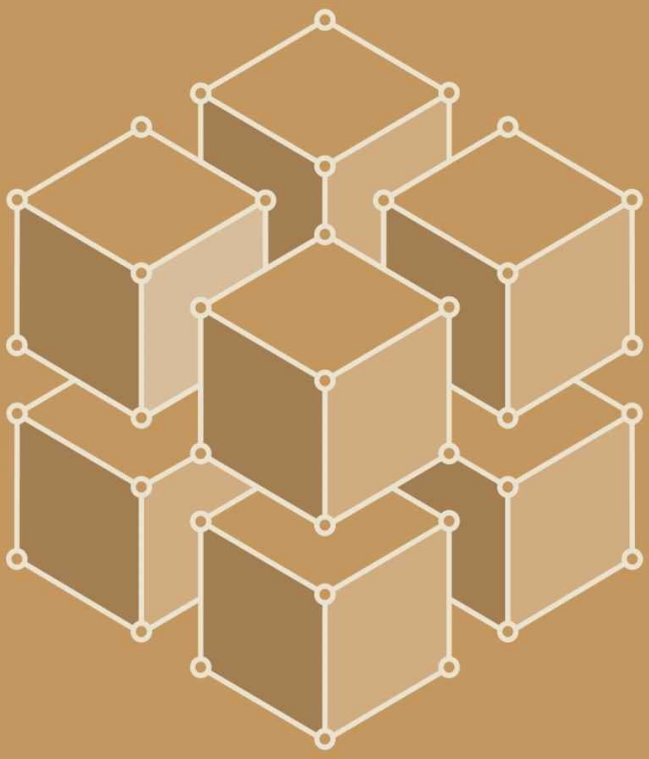
YLD publishing presents



DATABASE PATTERNS

VOLUME I

LevelDB, Redis and CouchDB

Pedro Teixeira

Database Patterns

Pedro Teixeira

This book is for sale at http://leanpub.com/databasepatterns

This version was published on 2015-02-11

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

1. The source code

You can find the source code for the entire series of books at the following URL:

https://github.com/pgte/node-patterns-code

You can browse the code for this particular book here:

https://github.com/pgte/node-patterns-code/tree/master/06-database-patterns

You are free to download it, try it and tinker with it. If you have any issue with the code, please submit an issue on Github. Also, pull requests with fixes or recommendations are welcome!

2. Introduction

Node.js has been designed to do quick and efficient network I/O. Its event-driven streams make it ideal to be used as a kind of smart proxy, often working as the glue between back-end systems and clients. Node was originally designed with that intention in mind, but meanwhile it has been successfully used to build traditional Web applications: an HTTP server that serves HTML pages or replies with JSON messages and uses a database to store the data. Even though Web frameworks in other platforms and languages have preferred to stick with traditional open-source relational databases like MySQL or PostgreSQL, most of the existing Node Web frameworks (like Express, Hapi and others) don’t impose any particular database, or even any type of database at all. This bring-your-own-database approach has been fed in part by the explosion in the variety of database servers now available, but also by the ease with which the Node module system and NPM allow you to instal and use third-party libraries.

In this short book we will analyse some of the existing solutions for interacting with some types of databases.

3. An embedded database using LevelDB

LevelDB is a fast key-value store written by Google engineers that has the following characteristics:

- Keys and values are arbitrary byte arrays.
- It stores data sorted by key.
- It provides basic `put(key, value)`, `get(key)` and `delete(key)` primitives.
- It allows multiple changes to be performed in one atomic batch.
- It allows you to read a transient consistent snapshot of the data.

LevelDB can be used from within Node.js by installing the `level` package. When doing so, it compiles and instals LevelDB as a library, which you can then access through JavaScript.

LevelDB is thread-safe, but is not suited to being accessed from multiple processes. This means that you should only have a LevelDB database open from a single Node process. If you have multiple Node processes, a LevelDB database cannot be shared between them.

Also, LevelDB just exposes a simple key-value store API, on top of which a multitude of extension plugins exist, implementing functions as diverse as job queues, map-reduce views, partitioning, replication, locking, network exposure, and others. In this chapter we will explore some of these.

3.1 Installing LevelDB

To instal LevelDB you should add the `level` package to your `package.json` application manifest using NPM from the command line:

```
$ npm install level --save
```

Here we assume you already have a `package.json` file at the root directory of your application.

This last commands adds the `level` dependency to your manifest and installs it. Now you can require it and use it inside your application:

db.js:

```
var level = require('level');
var path = require('path');

var dbPath = process.env.DB_PATH || path.join(__dirname, 'mydb');
var db = level(dbPath);

module.exports = db;
```

Here we created a singleton module that exports a LevelDB database. We start by requiring the `level` module and use it to instatiate the database, providing it with a path. This path is the path of the directory where LevelDB will store its files. This directory should be fully dedicated to LevelDB, and may not exist at the beginning. In our case the path for the database is contained in the `DB_PATH` envionment variable or, if not defined, defaults to the `mydb` directory inside the directory of the current source file.

3.2 Using LevelDB

Now, from another module we can get a reference to the database by requiring our singleton `db.js` module:

app.js:

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

After this we can use it immediately to put some values:

```
db.put('key 1', 'value 1');
db.put('key 2', 'value 2');
```

You can start issuing commands and queries into a LevelDB database object that has recently been created, without having to wait for it to actually open the database. While the database isn't ready for action, the queries and commands are queued.

You can also use the `db` object to get some values:

```
db.get('key 1', function(err, value) {
  if (err) {
    return handleError(err);
  }
  console.log('value:', value);
});
```

or even delete some entries:

```
db.del('key 1');
```

Any of these operations take an optional callback as the last argument. When an error happens while performing the operation, this callback gets the error object as the first argument:

```
db.put('key 1', 'value 1', function(err) {
  if (err) {
    console.error('error putting key 1:', err);
  }
});
```

3.3 Encodings

LevelDB supports any binary value as a key or a value: you can pass in buffer objects, but in our previous cases we're using strings which, by default, LevelDB assumes are UTF-8 encoded. You can, if you wish to, use other key or value encodings by setting the `keyEncoding` or the `valueEncoding` in the constructor `options` object:

```
var options = {
  keyEncoding: 'hex',
  valueEncoding: 'base64'
};

var db = level(dbPath, options);
```

Here we're telling LevelDB to assume that keys are hex-encoded and that values are base64-encoded. Alternatively, you can use any encoding that the `Buffer.toString()` method allows: namely "hex", "utf8", "ascii", "binary", "base64", "ucs2", or "utf16le".

```
db.get('6b6579', function(err, value) {
  if (err) {
    return handleError(err);
  }
  console.log('base64-encoded value:', value);
});
```

3.4 Using JSON for encoding values

You can specify `json` as a special value-encoding type, which allows you to use arbitrary JavaScript objects as values:

use_json.js:

```
var level = require('level');
var path = require('path');
var assert = require('assert');

var dbPath = process.env.DB_PATH || path.join(__dirname, 'mydb');

var options = {
  keyEncoding: 'binary',
  valueEncoding: 'json'
};

var db = level(dbPath, options);

db.put(new Buffer([1, 2, 3]), { some: 'json' }, function(err) {
  if (err) {
    return console.error(err);
  }

  db.get(new Buffer([1, 2, 3]), function(err, value) {
    if (err) {
      return console.error(err);
    }

    assert.deepEqual(value, { some: 'json' });
    console.log(value);
  });
});
```

One common use of the `json` encoding is storing and retrieving complex JavaScript objects without the need to do the encoding and decoding yourself.

3.5 Batch operations

A powerful feature of LeveLDB is that it allows you to group operations in a batch to be executed automatically. Here is an example of a batch operation:

```
var batch = db.batch();
batch.put('key 1', 'value 1');
batch.put('key 2', 'value 2')
batch.del('key 3')
batch.write(function(err) {
  if (err) {
    console.error('batch operation failed:', err);
  }
});
```

Here you can see that we created a batch by calling `db.batch()`, and that we queued `put` and `del` operations. At the end we try to execute the entire batch by calling the `batch.write`.

Any batch queueing operation always returns the batch itself, permitting us to add operations using a chained style:

```
db.batch()
  .put('key 1', 'value 1')
  .put('key 2', 'value 2')
  .del('key 3')
  .write(function(err) {
    if (err) {
      console.error('batch operation failed:', err);
    }
  });
```

In this form you need to terminate the batch of commmands with `batch.write`, providing it with a completion callback. If you get an error on the callback you provided to `batch.write()`, you can safely assume that all the operations contained in the batch have failed. If you don't get any error, you can safely assume that all the operations have succeeded and have been committed.

Instead of using the chaining API, you can instead use the array version of `.batch()`, which can be useful for mapping objects into operations:

```
var users = // ...

var operations = users.map(function(user) {
  return { type: 'put', key: user.id, value: user };
});

db.batch(operations, function(err) {
  if (err) {
    console.error('error in user batch:', err);
  }
});
```

Here you are passing an array containing all the operations of the batch. Each operation is described by an object containing a `type` attribute that specifies the operation type (either `put` or `del`). Any operation must define a `key` attribute, and an operation with type `put` must also provide a `value` attribute.

3.6 Using a readable stream

We can create a readable stream from a LevelDB database like this:

```
var stream = db.createReadStream();
```

This streams emits one data object per record in the database:

```
stream.on('data', function(data) {
  console.log('%s = %j', data.key, data.value);
});
```

The emitted data is an object that contains a `key` attribute and a `value` attribute. Since LevelDB stores data sorted by key, we can expect this stream to emit the keys in a sorted order.

Besides emitting data, this readable stream emits some other interesting events:

```
stream.once('end', function() {
  console.log('no more data');
});

stream.once('close', function() {
  console.log('stream closed');
});

stream.once('error', function(err) {
  console.error('stream emitted error:', err);
});
```

The `end` event is emitted once the stream has emitted all the data, and the `close` event is immediately after. Also, the error event may be emitted if an error happens while reading the data.

3.6.1 Using ranges

In the previous example we were streaming the entire contents of the database. You can, alternatively, limit the emitted data to a range of keys by using the `gt` (greater-than), `gte` (greater-than-or-equal-to), `lt` (less-than), and `lte` (less-than-or-equal-to) options:

ranges.js:

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

var stream = db.createReadStream({
  gte: 'a',
  lte: 'z'
});

stream.on('data', function(record) {
  assert(record.key >= 'a' && record.key <= 'z');
});
```

Here we're creating a read stream that will emit a `data` event for every record that contains a key that is both greater than or equal to the string "a", and less than or equal to the string "z". Because of the inclusiveness we're using, if our database contains a record with a key "a" or "z", that record will be emitted. On the other hand, if it includes a record with a key named "z1", it will not be emitted (because "z1" is greater than "z").

3.6.2 Limiting the number of results

Instead of streaming the entire contents of your database or your range, you can limit the maximum number of emitted records using the `limit` option:

limit.js:

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

var stream = db.createReadStream({
  limit: 10 // only interested in the first 10 records
});

var count = 0;
stream.on('data', function(record) {
  assert(++ count <= 10);
});
```

3.6.3 A consistent snapshot

One of the many advantages of using a readable stream is that you get a consistent snapshot read. This means that, if any write or batch write happens after the stream begins, these writes will not be reflected in the stream. Let's see this in action:

consistent.js:

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

var batch = [
  {type: 'put', key: 'a', value: 'A'},
  {type: 'put', key: 'b', value: 'B'},
  {type: 'put', key: 'c', value: 'C'},
  {type: 'put', key: 'd', value: 'D'}
];

var index = 0;

db.batch(batch, function(err) {
  var stream;

  if (err) {
    console.error(err);
  }
  else {
    stream = db.createReadStream({gte: 'a', lte: 'd'});
    stream.on('data', onData);

    db.batch(batch.map(function(op) {
      return {type: op.type, key: op.key, value: 'other value'};
    }), function(err) {
      if (err) {
        console.error(err);
      }
      else {
        console.log('saved batch replacing with other value, resuming');
      }
    });
  }

});
```

```
function onData(record) {
  console.log('read: %j', record);
  assert.equal(record.value, batch[index ++].value);
}
```

If you run this script you will get the following result:

```
$ node consistent
read: {"key":"a","value":"A"}
saved batch replacing with other value, resuming
read: {"key":"b","value":"B"}
read: {"key":"c","value":"C"}
read: {"key":"d","value":"D"}
```

Some time between reading the values, the batch write replacing all the values with the string `other value` succeeded, but it is never reflected on the readable stream that was already open.

3.6.4 Using ranges to partition the data

LevelDB is a key-value store: an arbitrary value is stored under an arbitrary key, but this doesn't need to limit the type of values you store. Since it stores all the keys in sorted order, you can partition the key namespace into an unlimited set of separate namespaces.

For instance, if you're programming some kind of a social network and you have a database where you store user records, you can also use it to store all the user relationships.

bookface_populate.js

```
var path = require('path');
var level = require('level');

var dbPath = process.env.DB_PATH || path.join(__dirname, 'bookface');
var db = level(dbPath, {
  valueEncoding: 'json'
});

var batch = [];

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

users.forEach(function(user) {
  batch.push({
    type: 'put',
    key: user.email,
    value: {
      email: user.email,
      createdAt: new Date
    }
  });

  user.friends.forEach(function(friend) {
    batch.push({
      type: 'put',
      key: user.email + '!friendships!' + friend,
      value: {
        source: user.email,
```

```
          target: friend,
          createdAt: new Date
        }
      })
  });
});

db.batch(batch, function(err) {
  if (err) {
    throw err;
  }

  console.log('populated successfully');

});
```

In this last file there are some peripheral things going on. First we're populating a batch operation that contains all the initial users and their friendships, storing it in the `batch` variable. Next we're issuing the `db.batch` command, inserting all the users and their friendships into our database in one go. For this to work, you will need to define the `users.js` module:

users.js:

```
module.exports = [
  {
    email: 'person1@example.com',
    friends: [
      'person2@example.com',
      'person3@example.com'
    ]
  },
  {
    email: 'person2@example.com',
    friends: [
      'person1@example.com',
      'person3@example.com'
    ]
  },
  {
    email: 'person3@example.com',
    friends: [
      'person1@example.com'
    ]
  }
];
```

Now we can populate our `bookface` database:

```
$ node bookface_populate
populated successfully
```

Next, we can devise a command-line query script:

bookface.js:

```
var path = require('path');
var level = require('level');

var dbPath = process.env.DB_PATH || path.join(__dirname, 'bookface');
```

```
var db = level(dbPath, {
  valueEncoding: 'json'
});

function user(email, cb) {
  db.get(email, cb);
}

function friendships(email) {
  var key = email + '!' + 'friendships!';
  return db.createValueStream({
    gte: key,
    lte: key + '\xff'
  });
}

var email = process.argv[2];

user(email, function(err, user) {
  console.log('got user:', user);
});

var friends = friendships(email).on('data', function(friend) {
  console.log('friend:', friend.target);
});

friends.once('end', function() {
  console.log('no more friends');
});
```

We can now use this script to query users and their relationships from our database:

```
$ node bookface person1@example.com
got user: { email: 'person1@example.com',
  createdAt: '2015-01-06T15:43:01.196Z' }
friend: person2@example.com
friend: person3@example.com
no more friends
```

Here we're storing each friendship inside the `<EMAIL>!friendship!` namespace. Generically, we're using the `!` character to separate elements in our keys. The trick in retrieving the data is in the range query, as in this example:

```
function friendships(email) {
  var key = email + '!' + 'friendships!';
  return db.createValueStream({
    gte: key,
    lte: key + '\xff'
  });
}
```

Perhaps counter-intuitive, but particulary important is the higher-end limiting (`key + '\xff'`), so that we don't get more records than we want.

3.7 Using level-sublevel

Instead of creating the keys by hand, you can use the `level-sublevel` NPM module to partition the key space at will. Let's then instal it:

```
$ npm install level-sublevel --save
```

We'll also be needing to generate random unique IDs, so let's take the chance to instal the `cuid` NPM module:

```
$ npm install cuid --save
```

We can now use level and level-sublevel to create the two sub-databases – one for holding user records and another for holding user messages:

sublevels.js:

```
var level = require('level');
var path = require('path');
var sublevel = require('level-sublevel');

var dbPath = process.env.DB_PATH || path.join(__dirname, 'sublevels');
var db = sublevel(level(dbPath, {
  valueEncoding: 'json'
}));

exports.base = db;
exports.users = db.sublevel('users');
exports.messages = db.sublevel('messages');
```

Here we're wrapping the database returned by `level` with `sublevel`, storing it in a temporary variable named `db`. We then create the two sub-databases by calling `db.sublevel('<DATABASE NAME>')`, one for users and another for messages.

Now we can use this `sublevels` module to populate the database:

sublevels_populate.js:

```
var cuid = require('cuid');
var db = require('./sublevels');

var user = {
  name: 'John',
  email: 'user1@example.com'
};

db.users.put(user.email, user, function() {
  for(var i = 1 ; i <= 20; i ++) {
    var userMessages = db.messages.sublevel(user.email);
    userMessages.put(cuid(), {
      from: 'user' + i + '@example.com',
      to: 'user1@example.com',
      subject: 'Hey!',
      body: 'hey there, how you doing?'
    });
  }
});
```

Here we're creating a user record, and then creating 20 messages addressed to them. You can see how we can create a sublevel inside another sublevel: we're storing the user messages in a sub-database named after the user email inside the `messages` sub-database.

You may also have noticed that each message has a unique ID given to us by the `cuid` module.

Let's call our script:

```
$ node sublevels_populate
```

Now we can create a script to query our database:

sublevels_query.js:

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

var email = process.argv[2];

db.users.get(email, function(err, user) {
  if (err) {
    throw err;
  }
  console.log('User: %j', user);

  var userMessages = db.messages.sublevel(email);

  userMessages.createValueStream().on('data', function(message) {
    console.log('Message: %j', message);
  })
  .once('end', function() {
    console.log('no more messages');
  });
});
```

We can now test this script, using it to query our user:

```
$ node sublevels_query.js user1@example.com
User: {"name":"John","email":"user1@example.com"}
Message: {"from":"user1@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user2@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user3@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user4@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user5@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user6@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user7@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user8@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
Message: {"from":"user9@example.com","to":"user1@example.com","subject":"Hey!","\
body":"hey there, how you doing?"}
...
no more messages
```

3.7.1 Batch in different sublevels

Besides being able to use one LevelDB database as a set of different databases, we can also use `level-sublevel` to atomically perform a batch of updates into two or more distinct sublevels, all in an atomic way. For instance, when a message is created, you may want to atomically insert it into the sender outbox and the receiver inbox:

sublevels_insert_message.js:

```
var db = require('./sublevels');
var cuid = require('cuid');

exports.insert = insertMessage;

function insertMessage(to, from, subject, body, cb) {
  var id = cuid();
  var message = {
    to: to,
    from: from,
    subject: subject,
    body: body
  };

  var batch = [
    {
      type: 'put',
      key: id,
      value: message,
      prefix: db.messages.sublevel(from).sublevel('out')
    },
    {
      type: 'put',
      key: id,
      value: message,
      prefix: db.messages.sublevel(to).sublevel('in')
    }
  ];

  db.base.batch(batch, cb);
}
```

Here we can see that we're using `db.batch` to atomically insert two records, each into different sublevels. We specify which sublevel the operation is going to affect in the `prefix` property of each batch entry.

You will also notice that we're inserting the batch into the base database, and not into any sublevel. This is because the sublevel is already specified inside each batch entry.

3.8 Hooks

The `level-sublevel` is really the Swiss army knife for LevelDB, and it doesn't end here. We can use it to automatically make changes before a change occurs by using `db.pre`. We can use this to store a log of all the changes performed in a database (or a specific sub-level) in a separate sub-level. For instance, we could hook into our `users` sub-level and record every change made to it:

sublevels_user_hook.js:

```
var cuid = require('cuid');
var db = require('./sublevels');
var userChanges = db.base.sublevel('userchanges');
```

```
db.users.pre(function(change, add) {
  add({
    type: 'put',
    key: cuid(),
    value: {
      when: new Date(),
      change: change
    },
    prefix: userChanges.sublevel(change.key)
  });
});
```

We can now activate this hook by including this module in our sublevels.js file:

bottom of sublevels.js:

```
exports.base = db;
exports.users = db.sublevel('users');
exports.messages = db.sublevel('messages');

require('./sublevels_user_hook');
```

We can now make some number of user changes using our `sublevel_populate` script:

```
$ node sublevels_populate
$ node sublevels_populate
$ node sublevels_populate
```

We can now create a script to query our changes to a given user:

sublevels_user_log.js:

```
var db = require('./sublevels').base.sublevel('userchanges');

var email = process.argv[2];

var userChanges = db.sublevel(email);

userChanges.createValueStream().on('data', function(message) {
  console.log('User Change: %j', message);
})
.once('end', function() {
  console.log('no more changes');
});
```

And use it:

```
$ node sublevels_user_log.js user1@example.com
User Change: {"when":"2015-01-08T12:19:43.154Z","change":{"key":"user1@example.c\
om","value":{"name":"John","email":"user1@example.com"},"prefix":["users"],"type\
":"put"}}
User Change: {"when":"2015-01-08T12:19:43.946Z","change":{"key":"user1@example.c\
om","value":{"name":"John","email":"user1@example.com"},"prefix":["users"],"type\
":"put"}}
User Change: {"when":"2015-01-08T12:19:44.555Z","change":{"key":"user1@example.c\
om","value":{"name":"John","email":"user1@example.com"},"prefix":["users"],"type\
":"put"}}
no more changes
```

4. Redis

Redis is an open-source database server that has been gaining popularity recently. It's an in-memory key-value store. It keeps all the data in-memory for fast access, but also keeps the data in-disk if you tell it to. It's a basic key-value store where the keys and values are strings, but it also contains more interesting constructs like integers, lists, sets, sorted sets and dictionaries, and also contains some advanced features like pub-sub, blocking pop, key monitoring, and transactions.

Redis is the Swiss knife of in-memory databases: you can use it to implement many different use-cases, ranging from a data cache or a work queue to a statistics log.

4.1 Redis primitives

In Node, to access Redis you will need to instal a client library. The most used and battle-tested one is `redis`:

```
$ npm install redis --save
```

If you haven't yet done it, to instal Redis itself you should follow the instructions on the official website at http://redis.io/download.

redis.js:

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

module.exports = redis.createClient();
```

This local `redis` module uses the installed official `redis` NPM module to create a client object, which is what this module exports.

Optionally you can pass in some client options if you're hosting Redis in a different port, or in another network host:

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

var port = process.env.REDIS_PORT || 6379;
var host = process.env.REDIS_HOST || '127.0.0.1';

module.exports = redis.createClient(port, host);
```

To use this module from any other file you just have to require it like this:

client_example.js:

```
var redis = require('./redis');
var assert = require('assert');

redis.set('key', 'value', function(err) {
  if (err) {
```

```
    throw err
  }

  redis.get('key', function(err, value) {
    if (err) {
      throw err
    }

    assert.equal(value, 'value');

    console.log('it works!');

    redis.quit();
  });
});
```

Here in this last example we're setting a key and then getting the value for that key, asserting that it is indeed what we inserted.

Now let's look at some of Redis' useful primitives, and how to use them in Node.

4.1.1 Strings

In Redis all keys are strings, and values are typically strings. (We'll see some examples using numbers and objects later.) As we've seen, to set a key to a string you use `put`, providing the key and the value strings:

```
redis.put('key', 'value');
```

For every Redis command you can pass in a callback to be invoked when the command completes or errors:

```
redis.put('key', 'value', function(err) {
  if (err) {
    console.error('error putting key:', err)
  }
  else {
    console.log('key saved with success');
  }
});
```

You can also get the string value of any key using the `get` command:

```
redis.get('key', function(err, value) {
  if (err) {
    console.error('error getting key:', err);
  }
  else {
    console.log('key has the value %s', value);
  }
});
```

4.1.2 Key expiration

Besides the basic key-value operations, Redis has a load of useful functionalities. One of them is key expiration. You can define an expiration time after which the entry will be removed from Redis. Let's see this in action:

expiration.js:

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

redis.set('some key', 'some value');
redis.expire('some key', 2);

setInterval(function() {
   redis.get('some key', function(err, value) {
     if (err) {
       throw err;
     }
     if (value) {
       console.log('value:', value);
     }
     else {
       console.log('value is gone');
       process.exit();
     }
   });
}, 1e3);
```

Here we're setting a key and then expiring it after two seconds have elapsed. We then poll Redis for the value on that key until Redis removes that record, at which time we terminate the current script.

You may have noticed that in the previous example we are sometimes calling Redis without providing a callback. Is this safe? This is how it works: when issuing several commands on the same client, the client is only executing one command at a time; the pending commands are waiting on a queue. This happens because the Redis protocol only allows one running command per connection. If an error occurs and you didn't provide a callback, the error will be emitted on the client. When this happens, you lose all the context of the error. This is why you should always provide callbacks in the commands.

You can test this by doing:

```
$ node expiration
value: some value
value is gone
```

If you need to set the expiration time when you set the key, you can use the `SETEX` command instead:

expiration_setex.js:

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

redis.setex('some key', 2, 'some value');

setInterval(function() {
   redis.get('some key', function(err, value) {
     if (err) {
       throw err;
     }
     if (value) {
       console.log('value:', value);
     }
     else {
       console.log('value is gone');
       process.exit();
```

```
    }
  });
}, 1e3);
```

4.1.3 Transactions

In the previous example we saw a command that combines two commands into one (`SET` and `EXPIRE`). You can also choose to combine whatever commands you wish into one atomic transaction by using the `MULTI` command. For instance, if you want to atomically set two keys and set the expiration of another, you can compose it like this:

multi.js:

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

redis.multi().
  set('key A', 'some value A').
  set('key B', 'some value B').
  expire('some other key', 2).
  exec(function(err) {
    if (err) {
      throw err;
    }
    console.log('terminated');
    redis.quit();
  });
```

Here we're constructing a MULTI command using `redis.multi()` We're then adding commands to this transaction using the Redis API, but on the created multi. We then execute the transaction by calling `multi.exec()`, providing it with a callback.

If you get an error, then none of the commands were effective. If you don't get an error, this means that each command succeeded.

All these commands are documented on the official Redis website: http://redis.io/commands. With some few exceptions, the `redis` client follows the exact same argument order and type as documented.

4.1.4 Command results in Multi

Besides executing multiple commands, you can also execute multiple queries, or mix commands and queries, getting the results at the end. Let's see this in action:

multi_read.js:

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

redis.multi().
  set('key A', 'some value for A').
  get('key A').
  set('key A', 'some *OTHER* value for A').
  get('key A').
  exec(function(err, results) {
    if (err) {
      throw err;
    }
    console.log('terminated. results: %j', results);
```

```
    redis.quit();
  });
```

If you execute this, you get the following output:

```
$ node multi_read.js
terminated. results: ["OK","some value for A","OK","some *OTHER* value for A"]
```

Here you can see that the result passed to our callback is an array containing a position for the result of each operation in our transaction. The `SET` operations (positions 0 and 2) resulted in an `OK` string (this is how Redis indicates success); and the `GET` operations (positions 1 and 3) resulted in the key values at the time of execution.

4.1.5 Optimistic locking using WATCH

The `MULTI` command can be powerful, but you still can't issue atomic compare-and-set operations. For instance, let's say that you want to make an operation where you read an integer value from key `A` and then you add 1 to it and store it back:

```
redis.get('A', function(err, value) {
  if (err) {
    throw err;
  }
  var newValue = Number(value) + 1;
  redis.set('A', newValue);
});
```

This approach has an obvious problem though: it doesn't allow safe concurrent clients. If more than one client is performing this operation in parallel, more than one may read the same value, increment it, and then save the same result. This means that, instead of atomically incrementing the value, we would lose increments. This can be solved by using a combination of `WATCH` and `MULTI`:

```
var Redis = require('redis');

function increment(key, cb) {
  var replied = false;
  var newValue;

  var redis = Redis.createClient();
  redis.once('error', done);
  redis.watch(key);

  redis.get(key, function(err, value) {
    if (err) {
      return done(err);
    }
    newValue = Number(value) + 1;
    redis.multi().
      set(key, newValue).
      exec(done);
  });

  function done(err, result) {
    redis.quit();

    if (! replied) {
      if (!err && !result) {
```

```
          err = new Error('Conflict detected');
        }

        replied = true;
        cb(err, newValue);
      }
    }
}

increment('A', function(err, newValue) {
  if (err) {
    throw err;
  }
  console.log('successfully set new value to %j', newValue);
});
```

Here we're defining a generic `increment` function that takes a key and a callback. After defining this function we use it to increment the key `A`. When this is done we terminate the Redis client connection.

The `increment` function starts by watching the key, and then starting a `MULTI` command. We then calculate the new value for A and append the `SET` command to the multi. We then execute this one-command multi, passing our `done` function as a callback. This function detects conflicts. The `MULTI` command returns a `null` value (instead of the traditional `OK`) when a conflict is detected. A conflict happens when any of the watched keys (in our case, only one) is written by another connection. In this case, we detect the conflict and raise an appropriate error with a message stating that a conflict was detected.

There are several minute and perhaps non-obvious aspects to this script, though. First, we're using one connection per transaction. This is because Redis keeps one watch list per connection, and a `MULTI` execution is only aborted if a write on a watched key is performed on *a different* client connection. Also, after the multi has been executed, the watch list is discarded. Basically, it's not safe to share a connection if you're relying on the behaviour of a watch list.

Issuing more that one `WATCH` command adds a key to the connection watch list. You can clear the watch list using the `UNWATCH` command.

Also notice that, since we're creating one Redis connection per increment, we're being especially careful in closing that connection. This means that, in the case of an error, we always call the `done` function, which is responsible for closing the connection to Redis. We have to cover all error conditions – it's very easy to leak short-lived connections.

One connection per transaction means that extra client-side and server-side resources are needed. Be careful with dimensioning your systems when using this feature, as it's easy to burden the Node processes or the Redis server during peak traffic!

We can test the conflict detection by concurrently issuing more than one increment at the end of the last file:

end of increment_watch.js:

```
for(var i = 0 ; i < 10 ; i ++) {
  increment('A', function(err, newValue) {
    if (err) {
      throw err;
    }
    console.log('successfully set new value to %j', newValue);
  });
}
```

Let's run this script:

```
$ node increment_watch
successfully set new value to 14

/Users/pedroteixeira/projects/nodejs-patterns-book/code/08/redis/node_modules/re\
dis/index.js:602
                throw err;
                      ^
Error: Conflict detected
    at done (/Users/pedroteixeira/projects/nodejs-patterns-book/code/08/redis/in\
crement_watch.js:27:15)
```

But now that we can properly detect conflicts, we can handle them and retry performing the transaction:

increment_watch_retry.js:

```
var Redis = require('redis');

function _increment(key, cb) {
  var replied = false;
  var newValue;

  var redis = Redis.createClient();
  redis.once('error', done);
  redis.watch(key);

  redis.get(key, function(err, value) {
    if (err) {
      return done(err);
    }
    newValue = Number(value) + 1;
    redis.multi().
      set(key, newValue).
      exec(done);
  });

  function done(err, result) {
    redis.quit();

    if (! replied) {
      if (!err && !result) {
        err = new Error('Conflict detected');
      }

      replied = true;
      cb(err, newValue);
    }
```

```
  }
}

function increment(key, cb) {
  _increment(key, callback);

  function callback(err, result) {
    if (err && err.message == 'Conflict detected') {
      _increment(key, callback);
    }
    else {
      cb(err, result);
    }
  }
}


for(var i = 0 ; i < 10 ; i ++) {
  increment('A', function(err, newValue) {
    if (err) {
      throw err;
    }
    console.log('successfully set new value to %j', newValue);
  });
}
```

Here we renamed our `increment` function to `_increment` and created a new `increment` function that handles the special case where a conflict is detected. Since we are guaranteed that Redis will not commit if there is a conflict, we can safely try calling the `_increment` function again.

We can now test this new version and verify that all the transactions eventually succeeded, even though they were emitted concurrently:

```
$ node increment_watch_retry.js
successfully set new value to 15
successfully set new value to 16
successfully set new value to 17
successfully set new value to 18
successfully set new value to 19
successfully set new value to 20
successfully set new value to 21
successfully set new value to 22
successfully set new value to 23
successfully set new value to 24
```

Here we used a simple increment, but you can easily see that we can use optimistic locking to create any type of custom transactions as long as we're using different Redis connections for each transaction, and that we're watching the necessary keys.

4.1.6 Transactions using Lua scripts

Another way of performing arbitrarily complex operations in an atomic way in Redis is by using scripts written in Lua. Redis provides a way for us to inject and run Lua scripts inside it. When executing a Lua script in Redis, we are guaranteed that no other command or script is executing concurrently, which is exactly what we want.

Lua is a light scripting language that is somewhat similar to JavaScript (even though arrays start at index 1, not 0…).

If you don't know Lua, there are several resources out there if you want to learn it — but it's clearly out of the scope of this book. Nonetheless, we're going to present here some examples that you may find useful to base your own scripts on.

First we're going to port our `increment` transaction into a Redis Lua script:

lua_scripts/increment.lua:

```
local key = KEYS[1]

local value = redis.call('get', key) or 0
local newValue = value + 1
redis.call('set', key, newValue)

return newValue
```

Here you can see that this simple script starts by getting the name of the key from the `KEYS` variable. This is a special implicit variable that Redis passes to the script, which comes from the client invocation. After that we get the value stored in that key by calling the Redis engine. The Redis operations are available to be called using `redis.call`; the first argument is the operation name, and the following arguments are the arguments to the operation itself.

After getting the current value, we increment it, store it, and then return it as the result of the operation.

Here is the Node part that implements the `increment` function, and that delegates to the Lua script:

increment_lua.js:

```
var fs = require('fs');
var path = require('path');
var redis = require('./redis');

var script = fs.readFileSync(
  path.join(__dirname, 'lua_scripts', 'increment.lua'),
  {encoding: 'utf8'});

function increment(key, cb) {
  redis.eval(script, 1, key, cb);
}

for(var i = 0 ; i < 10 ; i ++) {
  increment('some key', function(err, newValue) {
    if (err) {
      throw err;
    }
    console.log('successfully set new value to %j', newValue);
    redis.quit();
  });
}
```

The first thing we do is to load the Lua script into memory. We do this by using `fs.readFileSync`.

Somewhere you may have heard that it's wrong to use the Node synchronous functions – you should always use the asynchronous version of the functions so that you don't block the Node's event loop. It's OK to use synchronous functions during module prepartion time – they exist here because of that. As a rule of thumb, you only have to avoid calling synchronous functions from inside a callback or an event listener.

Now that we have our Lua script in memory, we can implement the `increment` function, which loads the script into Redis and calls it by using the `EVAL` command. The first argument of `redis.eval` is the script code itself, and after that comes the number of keys we're going to pass. In our case, we'll only be passing one key argument. Finally, there's a callback for us to know when the operation failed or succeeded, and in the last case, what the result was.

Besides keys, you can also pass arbitrary arguments, which the Lua script can access using the `ARGS` implicit array variable.

We can now test our script:

```
$ node increment_lua.js
successfully set new value to 1
successfully set new value to 2
successfully set new value to 3
successfully set new value to 4
successfully set new value to 5
successfully set new value to 6
successfully set new value to 7
successfully set new value to 8
successfully set new value to 9
successfully set new value to 10
```

4.1.6.1 Caching Lua scripts

There's one problem with our last implementation: we are always passing in the Lua script before invoking. This bears the overhead of transmitting the script into Redis, Redis loading, parsing and running the script. We can optimise this by using the Redis `EVALSHA` command, which allows us to invoke a script given its SHA1 digest. Let's use it to avoid loading the same script all the time:

lua_scripts/index.js:

```
var fs = require('fs');
var path = require('path');
var crypto = require('crypto');

var NO_SCRIPT_REGEXP = /NOSCRIPT/;

var scriptNames = ['increment'];

var scripts = {};
```

```
scriptNames.forEach(function(scriptName) {
  var body = fs.readFileSync(
    path.join(__dirname, scriptName + '.lua'),
    {encoding: 'utf8'})

  scripts[scriptName] = {
    body: body,
    digest: crypto.createHash('sha1').update(body).digest('hex')
  };
});

exports.execute = execute;

function execute(redis, scriptName, keyCount) {
  var args = Array.prototype.slice.call(arguments);
  var cb = args[args.length - 1];
  var redis = args.shift();
  var scriptName = args.shift();
  var script = scripts[scriptName];
  if (!script) {
    cb(new Error('script is not defined: ' + scriptName));
  }
  else {
    var digest = script.digest;
    args.unshift(digest);
    args[args.length - 1] = callback;
    redis.evalsha.apply(redis, args);
  }

  function callback(err) {
    if (err && err.message.match(NO_SCRIPT_REGEXP)) {
      args[0] = script.body;
      redis.eval.apply(redis, args);
    }
    else {
      cb.apply(null, arguments);
    }
  };
}
```

This code is a generic module that manages the execution of scripts for you. The variable named `scriptNames` is an array that contains all the names of the available scripts. In our simple case we only have one script named `increment`, but we could have more. When this module loads, it loads the script bodies and calculates the SHA1 digest of each one.

Also, this module exports an `execute` function that takes a Redis connection, a script name and a set of arbitrary script execution arguments, finalised with a callback. This function starts by trying to execute the script using `redis.evalsha`, passing in the script SHA1 digest. If Redis cannot find the script, the execution yields a specific error message that contains the “NOSCRIPT” string. When we get such an error, Redis is telling us that it doesn’t yet contain the script: we then fall back to using `redis.eval`, passing in the script body instead of the script digest.

Now we can simply use this module from a client script like this:

increment_lua_sha.js:

```
var redis = require('./redis');
var luaScripts = require('./lua_scripts');

function increment(key, cb) {
  luaScripts.execute(redis, 'increment', 1, key, cb);
}

for(var i = 0 ; i < 10 ; i ++) {
  increment('some key', function(err, newValue) {
    if (err) {
      throw err;
    }
    console.log('successfully set new value to %j', newValue);
    redis.quit();
  });
}
```

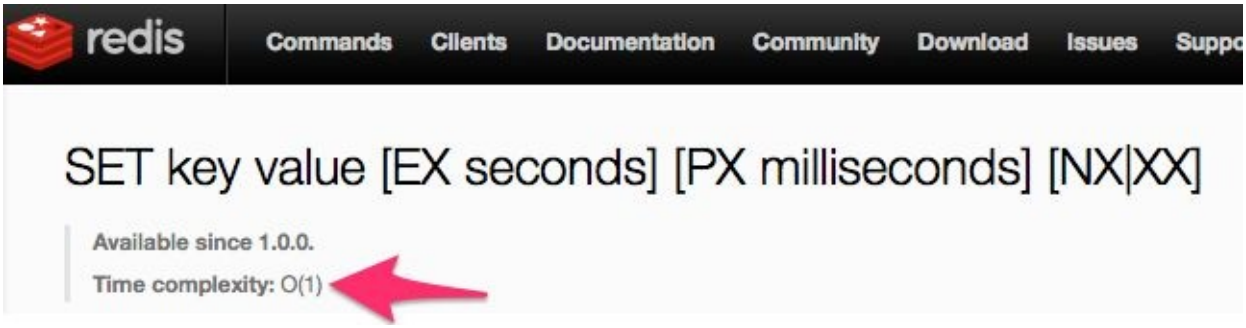
When executing this script, again you should see the same output as before – but now knowing in your heart that it's doing its best to reuse the cached script.

```
$ node increment_lua_sha.js
successfully set new value to 31
successfully set new value to 32
successfully set new value to 33
successfully set new value to 34
successfully set new value to 35
successfully set new value to 36
successfully set new value to 37
successfully set new value to 38
successfully set new value to 39
successfully set new value to 40
```

4.1.6.2 Performance

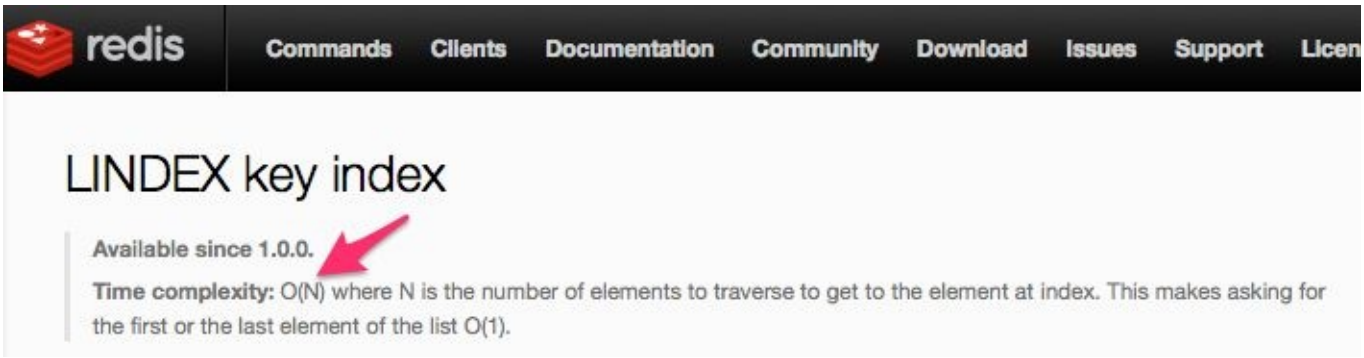
Since we know that all Redis queries and commands are performed in memory, and also that Lua scripts are executed quickly, if we're somewhat careful with the number and types of operations we perform we can also guarantee that each transaction is performed quickly enough for us to be able to accomodate a given workload.

Each Redis command has a time complexity that's given to us in *O()* notation. For instance, if a given operation is *O(1)*, we know that it will always take the same fixed amont of time.



Redis SET command complexity

For instance, the Redis `LINDEX` command, which gets the Nth element of a list, has a complexity of *O(N)*, where N is the number of elements the list has. If you're using this command, you must somehow ensure that the number of elements in this list in not unbounded.



Redis LINDEX command complexity

4.1.7 Integers

Earlier we saw how to perform atomic operations in Redis, and we showed the example of incrementing an integer value of a record. This example was only presented for the sake of showing a simple example, since it happens that Redis already has increment and decrement commands.

incr.js:

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

for(var i = 0 ; i < 10 ; i ++) {
  redis.incr('some key', done);
}

function done(err, result) {
  if (err) {
    throw err;
  }
  console.log('new value:', result);
  redis.quit();
}
```

If you execute this file you will get the same type of result as in the previous custom version:

```
$ node incr
new value: 1
new value: 2
new value: 3
new value: 4
new value: 5
new value: 6
new value: 7
new value: 8
new value: 9
new value: 10
```

Besides incrementing by one, Redis also allows incrementing by a specific integer value using the `INCRBY` command:

incrby.js:

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

for(var i = 0 ; i < 10 ; i ++) {
  redis.incrby('some other key', 2, done);
```

```
}

function done(err, result) {
  if (err) {
    throw err;
  }
  console.log('new value:', result);
  redis.quit();
}
```

In this case we're incrementing the record by the value of 2 ten times:

```
$ node incrby.js
new value: 2
new value: 4
new value: 6
new value: 8
new value: 10
new value: 12
new value: 14
new value: 16
new value: 18
new value: 20
```

Besides incrementing, we can also decrement:

decr.js:

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

for(var i = 0 ; i < 10 ; i ++) {
  redis.decr('some key', done);
}

function done(err, result) {
  if (err) {
    throw err;
  }
  console.log('new value:', result);
  redis.quit();
}
```

This yields the following expected results:

```
$ node decr
new value: 9
new value: 8
new value: 7
new value: 6
new value: 5
new value: 4
new value: 3
new value: 2
new value: 1
new value: 0
```

Redis also has a `DECRBY` command that allows us to decrement by a specific amount:

decrby.js:

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

```
for(var i = 0 ; i < 10 ; i ++) {
  redis.decrby('some other key', 2, done);
}

function done(err, result) {
  if (err) {
    throw err;
  }
  console.log('new value:', result);
  redis.quit();
}
```

And running this last script yields the following results:

```
$ node decrby
new value: 18
new value: 16
new value: 14
new value: 12
new value: 10
new value: 8
new value: 6
new value: 4
new value: 2
new value: 0
```

4.1.8 Using counters

These primitives give great ways to implement basic counters for storing and reading statistics. For instance, you can store per-user statistics on the number of API requests. If you want to throttle API usage per user, you can use Redis to store an API request counter per user, which gets automatically reset at a fixed time interval.

For instance, this is how you could implement a function to increment the API usage counter for a specific user:

api_throttling/incr_api_usage_counter.js:

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

var expirationSecs = 60;

module.exports = incrAPIUsageCounter;

function incrAPIUsageCounter(user, cb) {
  var key = 'api-usage-counter:' + user;
  redis.multi().
    incr(key).
    ttl(key).
    exec(callback);

  function callback(err, results) {
    if (err) {
      cb(err);
    }
    else {
      var newValue = results[0];
      var ttl = results[1];
      if (ttl == -1) {
```

```
          redis.expire(key, expirationSecs, expired);
        }
        else {
          cb(null, newValue);
        }
      }

      function expired(err) {
        if (err) {
          cb(err);
        }
        else {
          cb(null, newValue);
        }
      }
    }
}
```

Here we're using a Redis multi call to make two operations: one to increment the record, and another to get the TTL (time-to-live) of the record. If the record TTL has not been set yet, we set it by using `redis.expire()`. If the TTL is already set, we just terminate the operation.

The TTL command yields a `-1` value for records that exist and that don't yet have a TTL. If the record does not exist, it returns `-2`. We only need to check for the `-1` value because we're querying the TTL in a multi right after the record is updated, guaranteeing that the record exists.

You just have to call this function before any client-authenticated request, and it will increment the user API usage counter.

Now we have to create a function that queries this counter to determine whether or not the user can use the API:

api_throttling/can_user_use_api.js:

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

var maxAPICallsPerUser = 100;

module.exports = canUserUseAPI;

function canUserUseAPI(user, cb) {
  var key = 'api-usage-counter:' + user;
  redis.get(key, function(err, value) {
    if (err) {
      cb(err);
    }
    else {
      var n = Number(value);
      var allowed = n <= maxAPICallsPerUser;
      cb(null, allowed);
    }
  });
}
```

This module gets the value for the user API request counter. If it exceeds a given maximum, we say it's now allowed. Now the application just has to call this function before any authenticated client API request handling, something like this:

throttling_example.js:

```
var canUserUseAPI = require('./api_throttling/can_user_use_api');

app.use(function(req, res, next) {
  canUserUseAPI(req.user, function(err, can) {
    if (err) {
      next(err);
    }
    else if (can) {
      next();
    }
    else {
      res.status(429).send({error: 'Too many requests'});
    }
  });
});
```

In the implementation we are failing if there is an error, but you can instead choose to ignore the errors from Redis and proceed anyway, improving the availability of your service in case Redis is down:

throttling_example_resilient.js:

```
var canUserUseAPI = require('./api_throttling/can_user_use_api');

app.use(function(req, res, next) {
  canUserUseAPI(req.user, function(err, can) {
    if (err || can) {
      next();
    }
    else {
      res.status(429).send({error: 'Too many requests'});
    }
  });
});
```

By using Redis as a store for the API usage counters, we're able to implement a centralised and quick way to check whether the user has exceed the API usage quota. By using Redis' built-in TTLs, we don't need an aditional job to expire records: Redis does that for us.

4.1.9 Dictionaries

Besides simple string values, Redis also allows you to store string dictionaries where the keys and values are strings. The set of Redis commands start with `H` and include `HMSET`, `HMGET` and `HGETALL`, among others. These commands can map well from and into JavaScript shallow objects.

For instance, you can store a user profile with a single set command like this:

user_profile.js:

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

exports.set = setUserProfile;
exports.get = getUserProfile;

function setUserProfile(userId, profile, cb) {
  redis.hmset('profile:' + userId, profile, cb);
}

function getUserProfile(userId, cb) {
  redis.hgetall('profile:' + userId, cb);
}
```

You can use this module we have devised from your app:

```
var UserProfile = require('./user_profile');

var user = 'johndoe';
var profile = {
  name: 'John Doe',
  address: '31 Paper Street, Gotham City',
  zipcode: '987654',
  email: 'john.doe@example.com'
};

UserProfile.set(user, profile, function(err) {
  if (err) {
    throw err;
  }
  console.log('saved user profile');
  UserProfile.get(user, function(err, profile) {
    if (err) {
      throw err;
    }
    console.log('loaded user profile:', profile);
  });
});
```

Of course, instead of using Redis dictionaries you could simply JSON-encode and JSON-decode the user profile object. Using a Redis dictionary here can be useful if you want to get or set individual attributes without having to get and set the entire object.

For instance, to get the user email in this example, you can simply:

```
redis.hget('profile:' + userId, 'email', function(err, email) {
  if (err) {
    handleError(err);
  }
  else {
    console.log('User email:', email);
  }
});
```

Given the persistence and availability properties, I don’t think that Redis is appropriate for the main storage engine for any application. Instead, Redis can be quite useful as a secondary faster storage, and for use as a caching layer.

4.1.10 Redis dictionary counters

Another use for Redis dictionaries is to help with naming the keys by storing several counters under the same key. For instance, if you want to keep track of API access, you can atomically increment several user counters like this:

counters.js:

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

exports.APIAccess = countAPIAccess;

function countAPIAccess(user, cb) {
  var now = new Date();
  var year = now.getUTCFullYear();
  var month = format(now.getUTCMonth()+1)
  var day = [year, month, now.getUTCDate()].join('-');

  var key = 'counters:' + user;

  redis.multi().
    hincrby(key, year, 1).
    hincrby(key, month, 1).
    hincrby(key, year + '-', month, 1).
    hincrby(key, day, 1).
    hincrby(key, 'total', 1).
    exec(cb);
}

function format(n) {
  return ("0" + n).slice(-2);
}
```

Here we're exporting a function named `APIAccess` that clients call when they want to count an API access by a particular user. This function creates a `MULTI` transaction that bears several `HINCRBY` commands. For each of these commands, the base key is always the same – and is composed specifically for the given user ID. Each attribute key is then composed based on the current date. For instance, if the current date is `2014-01-15`, this will increment the counters named `2014`, `01`, `2014-01`, `2014-01-15` and `total`, all under the user-specific counter record. This way you will have counter buckets for each day, month, and year, as well as a grand total.

You will have to take some care maintaining the keys inside each Hash. After some time has passed, the number of keys for each active user will increase, occupying Redis memory and increasing operation latency.

4.1.11 Lists

Redis has another interesting type: lists. A list in Redis is internally implemented as a linked list, which means that it's cheap to insert and remove elements from it.

This, for instance, makes lists useful for implementing work queues. You insert work at one end of the list, and the workers pop out work from the other end of the list. Let's see how we could implement a work queue using Redis:

queue.js:

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

exports.push = push;
exports.pop = pop;

function push(work, cb) {
  redis.lpush('workqueue', JSON.stringify(work), cb);
}

function pop(cb) {
  redis.rpop('workqueue', function(err, work) {
    if (err) {
      cb(err);
    }
    else {
      if (work) {
        work = JSON.parse(work);
      }
      cb(null, work);
    }
  });
}
```

This queue module exports two functions: `push` and `pop`. The first one serves to push work items to the workers. The second one is for the workers to pop work from the queue.

Here is some code that exercises this module:

queue_test.js:

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

var missing = 10;

for(var i = 0 ; i < 10 ; i ++) {
  queue.push({some: 'work', id: i}, pushed);
}

function pushed(err) {
  if (err) {
    throw err;
  }
  if (-- missing == 0) {
    console.log('all work is pushed');
    poll();
  }
}

function poll() {
  queue.pop(popped);
}

function popped(err, work) {
  if (err) {
    throw err;
  }
  console.log('work:', work);
  if (! work) {
```

```
    setTimeout(poll, 1e3);
  }
  else {
    poll();
  }
}
```

This script above inserts 10 work items into the queue. Once they're all inserted, it goes out to pop them. If there is no more work, it waits for one second before trying to pop another item. Otherwise, it tries to pop another one immediately afterwards. You can execute this file:

```
$ node queue_test.js
all work is pushed
work: { some: 'work', id: 0 }
work: { some: 'work', id: 1 }
work: { some: 'work', id: 2 }
work: { some: 'work', id: 3 }
work: { some: 'work', id: 4 }
work: { some: 'work', id: 5 }
work: { some: 'work', id: 6 }
work: { some: 'work', id: 7 }
work: { some: 'work', id: 8 }
work: { some: 'work', id: 9 }
work: null
work: null
work: null
```

4.1.11.1 Avoid polling

In this last solution, the workers have to poll Redis for work, which is ugly, introduces a bit of overhead, and is also error-prone. Instead, a Redis client connection can use one of the list-blocking pop commands to block on a list while there are no elements. With this new knowledge, this is how we would then re-implement the queue module:

queue_block.js:

```
var Redis = require('redis');
var redis = require('./redis');

var popTimeout = 10;

exports.push = push;
exports.Worker = worker;

function push(work, cb) {
  redis.lpush('workqueue', JSON.stringify(work), cb);
}

function worker(fn) {
  var conn = Redis.createClient();

  next();

  function next() {
    conn.brpop('workqueue', popTimeout, popped);

    function popped(err, results) {
      if (err) {
```

```
          cb(err);
        }
        else {
          var work = results[1];
          if (work) {
            fn(null, JSON.parse(work));
          }
        }
        next();
      }
    }

    function close() {
      conn.quit();
    }

    return {
      close: close
    };
}
```

Here, instead of exposing a `pop` function, we expose a constructor for a worker. This constructor receives one function that will be called when a work item gets popped, or when an error occurs when doing it.

You can see here that we create one Redis connection per worker. This is because the blocking pop blocks the connection, only replying when the given timeout expires, or when an item gets popped.

4.1.11.2 Not losing work

One problem that may arise from any of the previous work-queue implementations is that work may be lost if the worker process goes down. If a worker dies while processing some work, that piece of work was already popped from Redis, and we have already lost it. In some applications this may not be a problem if it happens rarely; but in some others this may not be tolerable.

It usually comes down to whether the work has to be performed *at most once* or *at least once*. If you have to perform the work at least once, the operations resulting from this work should be idempotent: that is, if the same operation happens more than once, it will yield the same result. An example of this is an operation of propagating a user password change to a foreign system. It should not matter whether the change is propagated more than once, as it will yield the same result, which is setting the password in a foreign system.

There is usually a way of making operations execute *exactly once*, and it usually involves creating a unique operation identifier and making sure the same operation is not applied twice.

Let’s see how we could create such a system using Redis queues. First, you will need to instal this `cuid` for generating unique IDs:

```
$ npm install cuid --save
```

We’re now ready to create a version of the queue that’s safer:

queue_block_safe.js:

```
var cuid = require('cuid');
var Redis = require('redis');
var redis = require('./redis');
var EventEmitter = require('events').EventEmitter;

var popTimeout = 10;

exports.push = push;
exports.Worker = Worker;

function push(work, cb) {
  var id = cuid();
  var item = {
    work: work,
    created: Date.now(),
    id: id
  };

  redis.lpush('workqueue:in', JSON.stringify(item), function(err) {
    if (err) {
      cb(err);
    }
    else {
      cb(null, id);
    }
  });
}

function Worker(fn) {
  var conn = Redis.createClient();

  setImmediate(next);

  var worker = new EventEmitter();
  worker.close = close;

  return worker;

  function next() {
    conn.brpoplpush('workqueue:in', 'workqueue:processing', popTimeout, popped);

    function popped(err, item) {
      if (err) {
        worker.emit('error', err);
      }
      else {
        if (item) {
          var parsed = JSON.parse(item);
          fn.call(null, parsed.work, parsed.id, workFinished);
        }
      }

      function workFinished() {
        conn.lrem('workqueue:processing', 1, item, poppedFromProcessing);
      }

      function poppedFromProcessing(err) {
```

```
          if(err) {
            worker.emit('error', err);
          }
          next();
        }
      }
    }

    function close() {
      conn.quit();
    }
}
```

Here, our `next` function uses the `BRPOPLPUSH` command, which atomically pops from a list and pushes into another list. This makes sure that we always have the work in Redis while it's being processed. When the worker finishes processing the item, it calls a callback function (`workFinished`), which removes the item from the `work:processing` queue.

Now, to recover from dead workers, a process can be responsible for peeking into the `work:processing` list and requeueing the work items that have exceeded a certain execution time.

One thing to bear in mind is that, in the event of a load problem where the workers don't have enough capacity to consume the work in a timely fashion, the work items may eventually timeout and get requeued, only making the load problem worse. To avoid this you should a) set a long enough timeout, and b) log every requeue event and monitor its frequency so that you get alerted when it gets too high.

4.1.12 Sets

Redis has other types of data that allow multiple values: Redis sets allow you to store multiple *unsorted* values. They also allow you to quickly test membership or to calculate the intersection of two sets.

Sets are often used to group records. For instance, say that your application has several user groups: one for registered users, one for paying users, one for moderators and another for administrators. We can then create a module to manage the belonging to these groups:

user_sets.js:

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

exports.add = add;

function add(group, member, cb) {
  redis.sadd(key(group), member, cb);
}

exports.remove = remove;

function remove(group, member, cb) {
  redis.srem(key(group), member, cb);
}

exports.belongs = belongs;
```

```
function belongs(group, member, cb) {
  redis.sismember(key(group), member, function(err, belongs) {
    cb(err, belongs == 1);
  });
}

function key(group) {
  return 'group:' + group;
}
```

Here we're using some of the s-prefixed functions of Redis to manage sets. `exports.add` adds a member to a group and `exports.remove` uses `srem` to remove a member from a group. We can also test whether a certain user belongs to a given group. We can use this to verify whether a give user has permissions to execute certain sensible operations:

user_sets_example.js

```
var userSets = require('./user_sets');

userSets.add('admins', 'user1', function(err) {
  if (err) {
    throw err;
  }

  console.log('added user1 to group');

  ['user1', 'user2'].forEach(function(user) {
    userSets.belongs('admins', user, function(err, belongs) {
      if (err) {
        throw err;
      }

      console.log('%s belongs to group: %j', user, belongs);
    });
  });
});
```

The `SISMEMBER` Redis query has a fixed-time cost, making it very efficient for testing whether a certain member belongs to a given set. In our case this makes it efficient to, for instance, test whether a user belongs to a given user group before executing a privileged operation.

Adding a member to a set is an idempotent operation: by definition, a set will not hold repeated items.

We can now run this example:

```
$ node user_sets_example.js
added user1 to group
user1 belongs to group: true
user2 belongs to group: false
```

4.1.12.1 Intersecting sets

Redis sets allow you to calculate the intersection: given two or more sets, Redis can tell you which members are common to all of them. We can, for instance, calculate which

users are both moderators and paying users:

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

function key(group) {
  return 'group:' + group;
}

redis.sinter(key('mods'), key('paying'), function(err, users) {
  if (err) {
    throw err;
  }

  console.log('paying mods: %j', users);
});
```

4.1.13 Sorted Sets

Redis keeps elements in a set in no particular order: the order in which you add them is not necessarily the order that Redis retrieves them in, making them very useful for little else other that membership-related operations.

If you need sorted sets, Redis has your back: there are a group of Z-prefixed operations coming to your rescue.

Each element of a set has a score, which is a natural integer. All elements are stored, sorted, and indexed by this value, which means that you can retrieve all elements within a range of scores, all sorted by score.

Let's say that you are running an online collaborative real-time game and that you want to keep a ranking of scores for each given game. Each player has a score that can increase or decrease, and you want to present an up-to-date ranking with that score. Let's create a module to manage that:

game_scores.js:

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

exports.score = score;

function score(game, player, diff, cb) {
  redis.zincrby(key(game), diff, player, cb);
}

exports.rank = rank;

function rank(game, cb) {
  redis.zrevrange(key(game), 0, -1, "WITHSCORES", function(err, ret) {
    if (err) {
      cb(err);
    }
    else {
      var rank = [];
      for (var i = 0 ; i < ret.length ; i += 2) {
        rank.push({player: ret[i], score: ret[i+1]});
      }
      cb(null, rank);
    }
```

```
  });
}

function key(game) {
  return 'game:' + game;
}
```

This module exports two functions. The first, named `score`, accepts the name of a game, the name of a player, and a number; and just adds that number to the score of a player in that game.

If the player in that room does not exist, it gets created by Redis with a score of 0.

The second function is named `rank` and gives you a rank of a given game. Here we're using the `ZREVRANGE` Redis query that returns a range of elements in that set, sorted in reverse order of score. If we wanted this to return the user with the lowest score first, we would be using `ZRANGE` instead. We're requesting every element of the set by specifying `0` as the minimum range and `-1` as the maximum range. Giving `-1` as the maximum value makes the range have no upper bound, effectively returning all elements of that set.

We're also passing in the `WITHSCORES` option, which makes Redis interleave the scores in the response (one array element for the entry, one array element for the score, etc.). Here we're parsing the response and constructing a more appropriate rank array where each element has a `player` property and a `score` property.

We can now simulate a game using this module:

game_scores_example.js:

```
var gameScores = require('./game_scores');

var room = 'room1';

setInterval(function() {
  var player = 'player' + Math.floor(Math.random() * 10);
  gameScores.score(room, player, Math.floor(Math.random() * 10), function(err) {
    if (err) {
      throw err;
    }
  });
}, 1e2);

setInterval(function() {
  gameScores.rank(room, function(err, ranks) {
    if (err) {
      throw err;
    }
    console.log('%s ranking:\n', room, ranks);
  });
}, 1e3);
```

Here we're randomly incrementing the score of a random player every 100 milliseconds. We're also printing the current rank of the game every second:

```
$ node game_scores_example.js
room1 ranking:
 [ { player: 'player5', score: '14' },
   { player: 'player7', score: '11' },
   { player: 'player1', score: '10' },
   { player: 'player4', score: '5' } ]
room1 ranking:
 [ { player: 'player4', score: '20' },
   { player: 'player1', score: '20' },
   { player: 'player5', score: '14' },
   { player: 'player3', score: '14' },
   { player: 'player7', score: '11' },
   { player: 'player8', score: '10' },
   { player: 'player6', score: '4' } ]
room1 ranking:
 [ { player: 'player4', score: '32' },
   { player: 'player3', score: '30' },
   { player: 'player1', score: '20' },
   { player: 'player7', score: '17' },
   { player: 'player8', score: '14' },
   { player: 'player5', score: '14' },
   { player: 'player0', score: '7' },
   { player: 'player6', score: '4' } ]
```

4.1.14 Pub-sub

Besides all this key-values, lists, queues, and sets types and operations, you can also use Redis for managing inter-process communication. Redis provides a publish-subscribe model over named channels that allows message producers and message consumers to communicate using Redis as a message broker.

To publish a message to a channel, you use the `PUBLISH` command, passing in the channel name and the message string:

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

redis.publish('some channel', 'some message', function(err) {
  if (err) {
    console.error('error publishing:', err);
  }
});
```

Besides bare strings you can also publish complex objects by JSON-encoding them:

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

var message = {
  some: 'attribute',
  and: 'more'
};

redis.publish('some channel', JSON.stringify(message), function(err) {
  if (err) {
    console.error('error publishing:', err);
  }
});
```

To receive messages you have to dedicate one Redis connection to it, turning on the *subscriber* mode by issuing the `SUBSCRIBE` or the `PSUBSCRIBE` command. From that point

on the connection only allows commands that change the subscription set. Let's see this in action:

```
var redis = require('redis').createClient();

redis.on('message', function(channel, message) {
  console.log('new message from channeg %s: %j', channel, message);
});

redis.subscribe('some channel');
redis.subscribe('some other channel');
```

Here we are subscribing to two channels named "some channel" and "some other channel". Each time a message gets published in Redis to any of these channels, Redis distributes the message to all the active connections that have a subscription to this channel.

As you can see from the previous example, you can dynamically add subscriptions to the connection. You can also dynamically remove subscriptions from a connection by calling `redis.unsubscribe()`:

```
redis.unsubscribe('some channel');
```

If you're expecting to receive complex JSON-encoded objects instead of strings, you can parse the string like this:

```
redis.on('message', function(channel, message) {
  message = JSON.parse(message);
  console.log('new message from channeg %s: %j', channel, message);
});
```

4.1.15 Distributed Emitter

Node has a similar pattern to the Redis Pub-sub: the Event Emitter. The event emitter allows you to detach the event producer from the event consumer, but it's all working inside the same Node process. We can change it to make it work between processes:

distributed_emitter.js:

```
var Redis = require('redis');
var EventEmitter = require('events').EventEmitter;

module.exports = DistributedEmitter;

function DistributedEmitter() {

  // Redis stuff

  var redis = {
    pub: Redis.createClient(),
    sub: Redis.createClient()
  };

  redis.pub.unref();
  redis.sub.unref();

  redis.pub.on('error', onRedisError);
  redis.sub.on('error', onRedisError);
```

```
  redis.sub.on('message', function(channel, message) {
    old.emit.call(emitter, channel, JSON.parse(message));
  });

  // Emitter stuff

  var emitter = new EventEmitter();

  var old = {
    emit: emitter.emit,
    addListener: emitter.addListener,
    removeListener: emitter.removeListener
  };

  emitter.emit = function emit(channel, message) {
    redis.pub.publish(channel, JSON.stringify(message));
  };

  emitter.addListener = emitter.on = function addListener(channel, fn) {
    if (!emitter.listeners(channel).length) {
      subscribe(channel);
    }
    old.addListener.apply(emitter, arguments);
  };

  emitter.removeListener = function removeListener(channel, fn) {
    old.removeListener.apply(emitter, arguments);
    if (!emitter.listeners(channel).length) {
      unsubscribe(channel);
    }
  };

  emitter.close = function close() {
    redis.pub.quit();
    redis.sub.quit();
  };

  return emitter;

  function subscribe(channel) {
    redis.sub.subscribe(channel);
  }

  function unsubscribe(channel) {
    redis.sub.unsubscribe(channel);
  }

  function onRedisError(err) {
    emitter.emit('error', err);
  }
}
```

Here we have created a module that exports just one function: a constructor for our distributed emitter, that returns a modified event emitter.

When creating a distributed emitter, we start by setting up two Redis connections. One connection serves as a publisher connection and the other serves as a dedicated subscriber connection. This has to be like this because of the Redis protocol: when a Redis

connection enters a subscriber mode, it cannot emit commands other than ones that alter the subscriptions.

Then we call `unref()` on each of these connections. This makes sure the Node process does not quit just because we have one of these client connections open.

Then we proceed to listening for `error` events on each of the Redis connections, which we just propagate to the returned event emitter. This allows clients to listen for and handle Redis-specific errors.

We also listen for `message` events, which the Redis client emits when events come in from the Redis Pub-sub system through the client connection. When this happens, we just propagate the event into the local event emitter, allowing the event emitter subscribers to get it.

Next, we modify the returned event emitter, replacing the `emit`, `addListener`, `on` and `removeListener` methods. When emitting an event, instead of emitting locally, we just publish the event to Redis, using the event name as the channel name.

We also wrap the `addListener` and `on` methods, which are used for listening to event types. When any of these are called, if it's the first subscription for this given event type, we subscribe to the respective channel on Redis. We then revert to the default behaviour, which is to add a listener function to Redis.

Similarly, we wrap the `removeListener` method to catch the case when there are no more listeners for a specific event type, in which case we cancel the respective channel subscription on our Redis client connection.

We keep around the event emitter old methods in the `old` variable so that we can call them from inside the wrapper methods.

Finally, we implement a specific `close` method that closes the Redis connection.

Let's now create a client module that instantiates two distributed emitters, using Redis to communicate between them, as would happen if they were in two separate processes:

distributed_emitter_example.js:

```
var DistributedEmitter = require('./distributed_emitter');

var emitter1 = DistributedEmitter();
var emitter2 = DistributedEmitter();

var channels = ['channel 1', 'channel 2'];

channels.forEach(function(channel) {
  emitter1.on(channel, function(msg) {
    console.log('%s message:', channel, msg);
  });
});

channels.forEach(function(channel) {
  setInterval(function() {
    emitter2.emit(channel, {time: Date.now()});
  }, 1e3);
});
```

This example client module creates these two distributed emitters. The first one subscribes to two event types, printing out these events as they come in. The second one emits these two event types every second. The payload of the event is just a timestamp. Let's run this:

```
node distributed_emitter_example.js
channel 1 message: { time: 1421746397411 }
channel 2 message: { time: 1421746397411 }
channel 1 message: { time: 1421746398418 }
channel 2 message: { time: 1421746398420 }
channel 1 message: { time: 1421746399426 }
channel 2 message: { time: 1421746399427 }
channel 1 message: { time: 1421746400431 }
channel 2 message: { time: 1421746400432 }
...
```

4.1.15.1 Beware of race conditions

This distributed event emitter behaves differently from a normal event emitter in one fundamental way: it propagates events by doing I/O. In Node, I/O is an asynchronous operation, while all the event emitter typical operations are just local and synchronous. This has an impact on the timing for local processes. For instance, consider the following code using a normal event emitter:

local_event_emitter.js:

```
var EventEmitter = require('events').EventEmitter;

var emitter = new EventEmitter();

emitter.on('some event', function() {
  console.log('some event happened');
});

emitter.emit('some event');
```

Running this would yield, as expected:

```
$ node local_event_emitter.js
some event happened
```

Good, all looks normal. Now let's try replacing the event emitter by one of our distributed emitters:

distributed_event_emitter_race.js:

```
var DistributedEmitter = require('./distributed_emitter');

var emitter = DistributedEmitter();

emitter.on('some event', function() {
  console.log('some event happened');
});

emitter.emit('some event', 'some payload');
```

Let's try to run this version then:

```
$ node distributed_event_emitter_race.js
```

The process just exits without outputting anything. This means that, when we call the `on` method, the distributed emitter will subscribe to the channel for the first time. But that envolves some I/O, so this is not done immediately: the command has to go to the network layer, has to be received, parsed and executed by Redis, and then a response comes back to Node. Before all this I/O even happened, though, we emit an event (the last line of the previous file). This event also involves I/O, which is also asynchronous. What happens is that both commands are racing to get to Redis using two different client connections. If the `PUBLISH` command reaches Redis before the `SUBSCRIBE` command, our client will never see that event.

To see exactly the sequence of events in Redis, we can use a little trick using the Redis command-line client bundled with Redis. With it you can monitor the Redis server to inspect which commands are being issued:

```
$ redis-cli monitor
OK
```

Now you can execute the previous node script and observe which commands happen on Redis:

```
1421747925.034001 [0 127.0.0.1:52745] "info"
1421747925.034762 [0 127.0.0.1:52746] "info"
1421747925.037992 [0 127.0.0.1:52745] "publish" "some event" "\"some payload\""
1421747925.039248 [0 127.0.0.1:52746] "subscribe" "some event"
```

There you go: the `subscribe` command arrived *after* the `publish` command, losing the race.

This serves to illustrate that a Pub-sub mechanism is not a persistent queue: listeners will only get new events after the subscription request is processed by the server, and not before. Remembering this will probably save you a lot of future headaches.

5. CouchDB

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.

5.1 Starting off

If you don't have CouchDB already installed, you can head to the official website (http://couchdb.apache.org/) to download and instal 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 instal 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":"1-56b8a3a98ed03fbb3a80\
4751a38611b2"}
```

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":"1-56b8a3a98ed03fbb3a804751a386\
11b2","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":"2-221c0d018a4442452549\
3a1c1ff34828"}
```

5.2 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 instal 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 instal 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.

But wait – before that, we need to perform a small but important tuning.

5.3 Overriding the HTTP agent socket pool

The Node HTTP Client library contains this feature called an Agent. Each request can have a specific agent, and each agent has a socket pool so that TCP connections to the same server and port can be re-used. If it's not specified, Node has a default Agent, which can be accessed using `http.globalAgent`. This agent comes with a default of five maximum open sockets per hostname and port. This means that, unless changed, each

Node process can only have a maximum of five ongoing HTTP requests to the same CouchDB server. What should we do?

The easiest thing to do is to override the default agent's maximum amount of sockets and set it to a very high number, like this:

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

http.globalAgent.maxSockets = Number.MAX_VALUE;
```

In practice, if you're running many parallel requests, your process will run out of file descriptors before reaching this value. To minimise the chance of this happening (at the expense of the app having to wait for available sockets), you should set this to an expected value that you can calculate based on the expected peak usage of your app for one given Node process.

You can place this setting at the beginning of your `couchdb.js` file:

couchdb.js:

```
var http = require('http');
http.globalAgent.maxSockets = Number(process.env.HTTP_MAX_SOCKETS) || 1024;

var nano = require('nano');

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

5.4 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.

5.5 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.
    ...
```

5.6 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 instal 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`.

5.7 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 instal `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));
}
```

5.7.1 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.

5.8 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.

5.8.1 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
```

5.8.2 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.

5.8.3 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 instal 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);
      }
    }
  }
}
```

5.9 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:

5.9.1 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
```

5.9.1.1 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' }
...
```

5.9.2 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.

5.9.3 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.

5.9.3.1 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.

5.9.3.2 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.

5.9.4 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….
```

5.10 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 instal 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.

5.10.1 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.

5.10.2 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.

5.10.3 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 instal 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);
  }
});
```

5.10.4 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).