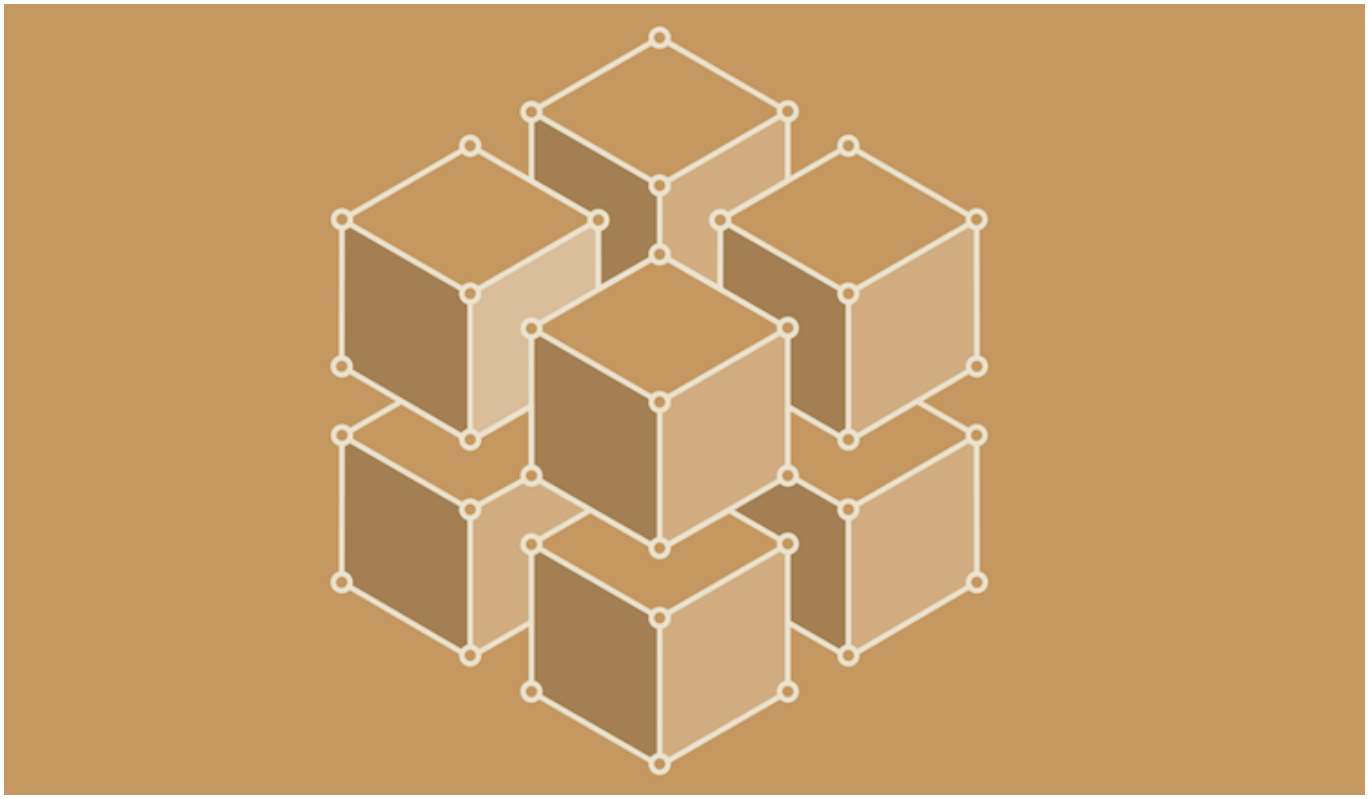
Node.js databases: Using CouchDB

YLD Follow
Nov 23, 2016 · 42 min read



Much like the previous two databases we presented here, CouchDB is an open-source key-value store. But it's also a bit more than that. Each record is not an opaque string: it's a JSON document that the engine understands.

By default, CouchDB does not impose any specific schema to the documents it stores. Instead, it allows any data that JSON allows — as long as we have an object as the root. Because of that, any two documents in the same database can hold completely different documents, and it's up to the application to make sense of them.

A schema-less document store like CouchDB is then optimised for flexibility and ease of use: there's no need to know the document schema upfront or to run expensive data migrations when you need to add another field.

Starting off

If you don't have CouchDB already installed, you can head to the official website (http://couchdb.apache.org/) to download and install it.

Once you have your CouchDB server started, you can begin interacting with it. CouchDB contains an HTTP server that you can use to make operations and issue queries on. You can use any command-line HTTP client like *curl* to interact with it:

> `curl` comes bundled with most operating system distributions, and is compatible with Windows. If you need to install it try using your favourite package manager, or head out to the official curl downloads page.

First, let's create a database we can play with:

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

Here we're sending an HTTP request to our local CouchDB HTTP server, which is listening to the default port 5984. We're specifying the HTTP method as `PUT`, which, in this case, instructs CouchDB to create the database we are specifying in the path portion of the URL: a database called `test`.

Each CouchDB server has one or more databases, and each database can hold any number of documents. Each document has a unique identifier. Let's then try to create one document inside this new `test` database:

```
$ curl -X POST http://127.0.0.1:5984/test -d '{"some": "data"}' -H
'Content-Type: application/json'
```

Here we're performing an HTTP request that specifies `POST` as the method. The URL is our `test` database URL, and we're specifying the request body payload to be this JSON-encoded object. We also have to add a request header which specifies that the content type is JSON.

On hitting the return key you shoud see a reply similar to the following:

```
{"ok":true,"id":"58767f1d0a41baca470d2af44f000bf2","rev":"1-
56b8a3a98ed03fbb3a804751a38611b2"}
```

This indicates that CouchDB has accepted our request, and that the new document was created and given the identifier contained in the response `id` property. The response also contains a `rev` property, which indicates the current document revision ID. We will later see what these revision identifiers are needed for.

Now we can try to use the ID returned to you to retrieve this document:

```
$ curl http://127.0.0.1:5984/test/ID
```

In your case, you will have to replace `ID` with the document ID returned to you when you first created it. In our case:

```
$ curl http://127.0.0.1:5984/test/58767f1d0a41baca470d2af44f000bf2
```

, which returns the document:

```
{"_id":"58767f1d0a41baca470d2af44f000bf2","_rev":"1-
56b8a3a98ed03fbb3a804751a38611b2","some":"data"}
```

You can now see that the simple document you inserted contains a few more attributes in it: `_id` and the `_rev` . CouchDB documents are augmented to contain the document

metadata: the unique document identifier and the revision identifier.

> In CouchDB, attributes prefixed with the underscore character _ are reserved for internal use.

Let's now try to get a document that doesn't exist in our database, this time inspecting the full HTTP response:

```
$ curl -i http://127.0.0.1:5984/test/does-not-exist
HTTP/1.1 404 Object Not Found
Server: CouchDB/1.6.1 (Erlang OTP/17)
Date: Wed, 21 Jan 2015 10:16:07 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 41
Cache-Control: must-revalidate

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

Here you can see that CouchDB replied with a status code 404, indicating that the requested document did not exist.

Let's now do another experiment: let's try to update the existing document from the command line:

```
$ curl -X PUT
http://127.0.0.1:5984/test/58767f1d0a41baca470d2af44f000bf2 -d
'{"some": "other", "attribute": true}' -H "Content-Type:
application/json" -i
```

If you replace the ID part of the URL with the ID of your document and hit the return key, you should see the following output:

```
HTTP/1.1 409 Conflict
Server: CouchDB/1.6.1 (Erlang OTP/17)
Date: Wed, 21 Jan 2015 10:19:06 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 58
Cache-Control: must-revalidate
```

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

Oops — CouchDB isn't letting us update our document. What's up there? This happened because of the way that CouchDB handles concurrency: to update a document you must specify the previous revision identifier you know. If CouchDB detects that the revision identifier you specify in the update request does not match the stored revision identifier for that document, it will indicate a conflict by replying with a 409 code. When two clients hold the same revision of the same document and do an update with the same revision ID, one of them will succeed — advancing the revision ID — and the other one will fail. By implementing conflict detection like this, it's up to the clients to handle conflicts.

> When it happens, a client can either give up or retry by querying the latest revision, perhaps merging the documents and then writing again, repeating this until successful.

Let's then specify the revision ID in our update command:

```
$ curl -X PUT
http://127.0.0.1:5984/test/58767f1d0a41baca470d2af44f000bf2 -d
'{"some": "other", "attribute": true, "_rev": "1-
56b8a3a98ed03fbb3a804751a38611b2"}' -H "Content-Type:
application/json" -i
```

If you type this last command, but first replace the ID in the URL and the revision identifier in the request data, you should get a reply indicating that the update was successful. You also get the identifier for the new revision of this document:

```
HTTP/1.1 201 Created
Server: CouchDB/1.6.1 (Erlang OTP/17)
Location: http://127.0.0.1:5984/test/58767f1d0a41baca470d2af44f000bf2
ETag: "2-221c0d018a44424525493a1c1ff34828"
Date: Wed, 21 Jan 2015 10:29:57 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{"ok":true,"id":"58767f1d0a41baca470d2af44f000bf2","rev":"2-
221c0d018a44424525493a1c1ff34828"}
```

Ladies and Gentlemen, start your Nodes

Now let's see how you can interact with a CouchDB server from a Node process. CouchDB speaks HTTP, so it would be enough to use the Node HTTP client or even the `request` NPM package. Instead, we're going to use this small wrapper around `request` that gives some nice convenient functions called `nano`. Let's install it:

```
$ npm install nano --save
```

> For the previous command to work, you should have a basic `package.json` file sitting in a new directory you can create for running the examples in this chapter.

Let's now create a basic module that exports a given server reference:

couchdb.js:

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

module.exports = nano(process.env.COUCHDB_URL ||
'http://127.0.0.1:5984');
```

As you can see, this module only requires the `nano` package and uses it to construct a database wrapper that points to the CouchDB server specified by the URL contained in the environment variable named `COUCHDB_URL`.

> If that environment variable isn't present, our `couchdb` module defaults to pointing to a local CouchDB installation, which can be useful during development time.
>
> Here we're assuming that you didn't specify any admin user with a password for your CouchDB server — your CouchDB server is still in "Admin Party" mode. If you specified a username and password, you can place the username and password in the URL in the form `http://username:password@127.0.0.1:5984`.

Let's now try to create a database on our server:

create_db.js:

```
var couch = require('./couchdb');

couch.db.create('test2', function(err) {
  if (err) {
    console.error(err);
  }
});
```

Let's try to run this:

```
$ node create_db
database test2 created successfully
```

You should see by the output that the database was successfully created. Now let's try to run this again:

```
$ node create_db
[Error: The database could not be created, the file already exists.]
 name: 'Error',
 error: 'file_exists',
 reason: 'The database could not be created, the file already
exists.',
 scope: 'couch',
 statusCode: 412,
 request:
  { method: 'PUT',
    headers:
     { 'content-type': 'application/json',
       accept: 'application/json' },
    uri: 'http://127.0.0.1:5984/test2' },
 headers:
  { date: 'Wed, 21 Jan 2015 11:29:09 GMT',
    'content-type': 'application/json',
    'cache-control': 'must-revalidate',
    statusCode: 412,
    uri: 'http://127.0.0.1:5984/test2' },
 errid: 'non_200',
 description: 'couch returned 412' }
```

You will now see that CouchDB returned an error because the `test2` database already existed. But it happens that we just want to make sure that the database exists, so we don't really care if this type of error happens. Let's then choose to ignore it:

create_db.js:

```
var couch = require('./couchdb');

couch.db.create('test2', function(err) {
  if (err && err.statusCode != 412) {
    console.error(err);
  }
  else {
    console.log('database test2 exists');
  }
});
```

Generally, when your Node process starts up, you want to make sure that all the necessary databases are up and running. Let's create a module to handle this initialisation step. But first you will need to install an NPM module we'll be using for helping us with the asynchronous flow control:

```
$ npm install async --save
```

Now, onto the module:

init_couch.js:

```
var async = require('async');
var couch = require('./couchdb');

var databases = ['users', 'messages'];

module.exports = initCouch;

function initCouch(cb) {
  createDatabases(cb);
```

```
}

function createDatabases(cb) {
  async.each(databases, createDatabase, cb);
}

function createDatabase(db, cb) {
  couch.db.create(db, function(err) {
    if (err && err.statusCode == 412) {
      err = null;
    }
    cb(err);
  });
}
```

This module exports this one function that only takes a callback function for when the initialisation is complete (or an unrecoverable error happens). This function then starts the database creation by calling the `createDatabases` function. This function uses `async` to create each database defined in the `databases` configuration array. Each database gets created by calling the `createDatabase`, which in turn uses `nano` to create the database, ignoring any error that occurs if the database already exists.

> If you're unsure about how the asynchronous control flow works, there is another book in this series named "Flow Control Patterns" that addresses this subject.

You can now use this module to initialise the state of you CouchDB server when the app is initialising:

app.js:

```
var initCouch = require('./init_couch');

initCouch(function(err) {
  if (err) {
    throw err
  }
  else {
    console.log('couchdb initialized');
  }
});
```

> Sometimes applications have separate scripts to initialise the database, but I find it much more convenient to have it transparently done at app start-up time, since there is no penalty involved, and it gracefully handles concurrent processes trying to initialise CouchDB at the same time.

Let's then run the `app` set-up:

```
$ node app
couchdb initialized
```

Now that the `users` and `messages` databases are created in our CouchDB server, we can start putting documents there.

The directory structure

As you may already have guessed, our application is going to handle users and messages between them. Instead of throwing the modules that handle these into the root directory, we're going to create a specific directory named `db`.

> Other common names for a directory holding data-access objects would be `models` or even `data`.
>
> When creating a real application, consider using a specific separate module to wrap database access instead of just one directory. This enables you to a) properly version this part of the system; b) have specific automated tests for this module; and c) increase the separation of concerns.

Creating documents with a specific ID

When creating a document, CouchDB can manufacture a unique document ID for you if you don't specify one. But it may happen that occasionally you want to force the identifier: like, for instance, when you want to reference a user document by the user ID or email. This has the automatic advantages of a) making it easy to fetch a given record, and b) avoiding duplicate entries.

Here is the minimum function for creating a user record:

db/users.js:

```
var users = require('../couchdb').use('users');

exports.create = function create(user, cb) {
  users.insert(user, user.email, cb);
};
```

> For each document type we're trying to mostly follow a REST-like convention for verbs. I usually try to stick with the verbs `create`, `get`, `list`, `destroy`, with some exceptions. One example of an exception is the getters or finders like `messages.getForUser`. Experts in REST may disagree with me...

This module starts out by getting a reference to the CouchDB users database in our CouchDB server.

> Unlike the two previous databases we addressed, this `users` object does not hold an actual database connection. Instead, it points to the base URL of that database, which in our case is `http://127.0.0.1:5984/users`.

Then it exports a `create` function. This function receives a user record as the first argument and inserts a document into the CouchDB `users` database. It specifies the ID as being the user email.

Let's use this module to create one user document:

user*insert*test.js:

```
var users = require('./db/users');

var user = {
  email: 'johndoe@example.com',
  name: 'John Doe',
  address: '1 Sesame Street'
};

users.create(user, function(err) {
  if (err) {
```

```
    throw err;
  }
  else {
    console.log('user inserted');
  }
});
```

If you try to run this, you should see a success message:

```
$ node user_insert_test.js
user inserted
```

When you try to run this for the second time, you should see the following conflict error, caused by a record with the same ID already existing:

```
$ node user_insert_test.js

Error: Document update conflict.
   ...
```

Forcing a schema

The current implementation of the user creation is too simple. It lacks at least two things: schema validation and error unification.

Currently, the database user creation API doesn't verify that the user-object argument is formatted as expected; it doesn't even validate that the user is an object. What we would want is to validate that the user document conforms to an expected schema, and not even try to create that user in the database if that schema is not respected.

To represent and validate schemas we're going to use an NPM module called `joi`. Let's then install it:

```
$ npm install joi --save
```

First, let's create a `schemas` directory where we will keep all the schemas our application will use:

```
$ mkdir schemas
```

Inside it, let's then create our user document schema:

schemas/user.js:

```
var Joi = require('joi');

module.exports = Joi.object().keys({
  email: Joi.string().email().required(),
  username: Joi.string().alphanum().min(3).max(30).required(),
  password: Joi.string().regex(/[a-zA-Z0-9]{3,30}/),
  access_token: [Joi.string(), Joi.number()],
  birthyear: Joi.number().integer().min(1900).max((new
Date()).getFullYear()),
});
```

Here we're using the Joi API to define a schema in an easy-to-read manner: a user is an object that contains the following keys:

- an email, which must be a valid email address and is required to exist;
- a username, which is a required alphanumerical string, containing at least three characters and a maximum of 30;
- a password, which must respect a certain regular expression;
- an access token, which is an optional string or number; and
- a birthyear, which is an integer between 1900 and the current year.

> This just serves as an example; Joi has many other types and options, described in the package instructions (https://github.com/hapijs/joi#readme).

Now we need a way to verify whether a certain object respects this schema or not:

schemas/index.js:

```
var schemaNames = ['user'];

var schemas = {};

schemaNames.forEach(function(schemaName) {
  schemas[schemaName] = require('./' + schemaName);
});

exports.validate = validate;

function validate(doc, schema, cb) {
  if (typeof schema == 'string') {
    schema = schemas[schema];
  }

  if (! schema) {
    cb(new Error('Unknown schema'));
  }
  else {
    Joi.validate(doc, schema, cb);
  }
};

exports.validating = function validating(schemaName, fn) {
  var schema = schemas[schemaName];
  if (! schema) {
    throw new Error('Unknown schema: ' + schemaName);
  }

  return function(doc, cb) {
    validate(doc, schema, function(err, doc) {
      if (err) {
        cb(err);
      }
      else {
        fn.call(null, doc, cb);
      }
    });
  };

};
```

This module collects the schema names in a `schemaNames` variable. (Now it just contains the `user` document schema, but in the future it may contain more.) It uses these names to load the schema modules from the current directory. This module then exports a

`validating` function, which accepts a schema name and a continuation function and returns a function. This function will check validation of the given document, and call the continuation function if it is valid. If the given document does not respect the schema, instead of calling the continuation function it will directly call the callback with the validation error.

This lets us easily plug the validation into the user creation API like this:

db/users.js:

```
var schemas = require('../schemas');

var users = require('../couchdb').use('users');

/// Create user

exports.create = schemas.validating('user', createUser);

function createUser(user, cb) {
  users.insert(user, user.email, cb);
}
```

Now, when our `createUser` function gets called, we are already assured that the given user object is valid, and that we can proceed to insert it into the database.

> If you require a directory path, and that directory contains an `index.js` file, that file gets loaded and evaluated as the value of that directory. The call `require('../schemas')` loads the module in `../schemas/index.js`.

Unifying errors

When an error happens at the validation layer, Joi calls our callback function with an error object that contains a descriptive message. If, on the contrary, the user object is a valid one, we proceed to try inserting it on CouchDB by handing it off to `nano`. If an error happens here, nano calls back with that error. This time the error can be an error not related to CouchDB (like when the CouchDB server is unreachable or times out) or related to CouchDB (like when there is already a user with that particular email address). How does a client handle these errors?

Imagine that we're building an HTTP API server. What HTTP status codes should we use for any of these errors? When a validation occurs, we should probably reply with a 400 (Bad Request) status code. When we try to create a user with an email that already exists, CouchDB replies with a 409 status code, which is the same code we should reply to the client, indicating a conflict. When we're having problems connecting or getting a response fromthe CouchDB server, we should return an internal error on the 5xx range, a 502 (Bad Gateway), a 504 (Gateway Timeout), or simply an opaque 500 (Internal Server Error).

In any case, we should make this easy on the API HTTP server implementation, and always return a unified error type which we can easily propagate to the client.

I usually resort to using `boom`, an NPM package that provides HTTP-friendly error codes.

> Why translate all errors to HTTP status codes? Because HTTP status codes are the closest thing we've got to a universal agreement over error codes; and you are probably going to serve your application over an HTTP API anyway.

Let's then install `boom`:

```
$ npm install boom --save
```

Next, we need to convert validation errors into a proper Boom error. Let's change our `schemas.validating` function to do just that:

schemas/index.js:

```
var Joi = require('joi');
var Boom = require('boom');

var schemaNames = ['user'];

var schemas = {};

schemaNames.forEach(function(schemaName) {
  schemas[schemaName] = require('./' + schemaName);
});
```

```
exports.validate = validate;

function validate(doc, schema, cb) {
  if (typeof schema == 'string') {
    schema = schemas[schema];
  }

  if (! schema) {
    cb(new Error('Unknown schema'));
  }
  else {
    Joi.validate(doc, schema, function(err, value) {
      if (err) {
        Boom.wrap(err, 400);
        cb(err);
      }
      else {
        cb(null, doc);
      }
    });
  }
};

exports.validating = function validating(schemaName, fn) {
  var schema = schemas[schemaName];
  if (! schema) {
    throw new Error('Unknown schema: ' + schemaName);
  }

  return function(doc, cb) {
    validate(doc, schema, function(err, doc) {
      if (err) {
        cb(err);
      }
      else {
        fn.call(null, doc, cb);
      }
    });
  };
};
```

In the case where we catch a validation error after invoking Joi, we wrap the error using `Boom.wrap`, turning it into a proper Boom error.

> Wrapping errors is generally better than replacing them: this way we don't lose context information that may be helpful for debugging a server or client problem.

Next, we can wrap the calls to CouchDB, turning any nano/CouchDB errors into Boom errors. We're going to create an `errors` module to do just that:

errors.js:

```
var Boom = require('boom');

exports.wrapNano = function wrapNanoError(cb) {
  return function(err) {
    if (err) {
      Boom.wrap(err, err.statusCode || 500);
    }
    cb.apply(null, arguments);
  };
}
```

Here we're exporting a `wrapNano` function that wraps the callback for a call to Nano, always calling back with a Boom error. Nano errors usually have a `statusCode` attribute (if they failed at the CouchDB server). We try to propagate that code. If we don't have an error code, we fall back into using a generic `500` error status code. After certifying that we have a Boom error or none at all, we delegate all arguments into the given callback.

Now we just need to use this new function to wrap every nano call:

db/users.js:

```
var schemas = require('../schemas');
var errors = require('../errors');

var users = require('../couchdb').use('users');

/// Create user

exports.create = schemas.validating('user', createUser);

function createUser(user, cb) {
  users.insert(user, user.email, errors.wrapNano(cb));
}
```

How to consume Boom errors

Now that we guarantee that all errors given by `users.create` are Boom errors, an HTTP JSON API just needs to propagate the status codes to the clients. If the HTTP JSON API server is implemented using Hapi.js, we don't need to do anything: Hapi already accepts Boom errors and will construct a proper reply to the client. If, for instance, you're using Express, you can create a simple error-handling middleware to respond to the client:

expressboom.js:

```
module.exports = function (err, req, res, next) {
  res.set(err.output.headers);
  res.status(err.output.statusCode);
  res.json(err.output.payload);
};
```

Here we're using the `output` property (present on all Boom errors) to propagate the headers, status code and error object into the client response. This error-handling middleware can then be included in an Express app to help the API users to hopefully get meaningful status codes when an error occurs.

Updating specific fields while handling conflicts

When we need to update some fields on a given document (like when the user updates their profile data), we need to send it to CouchDB. Unlike some databases, CouchDB has an opinion about concurrency: if two updates to the same document occur in concurrency, only one of them will win. To implement this, all CouchDB document updates must contain a revision ID. CouchDB will only accept to commit changes to a given document if the given revision ID matches the latest revision ID stored for that document.

Revision IDs are metadata contained inside a document. Let's see what they look like:

```
$ curl http://127.0.0.1:5984/users/whaa@example.com
{
  "_id":"whaa@example.com",
  "_rev":"1-25ee577ef2de8819d642687c38d6b777",
  "username":"johndoe",
```

```
  "email":"whaa@example.com"
}
```

Here you can spot the revision inside an attribute named `_rev`. To update a given document you have to pass in the whole document to CouchDB, which must include the revision ID.

As we already surfaced, this leaves us two basic choices of how to implement conflict-handling: we either delegate to the client (which is what CouchDB does) or we try to handle it on the application.

Delegate conflicts entirely to the client.

When delegating conflicts to the client, the easiest way to implement this is to force the client to give us the entire document (including the revision ID). In this case, updating the user record would look something like this:

end of db/users.js:

```
// ...
/// Update user

exports.update = schemas.validating('user', updateUser);

function updateUser(user, cb) {
  users.insert(user, errors.wrapNano(cb));
}
```

To allow a user object to have a `_rev` and `_id` attribute, we must first allow it on the schema:

schemas/user.js:

```
var Joi = require('joi');

module.exports = Joi.object().keys({
  _rev: Joi.string(),
  _id: Joi.string(),
```

```
  username: Joi.string().alphanum().min(3).max(30).required(),
  password: Joi.string().regex(/[a-zA-Z0-9]{3,30}/),
  access_token: [Joi.string(), Joi.number()],
  birthyear: Joi.number().integer().min(1900).max((new
Date()).getFullYear()),
  email: Joi.string().email()
});
```

We can now create a small script to try to update a specific user document:

update*user*test.js:

```
var users = require('./db/users');

var user = {
  _id: 'whaa@example.com',
  _rev: process.argv[2],
  username: 'johndoe',
  email: 'whaa@example.com',
  access_token: 'some access token'
};

users.update(user, function(err) {
  if (err) {
    console.error(err);
  }
  else {
    console.log('user updated');
  }
});
```

Here we're specifying that the revision ID is given by a command-line argument. Once you find out the current revision ID of your `johndoe` user document, you can use it to invoke this script:

```
$ node user_update_test.js 1-25ee577ef2de8819d642687c38d6b777
user updated
```

Diff doc with last write wins.

Instead of having to specify the entire user document, you can just require that the client specifies which fields are changing:

end of users.js:

```
exports.updateDiff = updateUserDiff;

function updateUserDiff(userDiff, cb) {
  merge();

  function merge() {
    users.get(userDiff._id, errors.wrapNano(function(err, user) {
      if (err) {
        cb(err);
      }
      else {
        extend(user, userDiff);
        schemas.validate(user, 'user', function(err) {
          if (err) {
            cb(err);
          }
          else {
            users.insert(user, errors.wrapNano(done));
          }
        })
      }
    }));

    function done(err) {
      if (err && err.statusCode == 409 && !userDiff._rev) {
        merge(); // try again
      }
      else {
        cb.apply(null, arguments);
      }
    }
  }
}
```

Here our `db/users` module exports a new `updateDiff` function that accepts an incomplete user document, containing only the attributes that have changed. This function starts by declaring this `merge` function, which is responsible for 1) getting the latest version of the given document; 2) applying the given changes to this document; and 3) trying to save it into CouchDB. If this last step has a conflict error (which can

happen when two or more clients are updating the same document concurrently), we try again from the beginning.

> Before retrying we make sure that the user didn't specify the revision ID in his differential document. If they did, this `merge` function would always fail and retry indefinitely because the revision ID is irredeemably outdated.

If the saving of the merged document succeeds, or we cannot recover from an error, we just apply the response arguments into the callback.

Disallowing changes to specific fields

Sometimes you may want to disallow changes to some specific fields in some document types. Let's say that you don't want to allow changes to the email address of a user. Optimally, we would like to be able to easily verify this both in our integral update `users.update` function and also in our partial `users.updateDiff` function. How would we implement such a change to the API flow in a way that's easy to implement for other cases?

What we need is a way to have two different schemas: one for when the user document is being created, and another for when the user document is getting updated. Typically, the updating schema is a subset of the creation schema: the first one is a trimmed-down version of the last.

We need to be able to define two schemas, depending on the operation. Let's then add two schemas to `schemas/user.js` :

schemas/user.js:

```
var extend = require('util')._extend;
var Joi = require('joi');

var updateAttributes = {
  _id: Joi.string(),
  _rev: Joi.string(),
  password: Joi.string().regex(/[a-zA-Z0-9]{3,30}/),
  access_token: [Joi.string(), Joi.number()],
  birthyear: Joi.number().integer().min(1900).max((new
Date()).getFullYear())
};
```

```
exports.update = Joi.object().keys(updateAttributes);

var createAttributes = extend({
  username: Joi.string().alphanum().min(3).max(30).required(),
  email: Joi.string().email()
}, updateAttributes);

exports.create = Joi.object().keys(createAttributes);
```

Here we're exporting one Joi schema for each operation: one for `update` and another for `insert`, the last one extending the first.

Now we need to add an option to the validation functions:

schemas/index.js:

```
var Joi = require('joi');
var Boom = require('boom');

var schemaNames = ['user'];

var schemas = {};

schemaNames.forEach(function(schemaName) {
  schemas[schemaName] = require('./' + schemaName);
});

exports.validate = validate;

function validate(doc, schema, op, cb) {
  if (typeof schema == 'string') {
    schema = schemas[schema];
  }

  if (! schema) {
    cb(new Error('Unknown schema'));
  }
  else {
    schema = schema[op];

    if (! schema) {
      throw new Error('Undefined op ' + op);
    }

    else {
      Joi.validate(doc, schema, function(err, value) {
```

```
          if (err) {
            Boom.wrap(err, 400);
            cb(err);
          }
          else {
            cb(null, doc);
          }
        });
      }
    }
  };

  exports.validating = function validating(schemaName, op, fn) {
    var schema = schemas[schemaName];
    if (! schema) {
      throw new Error('Unknown schema: ' + schemaName);
    }

    return function(doc, cb) {
      validate(doc, schema, op, function(err, doc) {
        if (err) {
          cb(err);
        }
        else {
          fn.call(null, doc, cb);
        }
      });
    };

  };
```

These are all the changes we need in the schema validation layer. Moving on to the database layer, we will need to install a utility module that helps us calculate the difference between two objects:

```
$ npm install object-versions --save
```

Now our `user.update` function has to get a little more complicated. Instead of validating the user document before sending it to CouchDB, it needs to get the current version, calculate the difference, and validate it:

middle of db/users.js:

```
var diff = require('object-versions').diff;

/// Update user

exports.update = updateUser;

function updateUser(user, cb) {
  users.get(user._id, errors.wrapNano(function(err, currentUser) {
    if (err) {
      cb(err);
    }
    else {
      var userDiff = diff(currentUser, user);
      schemas.validate(userDiff, 'user', 'update', function(err) {
        if (err) {
          cb(err);
        }
        else {
          users.insert(user, errors.wrapNano(cb));
        }
      });
    }
  }));
}
```

Our `users.updateDiff` also needs some changes: now that we're able to tell whether a user differential document is valid, we can validate it before merging the current document with the diff document:

end of db/users.js:

```
exports.updateDiff = updateUserDiff;

function updateUserDiff(userDiff, cb) {
  schemas.validate(userDiff, 'user', 'update', function(err) {
    if (err) {
      cb(err);
    }
    else {
      merge();
    }
  });

  function merge() {
    users.get(userDiff._id, errors.wrapNano(function(err, user) {
      if (err) {
        cb(err);
```

```
        }
        else {
          extend(user, userDiff);
          users.insert(user, errors.wrapNano(done));
        }
      }));

      function done(err) {
        if (err && err.statusCode == 409 && !userDiff._rev) {
          merge(); // try again
        }
        else {
          cb.apply(null, arguments);
        }
      }
    }
  }
```

Views

Up until now we have used CouchDB as a key-value store: we just index each document by its key. Unlike other databases that let you do slow queries that don't use indexes, CouchDB won't let you. If you want to search for a document or a set of documents using anything other than the document identifier, you will have to create a CouchDB view.

In essence, a CouchDB view is a transformation of a database into another database. The transformation is defined by some JavaScript functions that take each document as it gets inserted or updated and maps it into an alternative key and value. CouchDB stores the views in the same way that it stores a normal database, by using a file-based index that differs in just one main thing: it allows you to store more than one document for a given key.

Let's see some uses for CouchDB views:

Inverted indexes

In CouchDB we can search for documents where a specific attribute is equal to a given value. For that we'll have to create a specific view.

Let's say that, for instance, you want to search for messages that were addressed to a given user.

```
$ npm install deep-equal --save
```

Now we're going to create a directory where we will store all the CouchDB views, one file per database.

```
$ mkdir views
```

Let's create the one for the `messages` database:

views/messages.js:

```
exports.by_to = {
  map: function(doc) {
    if (doc.to) {
      emit(doc.to, {_id: doc._id});
    }
  }
};
```

This is a CouchDB view: it contains a map function that will run inside CouchDB. This function will be called each time there is an updated or a new message document. It receives the document as the sole argument, and then it uses the `emit` function to write changes to the view. The first argument of the `emit` function is the index key and the second argument is the value. In this case we're specifying that the key is the `to` attribute of the message, and that the emitted doc is one document containing only one `_id` field.

> We could emit the whole document, but here we're only emitting a document with an `_id` field. This is an optimisation: in this case CouchDB will use the `_id` field to look up and get the referenced document when we're consulting the view.

CouchDB stores the views as special documents. These are called *design documents*, and they're all prefixed by the `_design/` path. Now we need a module that takes the views' definitions and sends them to CouchDB.

> CouchDB design documents are also used for things other than views, but we're not going to use these here.

top part of views/index.js:

```
var async = require('async');
var equal = require('deep-equal');
var couch = require('../couchdb');

var databaseNames = ['messages'];

var views = {};

databaseNames.forEach(function(database) {
  views[database] = require('./' + database);
});

exports.populate = function populate(cb) {
  async.each(databaseNames, populateDB, cb);
};
```

Here we're just showing the top part of the `views/index.js` file. This file exports a `populate` function that will ensure that the views in CouchDB are up to date. When we call this function, `populate` uses `async.each` to call `populateDB` for each database.

Here is `populateDB`:

bottom part of views/index.js:

```
function populateDB(dbName, cb) {
  var db = couch.use(dbName);
  var dbViews = views[dbName];

  async.eachSeries(Object.keys(dbViews), ensureView, cb);

  function ensureView(viewName, cb) {
    var view = dbViews[viewName];

    var ddocName = '_design/' + viewName;
    db.get(ddocName, function(err, ddoc) {
      if (err && err.statusCode == 404) {
        insertDDoc(null, cb);
```

```
        }
        else if (err) {
          cb(err);
        }
        else if (equal(ddoc.views[viewName], view)) {
          cb();
        }
        else {
          insertDDoc(ddoc, cb);
        }
      });

      function insertDDoc(ddoc, cb) {
        if (! ddoc) {
          ddoc = {
            language: 'javascript',
            views: {}
          };
        }

        ddoc.views[viewName] = view;

        db.insert(ddoc, ddocName, function(err) {
          if (err && err.statusCode == 409) {
            ensureView(viewName, cb);
          }
          else {
            cb(err);
          }
        });
      }
    }
  }
```

This function fetches the views we defined for a given database and calls the `ensureView` function for each. This last function tries to get the design document. If it doesn't exist, it calls the `insertDDoc` function. Otherwise, if it exists, it uses the `deep-equal` module we just installed to check whether the view is up to date. If the view coming from CouchDB needs updating, it calls `insertDDoc`.

The `insertDDoc` function then creates or updates the design document, attaching it the latest version of the view definition. If there is a conflict on updating it, it tries to repeat the operation.

Now we need to change our `init_couch.js` module to populate the views after we have ensured the databases exist:

top of init_couch.js:

```
var async = require('async');
var couch = require('./couchdb');
var views = require('./views');

var databases = ['users', 'messages'];

module.exports = initCouch;

function initCouch(cb) {
  async.series([createDatabases, createViews], cb);
}

function createDatabases(cb) {
  async.each(databases, createDatabase, cb);
}

function createViews(cb) {
  views.populate(cb);
}

//...
```

Now we can run our simulated application bootstrap procedure in `app.js`:

```
$ node app.js
couchdb initialized
```

Before we can query our messages database, we must first create our database layer module:

db/messages.js:

```
var extend = require('util')._extend;
var schemas = require('../schemas');
var errors = require('../errors');
```

```
var messages = require('../couchdb').use('messages');

/// Create user

exports.create = schemas.validating('message', 'create',
createMessage);

function createMessage(message, cb) {
  message.createdAt = Date.now();
  messages.insert(message, errors.wrapNano(cb));
}
```

This file is similar to the `db/users.js` one, except that it only exports the `create` method.

Now we need to define a message document schema:

schemas/messages.js:

```
var Joi = require('joi');

var createAttributes = {
  from: Joi.string().email().required(),
  to: Joi.string().email().required(),
  subject: Joi.string().max(120).required(),
  body: Joi.string().max(1024).required(),
  createdAt: Joi.date()
};

exports.create = Joi.object().keys(createAttributes);
```

… and add it to the schemas list:

top of schemas/index.js:

```
var Joi = require('joi');
var Boom = require('boom');

var schemaNames = ['user', 'message'];
// ...
```

Next, we need to create a script that inserts some message documents:

messages_insert.js:

```
var extend = require('util')._extend;
var messages = require('./db/messages');

var message = {
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body'
};

var count = 10;
var left = count;

for(var i = 1 ; i <= count ; i ++) {
  messages.create(message, created);
}

function created(err) {
  if (err) {
    throw err;
  }
  if (-- left == 0) {
    console.log('%d messages inserted', count);
  }
}
```

This script creates 10 messages for our user. Let's run it:

```
$ node messages_insert
10 messages inserted
```

Query

Now we need to find a way, using this view, to get all the messages sent to a particular user. Let's add this method to `db/messages.js` :

bottom of db/messages.js:

```
/// Messages for a given user

exports.getFor = getMessagesFor;

function getMessagesFor(user, cb) {
  messages.view(
    'by_to', 'by_to', {keys: [user], include_docs: true},
    errors.wrapNano(function(err, result) {
      if (err) {
        cb(err);
      }
      else {
        result = result.rows.map(function(row) {
          return row.doc;
        });
        cb(null, result);
      }
    })
  );
}
```

This new message method uses the `db.view` method of nano to query a view. The first argument to this method is the design document name, and the second is the view name. In our case these two are equal — we create a design document named after the view for each.

After that we have some view arguments in an object: first, the `keys` argument contains all the keys we are looking for. In our case, we're looking for only one key, which value is the user ID. Next, we set the `include_docs` argument to `true` — this makes CouchDB fetch the document referenced in the `_id` field of the view records.

> This is why we only emitted one document with a single `_id` attribute: by setting the `include_docs` argument to `true`, we make CouchDB also fetch the referred document.

When the result comes, we need to fetch the documents from the `rows` attribute from it and, for each element of this array, fetch the document that resides inside the `doc` attribute.

Now we can create a small script to query the messages for a given user:

get_messages.js:

```
var user = process.argv[2];

if (! user) {
  console.error('please specify user');
  return;
}

var messages = require('./db/messages');

messages.getFor(user, function(err, messages) {
  if (err) {
    throw err;
  }

  console.log('messages for user %s:', user);
  messages.forEach(printMessage);
});

function printMessage(message) {
  console.log(message);
}
```

We can now query all the messages for our beloved user by doing:

```
$ node get_messages.js whaa@example.com
messages for user whaa@example.com:
{ _id: '712f741349de658d85795fffb4015103',
  _rev: '1-54e5f503f3ecbf537978a9d7adc6ce03',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: '2015-01-27T15:10:28.256Z' }
{ _id: '712f741349de658d85795fffb40151ce',
  _rev: '1-035b12416b21c1705eddfd82defc795d',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: '2015-01-27T15:10:28.260Z' }
...
```

Multi-value inverted indexes

The previous view had at least one problem: the view doesn't sort the messages for a given user by creation time — the order is undefined. CouchDB sorts by the keys, and in this case we have the same key for all the messages for a given user: the user ID. What we would like is to be able to filter by the value in the `to` property and then order by the `createdAt` property. Let's then create a new view that allows that:

bottom of views/messages.js:

```
exports.by_to_createdAt = {
  map: function(doc) {
    if (doc.to && doc.createdAt) {
      emit([doc.to, doc.createdAt], {_id: doc._id});
    }
  }
};
```

This new view emits a different type of key: instead of a string, we emit an array — CouchDB will treat an array key as a composed key, and will be able to sort it by the order of the elements, which is just what we need. Let's now create that view definition in CouchDB:

```
$ node app
couchdb initialized
```

Now we need to change our query implementation to use this view:

bottom of db/messages.js:

```
/// Messages for a given user

exports.getFor = getMessagesFor;

function getMessagesFor(user, cb) {
  messages.view(
    'by_to_createdAt', 'by_to_createdAt',
    {
      startkey: [user, 0],
      endkey: [user, Date.now()],
```

```
      include_docs: true
    },
    errors.wrapNano(function(err, result) {
      if (err) {
        cb(err);
      }
      else {
        result = result.rows.map(function(row) {
          return row.doc;
        });
        cb(null, result);
      }
    })
  );
}
```

Now we're passing different arguments into the CouchDB view: instead of passing a `keys` array, we're specifying that we want a range by specifying the `startkey` and the `endkey` arguments. The first one contains the minimum value of the keys it will be finding, and the second one contains the maximum one. Since we want to get the records for a given user, we always specify the same user in the first position of the key array, but we let the second position vary between 0 (the start of the computer's time) and the current timestamp. This query returns us all the messages created up until now that have a given user as the recipient.

We can now test this using our `get_messages` script from the command line as before:

```
$ node get_messages.js whaa@example.com
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d49818000152',
  _rev: '1-ab4a6fbdc966e6644fa7f470c3d2f414',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090485 }
{ _id: '89f2204c421281219758d4981800044b',
  _rev: '1-b17d04a94cfcd70b83e6b68707a59e58',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090489 }
...
```

Wait — but this query is returning the results in ascending timestamp order, and we probably want to present the most recent message first. Let's then change our query to reverse the order:

bottom part of db/messages.js:

```
function getMessagesFor(user, cb) {
  messages.view(
    'by_to_createdAt', 'by_to_createdAt',
    {
      startkey: [user, Date.now()],
      endkey: [user, 0],
      descending: true,
      include_docs: true
    },
    errors.wrapNano(function(err, result) {
      if (err) {
        cb(err);
      }
      else {
        result = result.rows.map(function(row) {
          return row.doc;
        });
        cb(null, result);
      }
    })
  );
}
```

Here we switched the value of `startkey` with `endkey` and set the `descending` argument to true. (It defaults to `false`.) Now you can see that the messages are being returned in reverse chronological order:

```
$ node get_messages.js whaa@example.com
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d49818002e6c',
  _rev: '1-76ed2ec67fadc424d6f7cfd1cd1327e9',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090494 }
{ _id: '89f2204c421281219758d49818002b17',
  _rev: '1-76ed2ec67fadc424d6f7cfd1cd1327e9',
```

```
    from: 'someone@example.com',
    to: 'whaa@example.com',
    subject: 'Test 123',
    body: 'Test message body',
    createdAt: 1422438090494 }
...
```

> CouchDB views are *materialised* views, which means that they're generated ahead of time; which means that, when you create or modify a view, CouchDB has to (re)generate the whole view. While CouchDB does that, your database may become irresponsive; view creation and change to big datasets has to be done with great care, since it may imply some database down-time.

Paginating results

There's yet another limitation with our query: we get the entire history of messages. A user interface displaying the messages would show only one page of messages at a time, allowing the user to cycle through pages.

The wrong way of doing pagination

Let's try to implement message pagination then:

bottom of db/messsages.js:

```
function getMessagesFor(user, page, maxPerPage, cb) {
  messages.view(
    'by_to_createdAt', 'by_to_createdAt',
    {
      startkey: [user, Date.now()],
      endkey: [user, 0],
      descending: true,
      include_docs: true,
      limit: maxPerPage,
      skip: page * maxPerPage
    },
    errors.wrapNano(function(err, result) {
      if (err) {
        cb(err);
      }
      else {
        result = result.rows.map(function(row) {
          return row.doc;
        });
        cb(null, result);
```

```
        }
      })
  );
}
```

Now our `getMesssagesFor` function accepts two additional arguments: the page number and the maximum number of messages per page. This allows us to calculate how many records CouchDB should be skipping before it reaches the first record of the page we need.

Let's change our `get_messages.js` script to accept these new arguments from the command line and apply them to the new version of the `messages.getFor` function:

get_messages.js:

```
var user = process.argv[2];

if (! user) {
  console.error('please specify user');
  return;
}

var start = Number(process.argv[3]) || 0;
var maxPerPage = Number(process.argv[4]) || 4;

var messages = require('./db/messages');

messages.getFor(user, page, maxPerPage, function(err, messages) {
  if (err) {
    throw err;
  }

  console.log('messages for user %s:', user);
  messages.forEach(printMessage);
});

function printMessage(message) {
  console.log(message);
}
```

Here we're using a maximum number of items per page of four if it's not specified in the command line arguments.

Let's test this then:

```
$ node get_messages.js whaa@example.com
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d49818002e6c',
  _rev: '1-76ed2ec67fadc424d6f7cfd1cd1327e9',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090494 }
{ _id: '89f2204c421281219758d49818002b17',
  _rev: '1-76ed2ec67fadc424d6f7cfd1cd1327e9',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090494 }
...

$ node get_messages.js whaa@example.com 1
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d498180019ef',
  _rev: '1-145100821a440accea8c7127fd7ed3ef',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090491 }
{ _id: '89f2204c421281219758d498180018c5',
  _rev: '1-145100821a440accea8c7127fd7ed3ef',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090491 }
...
```

Looks like it's working.

This approach has one problem though: CouchDB stores the index in a B-Tree and will be scanning all the elements that are to be skipped. This means that the performance of this query will decrease as we get more pages; CouchDB will have to count and skip more records.

A better way of paginating

So what's the alternative? Instead of telling CouchDB how many records to skip, we should be telling CouchDB which record key to begin at. Here is how we can implement that:

bottom of db/messages.js:

```
function getMessagesFor(user, startKey, maxPerPage, cb) {
  messages.view(
    'by_to_createdAt', 'by_to_createdAt',
    {
      startkey: [user, startKey],
      endkey: [user, 0],
      descending: true,
      include_docs: true,
      limit: maxPerPage + 1
    },
    errors.wrapNano(function(err, result) {
      if (err) {
        cb(err);
      }
      else {
        result = result.rows.map(function(row) {
          return row.doc;
        });

        if (result.length > maxPerPage) {
          // remove the last record
          var next = result.pop().createdAt;
        }

        cb(null, result, next);
      }
    })
  );
}
```

Now our `getMessagesFor` function accepts a start key instead of a page number. We use this start key as the past part of the `startkey` parameter we send to CouchDB, allowing it to jump to the correct first record immediately.

We're then requesting one more document than what the user requested. This allows us to calculate the start key of the next page. We then pop the last doc from the result set and pass its key into the result callback.

Let's see how a client can now implement pagination using this:

get_messages.js:

```
var user = process.argv[2];

if (! user) {
  console.error('please specify user');
  return;
}

var start = Number(process.argv[3]) || Date.now();
var maxPerPage = Number(process.argv[4]) || 4;

var messages = require('./db/messages');

messages.getFor(user, start, maxPerPage, function(err, messages,
next) {
  if (err) {
    throw err;
  }

  console.log('messages for user %s:', user);
  messages.forEach(printMessage);

  console.log('\nNext message ID is %s', next);
});

function printMessage(message) {
  console.log(message);
}
```

In addition to printing the messages, we also print the ID of the next message. Let's see this in action:

Request the first page:

```
$ node get_messages.js whaa@example.com
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d49818002e6c',
  _rev: '1-76ed2ec67fadc424d6f7cfd1cd1327e9',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
```

```
  createdAt: 1422438090494 }
...

Next message ID is 1422438090491
```

Now we have an ID we can use to get to the next page. Let's use it:

```
$ node get_messages.js whaa@example.com 1422438090491
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d498180019ef',
  _rev: '1-145100821a440accea8c7127fd7ed3ef',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090491 }
...

Next message ID is 1422438090489
```

Since we get four records per page, our next page will have two records and no message ID. Let's verify that:

```
node get_messages.js whaa@example.com 1422438090489
messages for user whaa@example.com:
{ _id: '89f2204c421281219758d4981800044b',
  _rev: '1-b17d04a94cfcd70b83e6b68707a59e58',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090489 }
{ _id: '89f2204c421281219758d49818000152',
  _rev: '1-ab4a6fbdc966e6644fa7f470c3d2f414',
  from: 'someone@example.com',
  to: 'whaa@example.com',
  subject: 'Test 123',
  body: 'Test message body',
  createdAt: 1422438090485 }

Next message ID is undefined
```

> So how do you create a link to the previous page? You will have to keep the previous start key around, passing it in the URL. This approach has one limitation: it doesn't allow you to jump to a page number.
>
> For the ID of the message we're using the timestamp. A timestamp has a resolution of milliseconds. If we have more than one message being created during the same timestamp, our pagination scheme won't work. To remedy this, we need to tell which exact record to start at by specifying the `startdocid` view argument. This means that you will also have to pass this argument from the client to the query, and that the query result should also pass the first message ID of the next page to the client.

Reducing

Views are implemented by specifying a map function and also an optional reduce function. This reduce function can be used to, as the name says, somehow reduce the number of records stored in this view.

Let's build on our messages example and create a view that calculates the number of messages in a given user's inbox.

bottom of views/messages.js:

```
exports.to_count = {
  map: function(doc) {
    if (doc.to) {
      emit(doc.to, 1);
    }
  },
  reduce: function(keys, values) {
    return sum(values);
  }
}
```

This view now has a reduce function. This reduce function uses the CouchDB built-in `sum` function to return the sum of the given values. We start out by mapping each message to the value `1`, which we then get on the values in the reduce function. Our reduce function can be called iteratively and recursively, each time just blindly summing the values.

We can now query this view to find out how many messages a given user has addressed to them:

bottom of db/messages.js:

```
/// Count messages for a given user

exports.countFor = countMessagesFor;

function countMessagesFor(user, cb) {
  messages.view('to_count', 'to_count', {
    keys: [user],
    group: true
  }, errors.wrapNano(function(err, result) {
    if (err) {
      cb(err);
    }
    else {
      cb(null, result.rows[0].value);
    }
  }));
};
```

This view query now queries a specific key (the user ID), but tells it to use the reduced values by setting the `group` argument to `true`. We then expect the result to have only one row, from which we extract the value.

Now our `get_messages.js` client can query the number of messages to present it before getting the messages:

get_messages.js:

```
var user = process.argv[2];

if (! user) {
  console.error('please specify user');
  return;
}

var start = Number(process.argv[3]) || Date.now();
var maxPerPage = Number(process.argv[4]) || 4;
```

```
var messages = require('./db/messages');

messages.countFor(user, function(err, count) {
  if (err) {
    throw err;
  }

  console.log('%s has a total of %d messages.', user, count);

  messages.getFor(user, start, maxPerPage, function(err, messages,
next) {
    if (err) {
      throw err;
    }

    console.log('messages for user %s:', user);
    messages.forEach(printMessage);

    console.log('\nNext message ID is %s', next);
  });

  function printMessage(message) {
    console.log(message);
  }
});
```

Let's test this:

```
$ node get_messages.js whaa@example.com
whaa@example.com has a total of 10 messages.
...
```

Using the Changes Feed

A CouchDB database has the amazing ability to provide a feed of all the changes it has gone through over time. This changes feed is what lies behind CouchDB's replication mechanism, but you can use it for many other things.

For instance, in our users-and-messages system, we can use the changes feed of the `messages` database to have a separate worker sending notification emails to the recipient of each message. Let's see how we could implement that:

First we will have to install the `follow` NPM package, which allows us to get the changes feed of a CouchDB database.

```
$ npm install follow --save
```

Now let's create an email-sending worker that listens to the changes feed from the `messages` database and sends emails. For that we will create a `workers` directory:

```
$ mkdir workers
```

Let's now create our worker:

workers/messages.sendmail.js:

```
var follow = require('follow');
var couch = require('../couchdb');
var messages = couch.use('messages');
var messages = couch.use('messages');

var feed = follow({
  db: couch.config.url + '/' + 'messages',
  include_docs: true
}, onChange);

feed.filter = function filter(doc) {
  return doc._id.indexOf('_design/') != 0 && !doc.notifiedRecipient;
};

function onChange(err, change) {
  if (err) {
    console.error(err);
  }
  else {
    console.log('change:', change);
    feed.pause();
    var message = change.doc;
    sendEmail(message, sentEmail);
  }

  function sentEmail(err) {
    if (err) {
```

```
        console.error(err);
      }
      else {
        message.notifiedRecipient = true;
      }
      messages.insert(message, savedMessage);
    }
  }

  function sendEmail(message, cb) {
    // Fake send email
    setTimeout(cb, randomTime(1e3));
  }

  function savedMessage(err) {
    if (err) {
      console.error(err);
    }
    feed.resume();
  }

  function randomTime(max) {
    return Math.floor(Math.random() * max);
  }
```

The worker starts out (using the `follow` package we just installed), by creating a feed on the `messages` database.

This feed object can be configured with a filter that defines whether a certain document change is interesting to us or not. In our case we're not interested in design documents (ones whose ID begins with `_design/`), and messages that we have marked with a `notifiedRecipient` property. (As you will see later, this property is `true` for all messages that have been successfully sent.)

Each change we get will be handled by our `onChange` function. This function starts by pausing the feed and then sending the email.

> Here we're using a fake email-sending function that just calls the callback after a random amount of time (smaller than one second) has passed.

Once the email has been sent, the `sentEmail` function gets called. We take this chance to flag the message as having been sent by setting the `notifiedRecipient` property to `true`.

We then persist the message into the database.

After saving the message we resume the feed, and the worker gets the next pending message if there is any, restarting the work cycle. If there is no pending change, the feed will sit waiting for changes.

Let's test this worker:

```
$ node workers/messages.sendmail.js
```

You should start to see a series of messages being processed, and then the process waits for more relevant changes.

Minimising the chance of repeated jobs

There's at least one problem with this set-up: there is the slight chance that a duplicate email will get sent:

If a worker process shuts down after sending the email, but before having the chance to save the message, the message stats will have been lost. Once the worker comes back up, this message will be picked up again by the changes feed, it will be selected by the filter, and a second email will be sent. There are several ways to minimise this risk.

The first way is for the worker process to have a signal handler. By listening to `SIGINT`, we can catch attempts to kill the worker process and react accordingly:

```
var working = false;
var quit = false;

process.once('SIGINT', function() {
  console.log('shutting down...');
  if (! working) {
    process.exit();
  }
  else {
    quit = true;
  }
});
```

We can set the `working` flag to `true` when we get a change:

```
//...
function onChange(err, change) {
  if (err) {
    console.error(err);
  }
  else {
    console.log(change);
    working = true;
    feed.pause();
    var message = change.doc;
    sendEmail(message, sentEmail);
  }
//...
```

…and reset it when a message is finished, also quitting the process if necessary:

```
//...
function savedMessage(err) {
  if (err) {
    console.error(err);
  }
  if (quit) {
    process.exit();
  }
  else {
    working = false;
    feed.resume();
  }
}
//...
```

> This pattern works when you run separate processes for separate workers. If you absolutely need to have more than one worker running in the same process, you will need to coordinate the shutdown procedure between them.

Anyway, this scheme won't work if your process dies abruptly without the chance to catch a `SIGINT` signal. If you need to handle this case, a proper queuing service (covered by another book in this series) should be used.

Recording the sequence

If you need to restart the worker process, the changes feed starts from the beginning of the database history. In this case, our filter function will filter out all the messages that have already been sent (the ones that have the `notifiedRecipient` property set to `true`), but it may take our feed to get past all the messages that have been processed. But there is a way around that.

Each change to a CouchDB database contains a sequence number. The first change to a database creates a change with sequence number 1, and it keeps increasing with every change you make. When you get the changes feed, each change is identified by that sequence number. Once the work is done, you can somehow record that sequence. When the process comes up, you start by querying that sequence. If it exists, you use it to specify the point from which the feed should start.

In this case we will use our CouchDB server to store the last processed sequence. It could alternatively be saved in a local file, but then we would have to periodically back up that file.

Then we will need to create a database where we will store the worker sequences:

top of init_couch.js:

```
var async = require('async');
var couch = require('./couchdb');
var views = require('./views');

var databases = ['users', 'messages', 'workersequences'];
//...
```

Then we need to create this database by running the app initialisation:

```
$ node app
couchdb initialized
```

Next, we will need to query the sequence before starting the feed:

middle of workers/messages.sendmail.js:

```
//...
var workerSequences = couch.use('workersequences');

workerSequences.get('messages.sendmail', function(err, sequence) {
  if (! sequence) {
    sequence = {
      _id: 'messages.sendmail',
      since: 0
    };
  }
  console.log('last sequence:', sequence);

  var feed = follow({
    db: couch.config.url + '/' + 'messages',
    include_docs: true,
    since: sequence.since
  }, onChange);
//...
```

Here we're using the `since` parameter to the `follow` feed constructor, specifying that we want to use the saved sequence. If no sequence has been saved, we create a `sequence` object where the `since` attribute is 0, which will make the feed start from the beginning of the history.

Next we need to update the sequence number when we get a change:

```
//...
function onChange(err, change) {
  if (err) {
    console.error(err);
  }
  else {
    console.log(change);
    sequence.since = change.seq;
//...
```

Now we need to save the sequence when we finish processing a message:

```
//...
function savedMessage(err) {
  if (err) {
    console.error(err);
  }
  if (quit) {
    process.exit();
  }
  else {
    saveSequence();
  }
}

function saveSequence() {
  workerSequences.insert(sequence, savedSequence);
}

function savedSequence(err, result) {
  if (err) {
    throw(err);
  }

  sequence._rev = result.rev;
  working = false;
  feed.resume();
}
//...
```

Here we're making sure that we update the new revision number on the `sequence` object after we saved it to prevent CouchDB declaring a conflict.

> You may have noticed that we're throwing an error if we get an error saving the sequence. This may need some refinement, but it's mainly because the error is almost certainly caused by a CouchDB conflict. A conflict may arise if you're running more than one worker process, in which case it's good that we throw and stop: this set-up doesn't support multiple worker processes of the same type.

Scaling: how to support more than one job in parallel

Using this set-up we can only have one worker process. This can be OK if the feed frequency is not too high; but if that's not the case, we have two choices. The first one is feasible if, and only if, the type of work is I/O-intensive (as was the case of sending emails). If that's the case, we can easily support more than one pending message waiting

to be processed at the same time, which will increase the overall throughput of one single worker.

To support more than one worker we need to make a set of considerable changes. First, we will be having individual emails being sent in parallel that can finish in any order. We must be sure not to save a sequence number that is higher than any pending change, or else we may lose data. To enable this we will use a sorted list where we will store all the sequences that are pending. Let's install an NPM package that allows us to have a sorted list:

```
$ npm install sortedlist --save
```

Next we will need to create a sorted list that will contain all the pending sequences:

top of workers/messages.sendmail.parallel.js:

```
var follow = require('follow');
var couch = require('../couchdb');
var messages = couch.use('messages');
var SortedList = require('sortedlist');

var pendingSequences = SortedList.create();
//...
```

After that we need to define a variable that will hold the number of messages currently pending:

```
//...
var worker = 'messages.sendmail';

var maxParallel = 5;
var pending = 0;
//...
```

Next we need to update the `SIGINT` signal handler accordingly:

```
//...
var quit = false;

process.once('SIGINT', function() {
  console.log('shutting down...');
  if (! pending) {
    process.exit();
  }
  else {
    quit = true;
  }
});
//...
```

When starting up, we need to query the last known sequence ID and start the feed, somewhat similar to before:

```
//...
var workerSequences = couch.use('workersequences');

workerSequences.get(worker, function(err, sequence) {

  var since = sequence && sequence.since || 0;

  console.log('since:', since);
  var feed = follow({
    db: couch.config.url + '/' + 'messages',
    include_docs: true,
    since: since
  }, onChange);
//...
```

The feed filter function remains unchanged:

```
//...
  feed.filter = function filter(doc) {
    return doc._id.indexOf('_design/') != 0 &&
!doc.notifiedRecipient;
  };
//...
```

The change handler needs to insert the change sequence into the sorted list of pending sequences:

```
//...
  function onChange(err, change) {
    if (err) {
      console.error(err);
    }
    else {
      console.log(change);
      pendingSequences.insert(change.seq);
      pending ++;
      maybePause();
      var message = change.doc;
      sendEmail(message, sentEmail);
    }
//...
```

Note that we're now using a function called `maybePause` (which we define later), that will pause the feed if the number of pending messages has reached the maximum defined in `maxParallel` (bluntly hard-coded to 5 in our case).

The `sentEmail` function remains unchanged:

```
function sentEmail(err) {
      if (err) {
        console.error(err);
      }
      else {
        message.notifiedRecipient = true;
      }
      messages.insert(message, savedMessage);
    }
```

But the `savedMessage` callback function now calls `maybeSaveSequence`, which is then responsible for saving the sequence number to CouchDB if, and only if, the current job is the pending job with the smallest sequence:

```
//...
    function savedMessage(err) {
```

```
      if (err) {
        console.error(err);
      }
      maybeSaveSequence();
    }

    function maybeSaveSequence() {
      var pos = pendingSequences.key(change.seq);
      pendingSequences.remove(pos);
      if (pos == 0) {
        saveSequence();
      }
      else {
        savedSequence();
      }
    }

    function saveSequence() {
      workerSequences.get(worker, function(err, sequence) {
        if (! sequence) {
          sequence = {
            _id: worker,
            since: 0
          };
        }
        if (sequence.since < change.seq) {
          sequence.since = change.seq;
          workerSequences.insert(sequence, savedSequence);
        }
        else {
          savedSequence();
        }
      });
    }
//...
```

Since now there is the possibility of concurrent sequence updates, the `savedSequence` callback should now handle a conflict error by retrying to save the sequence:

```
function savedSequence(err) {
      if (err && err.statusCode == 409) {
        saveSequence();
      }
      else if (err) {
        throw(err);
      }
      else {
        pending --;
        console.log('PENDING: %d', pending);
```

```
          maybeQuit();
          maybeResume();
        }
      }
    }
  //...
```

This function now calls `maybeQuit`, which detects whether we need to quit. (We need to quit if we caught a `SIGINT` signal and we no longer have pending messages.) It also calls the `maybeResume` function, which resumes the feed if we're not quitting and we still have room for more parallel operations.

Here is the rest of the file, containing the implementation of the fake email-sending (the same as before) and the `maybe...` functions:

```
function sendEmail(message, cb) {
    // Fake send email
    setTimeout(cb, randomTime(1e3));
  }

  function maybePause() {
    if (quit || pending > maxParallel) {
      feed.pause();
    }
  }

  function maybeResume() {
    if (!quit && pending < maxParallel) {
      feed.resume();
    }
  }

  function maybeQuit() {
    if (quit && !pending) {
      process.exit();
    }
  }

  function randomTime(max) {
    return Math.floor(Math.random() * max);
  }
});
```

Here is the complete file for your delight:

workers/messages.sendmil.parallel.js:

```
var follow = require('follow');
var couch = require('../couchdb');
var messages = couch.use('messages');
var SortedList = require('sortedlist');

var pendingSequences = SortedList.create();

var worker = 'messages.sendmail';

var maxParallel = 5;
var pending = 0;
var quit = false;

process.once('SIGINT', function() {
  console.log('shutting down...');
  if (! pending) {
    process.exit();
  }
  else {
    quit = true;
  }
});

var workerSequences = couch.use('workersequences');

workerSequences.get(worker, function(err, sequence) {

  var since = sequence && sequence.since || 0;

  console.log('since:', since);
  var feed = follow({
    db: couch.config.url + '/' + 'messages',
    include_docs: true,
    since: since
  }, onChange);

  feed.filter = function filter(doc) {
    return doc._id.indexOf('_design/') != 0 &&
!doc.notifiedRecipient;
  };

  function onChange(err, change) {
    if (err) {
      console.error(err);
    }
    else {
      console.log(change);
      pendingSequences.insert(change.seq);
```

```
  pending ++;
  maybePause();
  var message = change.doc;
  sendEmail(message, sentEmail);
}

function sentEmail(err) {
  if (err) {
    console.error(err);
  }
  else {
    message.notifiedRecipient = true;
  }
  messages.insert(message, savedMessage);
}

function savedMessage(err) {
  if (err) {
    console.error(err);
  }
  maybeSaveSequence();
}

function maybeSaveSequence() {
  var pos = pendingSequences.key(change.seq);
  pendingSequences.remove(pos);
  if (pos == 0) {
    saveSequence();
  }
  else {
    savedSequence();
  }
}

function saveSequence() {
  workerSequences.get(worker, function(err, sequence) {
    if (! sequence) {
      sequence = {
        _id: worker,
        since: 0
      };
    }
    if (sequence.since < change.seq) {
      sequence.since = change.seq;
      workerSequences.insert(sequence, savedSequence);
    }
    else {
      savedSequence();
    }
  });
}
```

```
    function savedSequence(err) {
      if (err && err.statusCode == 409) {
        saveSequence();
      }
      else if (err) {
        throw(err);
      }
      else {
        pending --;
        console.log('PENDING: %d', pending);
        maybeQuit();
        maybeResume();
      }
    }
  }

  function sendEmail(message, cb) {
    // Fake send email
    setTimeout(cb, randomTime(1e3));
  }

  function maybePause() {
    if (quit || pending > maxParallel) {
      feed.pause();
    }
  }

  function maybeResume() {
    if (!quit && pending < maxParallel) {
      feed.resume();
    }
  }

  function maybeQuit() {
    if (quit && !pending) {
      process.exit();
    }
  }

  function randomTime(max) {
    return Math.floor(Math.random() * max);
  }
});
```

Balancing work: how to use more than one worker process

This set-up still doesn't allow us to use more than one worker process: if we spawn two of them, both will try to perform the same work, which in this case results in duplicate email messages.

To allow this you can either a) resort to a proper distributed message queue (discussed in another book of this series), or b) distribute the work amongst processes by splitting the workload.

Unfortunately, implementing the second strategy with our set-up is not trivial. There are at least two complicated problems: work sharding and saving sequences.

One way of distributing the work is by dividing the message ID space between workers. For instance, if you have two workers, one could be responsible for handling messages with an even message ID, and the other could be responsible for the odd message IDs. You would need to change the change filter to something like this:

```
var workerCount = Number(process.env.WORKER_COUNT);
var workerID = Number(process.env.WORKER_ID);

feed.filter = function filter(doc) {
  var id = Buffer(doc._id, 'hex');
  var forWorker = id[id.length - 1] % workerCount == workerID;
  return forWorker && doc._id.indexOf('_design/') != 0 &&
!doc.notifiedRecipient;
};
```

Here we're using environment variables to assign a different worker ID to each worker process.

One problem with this happens when you want to introduce another worker: you will first have to shut down all the workers, update the `WORKER_COUNT` environment variable on each, and then start each one.

The second problem is about saving sequences: each worker will have to save a sequence separately from all the other workers, to guarantee that one worker saving a higher sequence ID will not clobber another pending message, which can eventually lead to missing messages if a worker process restarts.

All in all, if you absolutely need to distribute work between processes, it's better that you stick with a traditional distributed work queue (discussed in another book in this series).

Written by Pedro Teixeira (extracted from **Databases-Volume I**, Node Patterns series) — published for YLD.

. . .

Interested in Node? Read more about it:

Using RabbitMQ and AMQP for Distributed Work Queues in Node.js

This is the last article in the Work Queue Patterns series. In the past article, we examined how to manage distributed...

medium.com

Node.js databases: using Redis for fun and profit

Redis is an open-source database server that has been gaining popularity recently. It's an in-memory key-value store...

medium.com

Nodejs JavaScript Database Couchdb Open Source