneier). Should the stored hash fall into the hands of an attacker, it is, by current standards, way too inconvenient (i.e., it'd take a lot of money and time) to find the plain-text password from the hash.

But what's with the `-hashed-` prefix? Well, remember how the configuration API works? When CouchDB starts up, it reads a set of `.ini` files with config settings. It loads these settings into an internal data store (not a database). The config API lets you read the current configuration as well as change it and create new entries. CouchDB is writing any changes back to the `.ini` files.

The `.ini` files can also be edited by hand when CouchDB is not running. Instead of creating the admin user as we showed previously, you could have stopped CouchDB, opened your `local.ini`, added `anna = secret` to the `[admins]` section, and restarted CouchDB. Upon reading the new line from `local.ini`, CouchDB would run the hashing algorithm and write back the hash to `local.ini`, replacing the plain-text password. To make sure CouchDB only hashes plain-text passwords and not an existing hash a second time, it prefixes the hash with `-hashed-`, to distinguish between plain-text passwords and hashed passwords. This means your plain-text password can't start with the characters `-hashed-`, but that's pretty unlikely to begin with.

Basic Authentication

(#authentication)

Now that we have defined an admin, CouchDB will not allow us to create new databases unless we give the correct admin user credentials. Let's verify:

```
> curl -X PUT $HOST/somedatabase
{"error":"unauthorized","reason":"You are not a server
admin."}
```

That looks about right. Now we try again with the correct credentials:

```
> HOST="http://anna:secret@127.0.0.1:5984"
> curl -X PUT $HOST/somedatabase
{"ok":true}
```

If you have ever accessed a website or FTP server that was password-protected, the `username:password@` URL variant should look familiar.

If you are security conscious, the missing `s` in `http://` will make you nervous. We're sending our password to CouchDB in plain text. This is a bad thing, right? Yes, but consider our scenario: CouchDB listens on `127.0.0.1` on a development box that we're the sole user of. Who could possibly sniff our password?

If you are in a production environment, however, you need to reconsider. Will your CouchDB instance communicate over a public network? Even a LAN shared with other colocation customers is public. There are multiple ways to secure communication between you or your application and CouchDB that exceed the scope of this book. ~~We suggest you read up on VPNs and setting up CouchDB behind an HTTP proxy (like Apache httpd's mod_proxy, nginx, or varnish) that will handle SSL for you. CouchDB does not support exposing its API via SSL at the moment. It can, however, replicate with other CouchDB instances that are behind an SSL proxy~~ CouchDB as of version 1.1.0 comes with **SSL built in (http://wiki.apache.org/couchdb/How_to_enable_SSL)**.

Update Validations Again

(#validation)

Do you remember **Chapter 7, Validation Functions (validation.html)**? We had an update validation function that allowed us to verify that the claimed author of a document matched the authenticated username.

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

What is this `userCtx` exactly? It is an object filled with information about the current request's authentication data. Let's have a look at what's in there. We'll show you a simple trick how to introspect what's going on in all the JavaScript you are writing.

```
> curl -X PUT $HOST/somedatabase/_design/log -d
'{"validate_doc_update":"function(newDoc, oldDoc,
userCtx) { log(userCtx); }"}'
{"ok":true,"id":"_design/log","rev":"1-
498bd568e17e93d247ca48439a368718"}
```

Let's show the `validate_doc_update` function:

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

This gets called for every future document update and does nothing but print a log entry into CouchDB's log file. If we now create a new document:

```
> curl -X POST $HOST/somedatabase/ -d '{"a":1}'
{"ok":true,"id":"36174efe5d455bd45fa1d51efbcff986","re
v":"1-23202479633c2b380f79507a776743d5"} -H "Content-
Type: application/json"
```

we should see this in our `couch.log` file:

```
[info] [<0.9973.0>] OS Process :: {"db":
"somedatabase","name": "anna","roles":["_admin"]}
```

Let's format this again:

```
{
  "db": "somedatabase",
  "name": "anna",
  "roles": ["_admin"]
}
```

We see the current database, the name of the authenticated user, and an array of `roles`, with one role `"_admin"`. We can conclude that admin users in CouchDB are really just *regular users* with the *admin role* attached to them.

By separating users and roles from each other, the authentication system allows for flexible extension. For now, we'll just look at admin users.

Cookie Authentication

(#cookies)

Basic authentication that uses plain-text passwords is nice and convenient, but not very secure if no extra measures are taken. It is also a very poor user experience. If you use basic authentication to identify admins, your application's users need to deal with an ugly, unstylable browser modal dialog that says *non-professional at work* more than anything else.

To remedy some of these concerns, CouchDB supports *cookie authentication*. With cookie authentication your application doesn't have to include the ugly login dialog that the users' browsers come with. You can use a regular HTML form to submit logins to CouchDB. Upon receipt, CouchDB will generate a one-time token that the client can use in its next request to CouchDB. When CouchDB sees the token in a subsequent request, it will authenticate the user based on the token without the need to see the password again. By default, a token is valid for 10 minutes.

To obtain the first token and thus authenticate a user for the first time, the username and password must be sent to the `_session` API. The API is smart enough to decode HTML form submissions, so you don't have to resort to any smarts in your application.

If you are not using HTML forms to log in, you need to send an HTTP request that looks as if an HTML form generated it. Luckily, this is super simple:

```
> HOST="http://127.0.0.1:5984"
> curl -vX POST $HOST/_session -H 'Content-Type:
application/x-www-form-urlencoded' -d
'name=anna&password=secret'
```

CouchDB replies, and we'll give you some more detail:

```
< HTTP/1.1 200 OK
< Set-Cookie: AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-
53hw1sJepVzcVxnriEw;
< Version=1; Path=/; HttpOnly
> ...
<
{"ok":true}
```

A `200` response code tells us all is well, a `Set-Cookie` header includes the token we can use for the next request, and the standard JSON response tells us again that the request was successful.

Now we can use this token to make another request as the same user without sending the username and password again:

```
> curl -vX PUT $HOST/mydatabase --cookie
AuthSession=YW5uYTo0QUIzOTdFQjrC4ipN-D-
53hw1sJepVzcVxnriEw -H "X-CouchDB-WWW-Authenticate:
Cookie" -H "Content-Type: application/x-www-form-
urlencoded"
{"ok":true}
```

You can keep using this token for 10 minutes by default. After 10

minutes you need to authenticate your user again. The token lifetime can be configured with the `timeout` (in seconds) setting in the `couch_httpd_auth` configuration section.

Please note that for cookie authentication to work, you need to enable the `cookie_authentication_handler` in your `local.ini`:

```
[httpd]
authentication_handlers = {couch_httpd_auth,
cookie_authentication_handler},
{couch_httpd_oauth,
oauth_authentication_handler},
{couch_httpd_auth,
default_authentication_handler}
```

In addition, you need to define a *server secret*:

```
[couch_httpd_auth]
secret = yours3cr37pr4s3
```

Network Server Security

(#network)

CouchDB is a networked server, and there are best practices for securing these that are beyond the scope of this book. **Appendix D, Installing from Source (source.html)** includes some of those best practices. Make sure to understand the implications.