# The CLI Book

**Writing successful Command Line Clients with NODE.JS**

ROBERT KOWALSKI

# Table of Contents

# Free Sample: The CLI Book

It’s finally here!

After half a year of hard work, the first version of *The CLI Book* is released!

Thank you for downloading this free sample of *The CLI Book*!

**The sample contains a section of the first part of the book and a section of the second, hands-on focussed part of the book.**

You can buy the full version at http://theclibook.com

I am very happy about feedback. Please send and feedback or corrections to theclibook@kowalski.gd. You can also contact me on twitter: @robinson_k

Enjoy,

Robert

# Preface

Command line clients are everywhere. Almost everyone, at least in tech, is using them.

There are a lot of successful command line clients out there: the Linux project has git and the Node.js project has npm. We use some of them multiple times per day. Apache CouchDB recently got nmo (speak: nemo), a tool to manage the database cluster. We can learn a lot from successful command line interfaces in order to write better command line clients.

When I started to get interested in command line clients I realised that there are a lot of discussions and informations on the web about writing APIs. The web is full of tutorials to teach you how to build APIs, especially  REST-APIs, but almost nothing can be found about writing good CLIs. This book tries to explain what makes a good CLI. In the second part of the book we will build a small command line client to learn how to use Node.js to create great command line clients that people love.

The goal of the book is to show the principles to build a successful command line client. The provided code should give you a good understanding what is important to build successful command line clients and how you could implement them.

Every section has their own code examples. Before you run the code, you have to run `npm install` in the folder that belongs to the section.

I trust you and published this book without DRM. Please buy a copy at http://theclibook.com in case you did not buy the book.

I am very happy about feedback. Please send and feedback or corrections to theclibook@kowalski.gd. You can also contact me on twitter: @robinson_k

I hope you enjoy the book – please recommend it in case you like it.

# What makes a good CLI?

In this chapter we will take a look at successful command line clients and what they are doing pretty well, which will help us to understand the problems users face using the Terminal. Understanding the problems of our users will help us to build better command line clients with Node later in the book.

Let’s take a look at how people usually use a CLI: most of the time a human sits in front of a keyboard and interacts with a terminal. We want to use simple and recognisable commands for our CLI. Sadly just easy recognisable commands don’t get us very far right now.

Maybe the problem is easier to understand if we take a look at something what I would call a bad CLI:

```
$ mycli -A -a 16 r foo.html
error: undefined is not a function
```

In my example I have to enter cryptic commands which is answered by a very cryptic error message. What does -A -a 16 and r mean? Why I am getting an error back, am I using it wrong? What does the error mean and how can I get my task done?

So what makes a good CLI? Let’s try it with the following three principles:

- you never get stuck
- it is simple and supports powerusers
- you can use it for all the things!

**In short:** A successful CLI is successful because its users are successful and happy.

## You never get stuck

Nobody likes to be in a traffic jam, stuck, just making a few meters per minute. We want to reach our target destination, that’s all we want! The same applies for our users. Both developers and users are extremely unhappy when the tools they use are standing in their way. They just want to get their task done.

So what does, „You never get stuck“ mean, exactly? It means that we should always offer our users a way to solve their task, a command should never be a dead end. Additionally the developers of the CLI should avoid every source of friction in their tool.

Let’s take a look at me, trying to use git:

```
$ git poll
git: 'poll' is not a git command. See 'git --help'.

Did you mean this?
  pull
```

In this example I entered a wrong command. git answers friendly: „Hey Robert, it looks like you entered a wrong command, but if you type in `git --help`, you can list all the existing commands. And hey, it just looks like you mistyped git pull, did you mean git pull?“

git offers us a way to continue our work and finish the task.

And if we take a look at npm, another successful CLI client, we'll see the same concept:

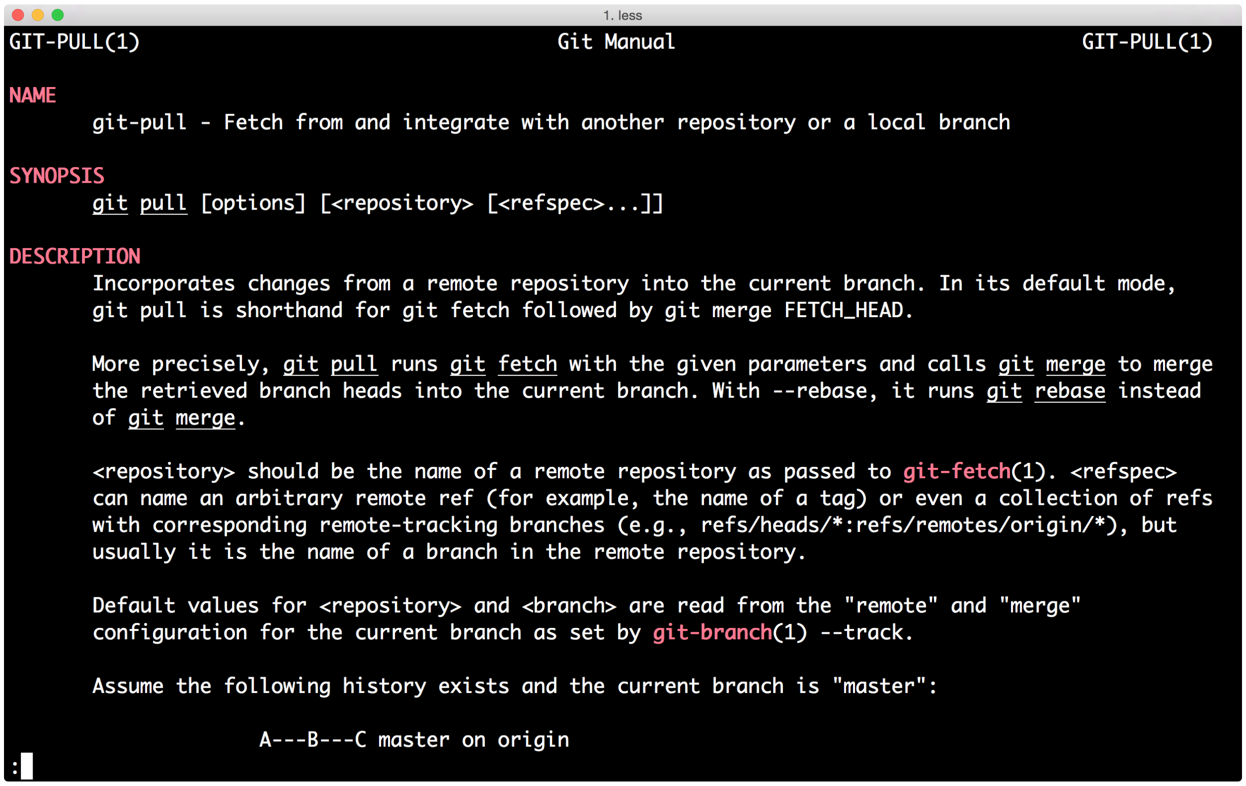
```
$ npm ragrragr
Usage: npm <command>

where <command> is one of:
    access, add-user, adduser, apihelp, author, bin, bugs, c,
    cache, completion, config, ddp, dedupe, deprecate, dist-tag,
    dist-tags, docs, edit, explore, faq, find, find-dupes, get,
    help, help-search, home, i, info, init, install, issues, la,
    link, list, ll, ln, login, logout, ls, outdated, owner,
    pack, prefix, prune, publish, r, rb, rebuild, remove, repo,
    restart, rm, root, run-script, s, se, search, set, show,
    shrinkwrap, star, stars, start, stop, t, tag, test, tst, un,
    uninstall, unlink, unpublish, unstar, up, update, upgrade,
    v, verison, version, view, whoami

npm <cmd> -h     quick help on <cmd>
npm -l           display full usage info
npm faq          commonly asked questions
npm help <term>  search for help on <term>
npm help npm     involved overview

Specify configs in the ini-formatted file:
    /Users/robert/.npmrc
or on the command line via: npm <command>   key value
Config info can be viewed via: npm help config

npm@2.7.4 /Users/robert/.nvm/versions/node/v0.12.2/lib/node_modules/npm
```

In this example I try to put garbage into npm, so npm answers friendly: „Hey Robert, I don't know that command, but here are all the commands that would be possible. You can use them like this and get help about them by typing in npm help <command>.“

Like git, npm immediately offers help to enable me to finish my task, even if I have no idea how to use npm at all.

## Still lost?

What if I still need help? Maybe I want to get some help before I just try out commands. Turns out there is a quite reliable way to ship documentation on Unix or Linux, man-pages!

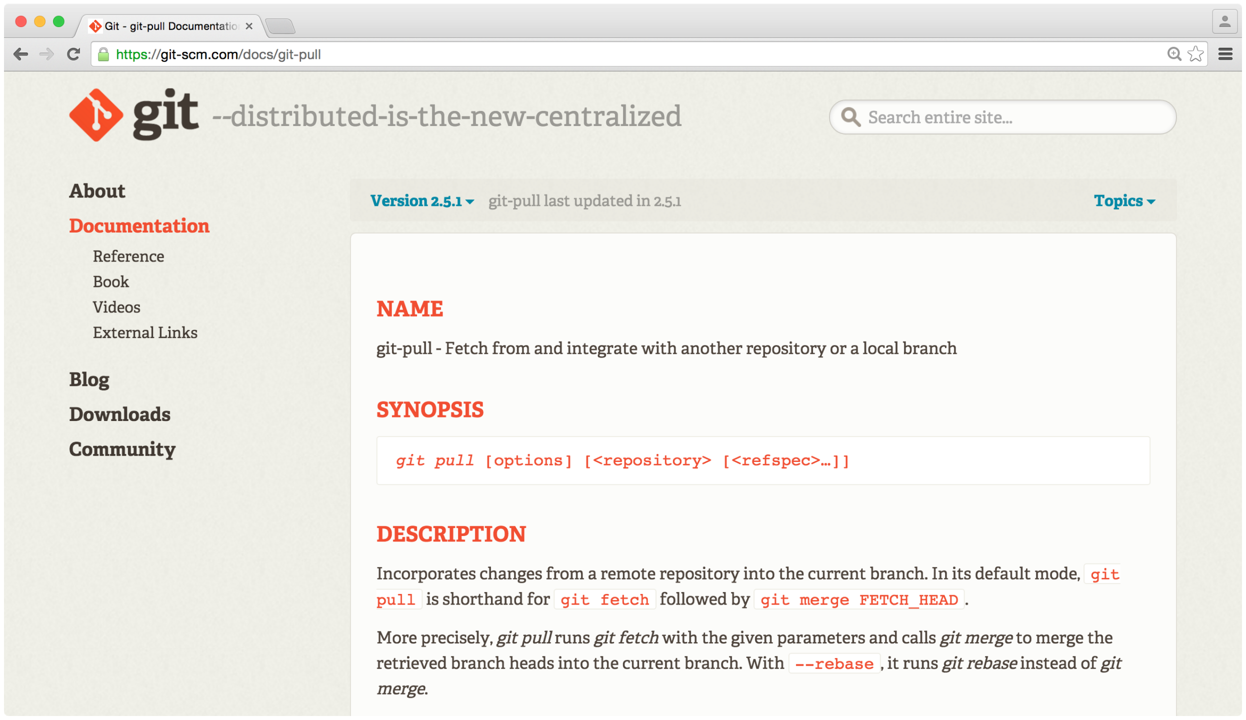


```
GIT-PULL(1)                        Git Manual                        GIT-PULL(1)

NAME
       git-pull - Fetch from and integrate with another repository or a local branch

SYNOPSIS
       git pull [options] [<repository> [<refspec>...]]

DESCRIPTION
       Incorporates changes from a remote repository into the current branch. In its default mode,
       git pull is shorthand for git fetch followed by git merge FETCH_HEAD.

       More precisely, git pull runs git fetch with the given parameters and calls git merge to merge
       the retrieved branch heads into the current branch. With --rebase, it runs git rebase instead
       of git merge.

       <repository> should be the name of a remote repository as passed to git-fetch(1). <refspec>
       can name an arbitrary remote ref (for example, the name of a tag) or even a collection of refs
       with corresponding remote-tracking branches (e.g., refs/heads/*:refs/remotes/origin/*), but
       usually it is the name of a branch in the remote repository.

       Default values for <repository> and <branch> are read from the "remote" and "merge"
       configuration for the current branch as set by git-branch(1) --track.

       Assume the following history exists and the current branch is "master":

                     A---B---C master on origin
:
```

*Figure 1. The man-page for git pull*

Man-pages are quite nice, as you don't need the internet to open them. You can also stay in the same terminal window to read them and don't have to switch to another window, e.g. a browser.

But some users don't know about man-pages or they don't like to use them. Additionally many of them will be on Windows which can't handle man-pages natively, so git and npm offer their documentation as webpages, too:


Git - git-pull Documentation
https://git-scm.com/docs/git-pull
git --distributed-is-the-new-centralized
Search entire site...
About
Documentation
Reference
Book
Videos
External Links
Blog
Downloads
Community
Version 2.5.1
git-pull last updated in 2.5.1
Topics

NAME

git-pull - Fetch from and integrate with another repository or a local branch

SYNOPSIS

```
git pull [options] [<repository> [<refspec>…]]
```

DESCRIPTION

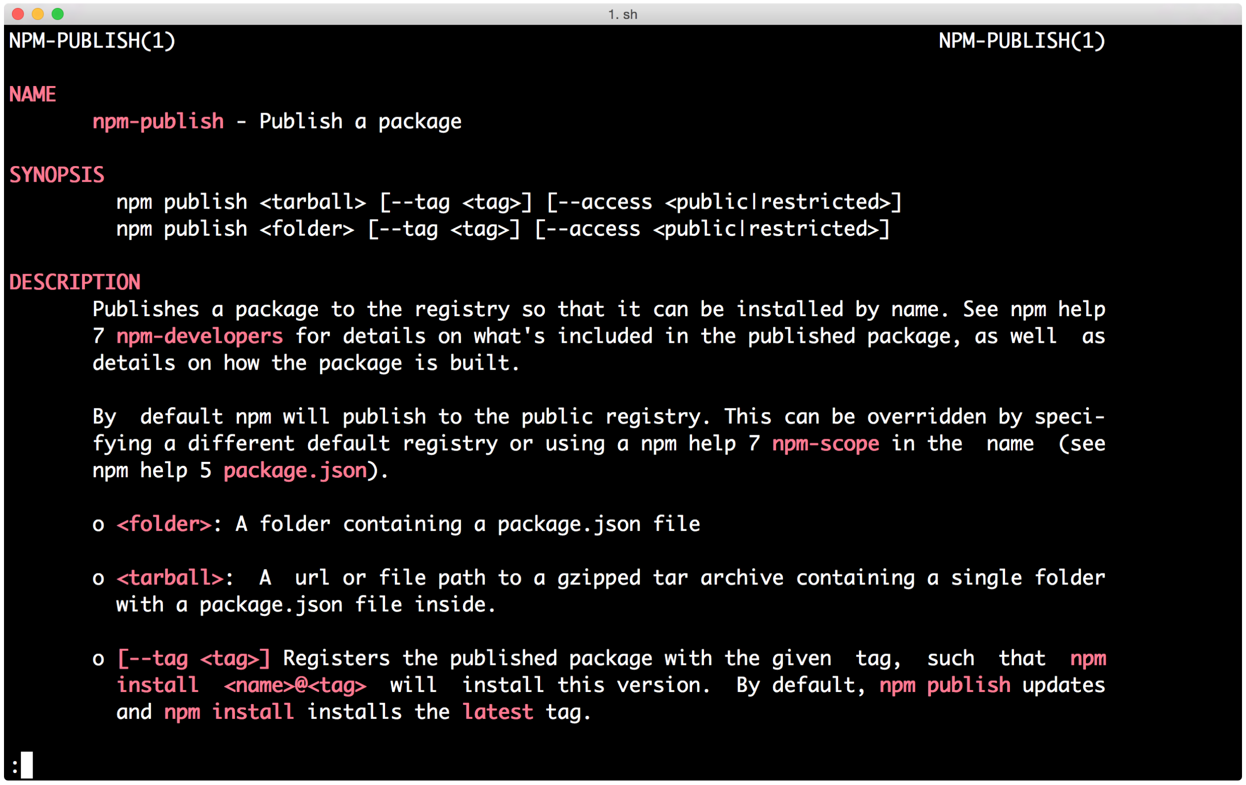
Incorporates changes from a remote repository into the current branch. In its default mode, `git pull` is shorthand for `git fetch` followed by `git merge FETCH_HEAD`.

More precisely, *git pull* runs *git fetch* with the given parameters and calls *git merge* to merge the retrieved branch heads into the current branch. With `--rebase`, it runs *git rebase* instead of *git merge*.

*Figure 2. The documentation website of the git project*

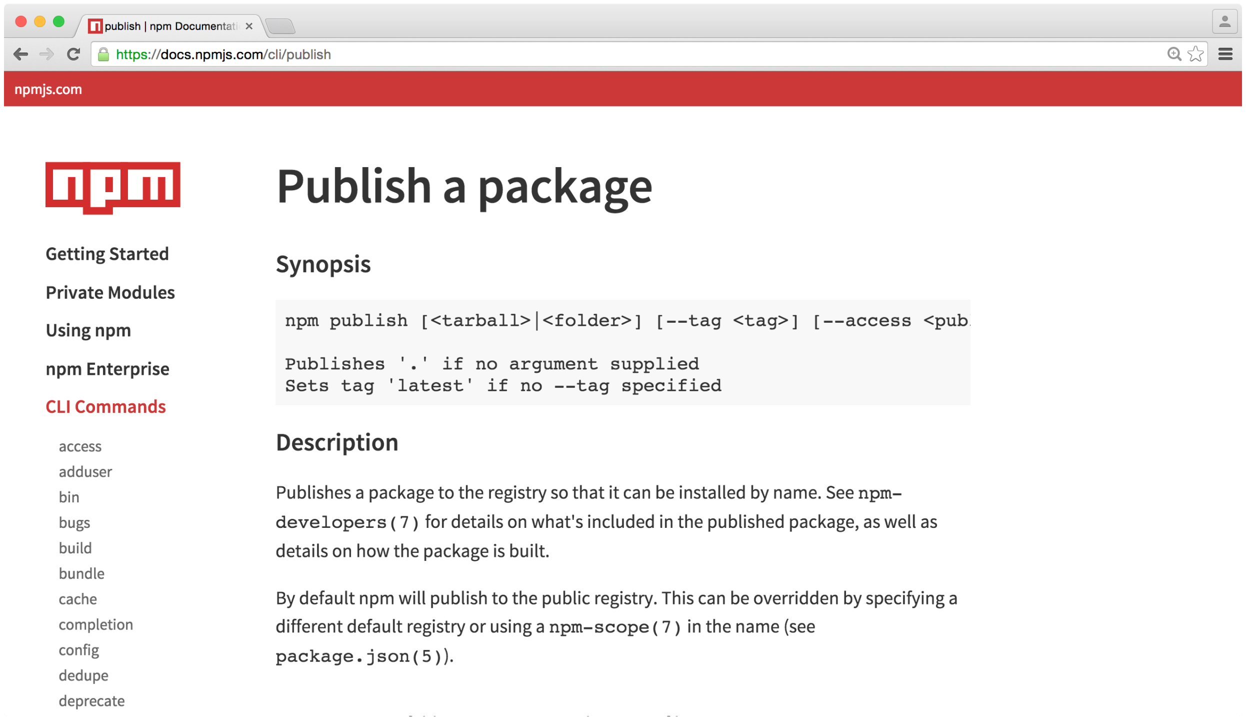
Both git and npm are making use of a trick: they write their documentation once (e.g. in Markdown or Asciidoc) and use the initital source as the base for the different formats of their docs. Later they convert them to different formats, e.g. to html.

If you take a look at the man-pages of git and npm, you will notice that their websites are basically framing the content from the man-page with a header and a sidebar.



```
NPM-PUBLISH(1)                                                  NPM-PUBLISH(1)

NAME
       npm-publish - Publish a package

SYNOPSIS
         npm publish <tarball> [--tag <tag>] [--access <public|restricted>]
         npm publish <folder> [--tag <tag>] [--access <public|restricted>]

DESCRIPTION
       Publishes a package to the registry so that it can be installed by name. See npm help
       7 npm-developers for details on what's included in the published package, as well  as
       details on how the package is built.

       By  default npm will publish to the public registry. This can be overridden by speci-
       fying a different default registry or using a npm help 7 npm-scope in the  name  (see
       npm help 5 package.json).

       o <folder>: A folder containing a package.json file

       o <tarball>:  A  url or file path to a gzipped tar archive containing a single folder
         with a package.json file inside.

       o [--tag <tag>] Registers the published package with the given  tag,  such  that  npm
         install  <name>@<tag>  will  install this version.  By default, npm publish updates
         and npm install installs the latest tag.

:
```

*Figure 3. The manpage for npm publish*



Publish a package

Synopsis

```
npm publish [<tarball>|<folder>] [--tag <tag>] [--access <pub

Publishes '.' if no argument supplied
Sets tag 'latest' if no --tag specified
```

Description

Publishes a package to the registry so that it can be installed by name. See `npm-developers(7)` for details on what's included in the published package, as well as details on how the package is built.

By default npm will publish to the public registry. This can be overridden by specifying a different default registry or using a `npm-scope(7)` in the name (see `package.json(5)`).

*Figure 4. The documentation website of npm*

## Error handling

Sometimes things go still horribly wrong… Let's take a look at my example for a bad CLI again:

```
$ mycli -A -a 16 r foo.py
events.js:85
      throw er; // Unhandled 'error' event
            ^
Error: ENOENT, open 'cli.js'
    at Error (native)
```

In this case we are getting back a stacktrace without much context. For most people these stacktraces look quite cryptic, especially for people that don't write Node.js on a daily basis.

And it is even worse: I really can't tell if I just hit a bug in the command line client or if I am just using the CLI in a wrong way. Looking at that small terminal, with no idea what to do, I get extremely unhappy and so our users will get unhappy.

One thing nmo supports is „usage errors“ — here is what they look like:

```
$ nmo cluster dsf
ERR! Usage:

nmo cluster get [<clustername>], [<nodename>]
nmo cluster add <nodename>, <url>, <clustername>
nmo cluster join <clustername>
```

If a user tries to use a command in a wrong way, nmo will tell them immediately how they can use the command to get their job done. No need to open the documentation.

nmo also shows stacktraces to a user, if nmo crashes for serious reasons:

```
$ nmo cluster join anemone
ERR! df is not defined
ERR! ReferenceError: df is not defined
ERR!     at /Users/robert/apache/nmo/lib/cluster.js:84:5
ERR!     at cli (/Users/robert/apache/nmo/lib/cluster.js:68:27)
ERR!     at /Users/robert/apache/nmo/bin/nmo-cli.js:32:6
ERR!
ERR! nmo: 1.0.1 node: v0.12.2
ERR! please open an issue including this log on
https://github.com/robertkowalski/nmo/issues
```

nmo adds the current nmo and node version to the stacktrace, like npm does. We also ask the user to copy the stacktrace and to open an issue containing the stacktrace.

The reports make it easy for the team to identify the bug, solve it, and release a new version of nmo by seeing the stacktrace.

And again the user is not stuck. The user gets help to solve their task, in the worst case we help them in our issue tracker.

# Part 2

After a sample of the first part of the book, the sample for the second part of the book starts here.

# Writing a database administration tool with Node.js

In this part of the book we will write a database administration tool named lounger and follow the principles that make a good CLI. The code for every section can be found in the zip file that comes with the book. If you want to play with the code, don't forget to run `npm install` in the folder of the section you want to test. The code also has a testsuite, you can run it with `npm test`.

After we wrote the code that bootstraps the client, you can run the client directly with `node bin/lounger-cli`. If you prefer to run the client like an installation from the registry, type `npm link` in the directory of the section you want to test. Afterwards you can run the version that is currently linked with `lounger`.

## Why use Node.js?

I sometimes get asked why I write command line clients using Node.js. For me the main reasons are:

- a huge ecosystem with modules in every flavour
- very fast development speed
- writing JavaScript is fun!

For me these three reasons make Node.js the perfect platform to write command line clients.

## Setup

We will write our command line client in ES2015 (also known as ES6), which is the most recent version of JavaScript. In order to use it, we have to install Node v4 from https://nodejs.org. If you want to support older Node.js versions, I can recommend the Babel transpiler to transpile ES6 code to ES5 compatible code. You can get Babel at https://babeljs.io/.

The tool that we'll write will be a small database administration tool for CouchDB / PouchDB. There are multiple ways to get a development database server up and running.

One way is to install Erlang and CouchDB for your Operating System. You can download official packages at http://couchdb.apache.org and many Linux distributions have CouchDB in their package repository, too.

I think the easiest way is to use the PouchDB Server that is available in the attached source code for the book or to get a CouchDB instance at https://cloudant.com which is free until you hit a limit.

## Using the PouchDB database server

The database server is located in `sourcecode/database`, in order to use it we have to install the needed dependencies:

```
$ cd sourcecode/database
$ npm install
```

To boot the database we just run:

```
$ npm run start
```

We can now interact with the database server via HTTP, as CouchDB and PouchDB are databases with an HTTP API:

```
$ curl -XGET http://127.0.0.1:5984/

{"express-pouchdb":"Welcome!","version":"1.0.1","vendor":{"name":"PouchDB
authors","version":"1.0.1"},"uuid":"4fad2c01-ba32-4249-8278-8786e877c397"}
```

Let’s create a database called `people`:

```
$ curl -XPUT http://127.0.0.1:5984/people

{"ok":true}
```

We can now insert documents into our database `people`:

```
$ curl -XPOST http://127.0.0.1:5984/people -d '{"name": "Rocko Artischocko", \
 "likes": ["Burritos", "Node.js", "Music"] }' -H 'Content-Type: application/json'

{"ok":true,"id":"21b5ad83-0ad6-47c7-86f8d9636113160a","rev":"1-
411894affa038a6fd7a164e1bfd84146"}
```

Using the `id` we can retrieve the documents from the database:

```
$ curl -XGET http://127.0.0.1:5984/people/21b5ad83-0ad6-47c7-86f8-d9636113160a

{"name":"Rocko Artischocko","likes":["Burritos","Node.js","Music"],"_id":"21b5ad83-0ad6-
47c7-86f8-d9636113160a","_rev":"1-411894affa038a6fd7a164e1bfd84146"}
```

Great! We have a database up and running!

# Troubleshooting

## Getting curl

curl is a command line client for HTTP requests. It is available for all major Operating Systems. OSX users can install it using `brew` and for Windows there are Windows builds available at http://curl.haxx.se/download.html.

## File watchers

On Linux I got an error because my user already watched too much files:

```
$ npm run start
> theclibook-database@1.0.0 start /home/rocko/clibook/sourcecode/database
> pouchdb-server --in-memory

fs.js:1236
    throw error;
    ^

Error: watch ./log.txt ENOSPC
    at exports._errnoException (util.js:874:11)
    at FSWatcher.start (fs.js:1234:19)
    at Object.fs.watch (fs.js:1262:11)
    at Tail.watch (/home/rocko/clibook/sourcecode/database/node_modules/pouchdb-
server/node_modules/tail/tail.js:83:32)
    at new Tail (/home/rocko/clibook/sourcecode/database/node_modules/pouchdb-
server/node_modules/tail/tail.js:72:10)
    at /home/rocko/clibook/sourcecode/database/node_modules/pouchdb-
server/lib/logging.js:69:20
    at FSReqWrap.cb [as oncomplete] (fs.js:212:19)

npm ERR! Linux 3.13.0-71-generic
npm ERR! argv "/home/rocko/.nvm/versions/node/v4.2.3/bin/node"
"/home/rocko/.nvm/versions/node/v4.2.3/bin/npm" "run" "start"
npm ERR! node v4.2.3
npm ERR! npm  v3.5.1
npm ERR! code ELIFECYCLE
npm ERR! theclibook-database@1.0.0 start: `pouchdb-server --in-memory`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the theclibook-database@1.0.0 start script 'pouchdb-server --in
-memory'.
npm ERR! Make sure you have the latest version of node.js and npm installed.
npm ERR! If you do, this is most likely a problem with the theclibook-database package,
npm ERR! not with npm itself.
npm ERR! Tell the author that this fails on your system:
npm ERR!     pouchdb-server --in-memory
npm ERR! You can get information on how to open an issue for this project with:
npm ERR!     npm bugs theclibook-database
npm ERR! Or if that isn't available, you can get their info via:
npm ERR!     npm owner ls theclibook-database
npm ERR! There is likely additional logging output above.

npm ERR! Please include the following file with any support request:
npm ERR!     /home/rocko/clibook/sourcecode/database/npm-debug.log
```

I fixed it with by raising the limit using this command:

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf && sudo sysctl
-p
```

## A simple status check

Our first command will check if the database is up and running. Our users can take a look if the database server is running and we can use the command internally for the commands which require a running database.

The command to check if a database server is online will look like this:

```
$ lounger isonline http://192.168.0.1:5984
http://192.168.0.1:5984 is up and running
```

The API would look like this:

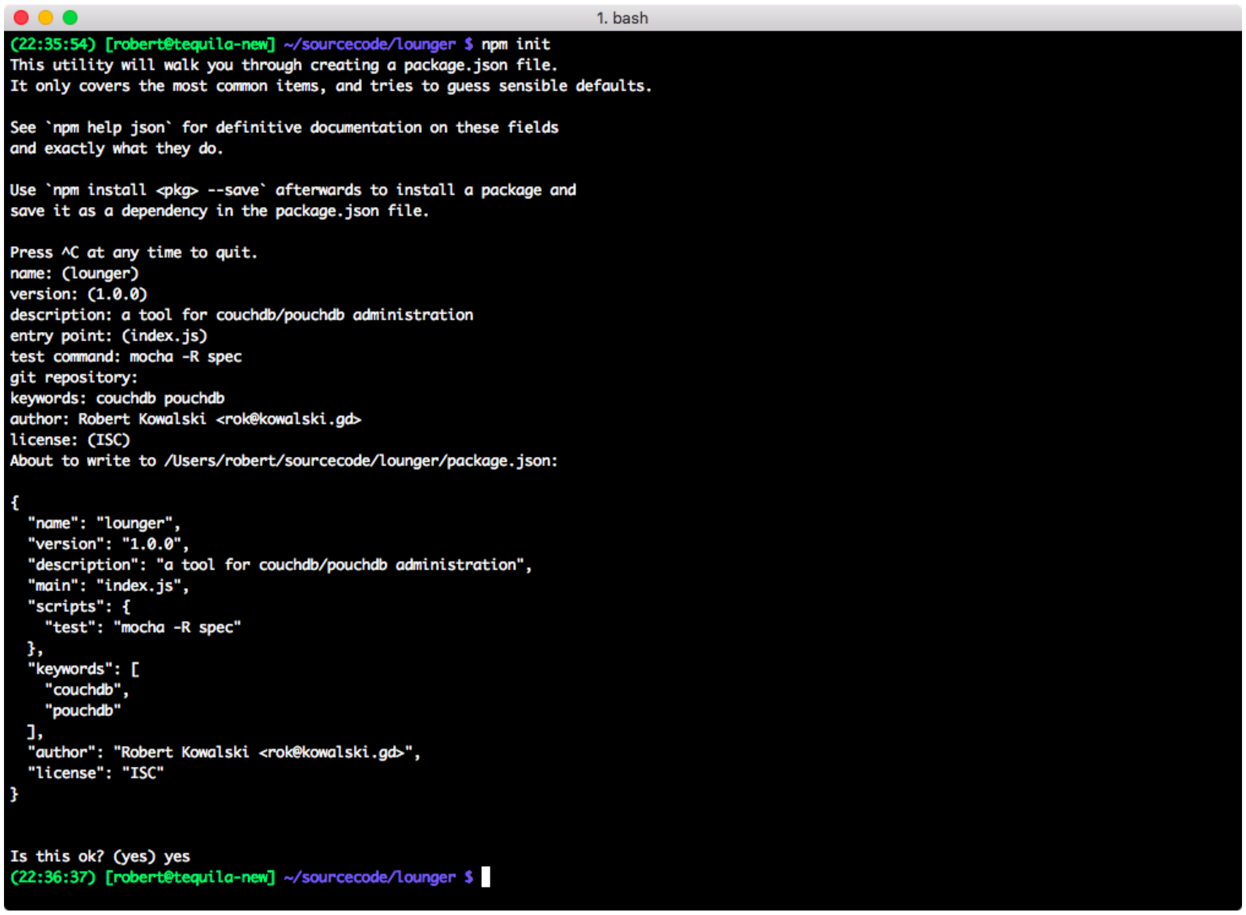
```
$ lounger.commands.isonline('http://example.com')
```

### Getting started from scratch

To get started we have to create a `package.json` file. Luckily npm provides a nice assistant to create those:

```
$ npm init
```

We then just answer the questions npm asks us.



```
(22:35:54) [robert@tequila-new] ~/sourcecode/lounger $ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (lounger)
version: (1.0.0)
description: a tool for couchdb/pouchdb administration
entry point: (index.js)
test command: mocha -R spec
git repository:
keywords: couchdb pouchdb
author: Robert Kowalski <rok@kowalski.gd>
license: (ISC)
About to write to /Users/robert/sourcecode/lounger/package.json:

{
  "name": "lounger",
  "version": "1.0.0",
  "description": "a tool for couchdb/pouchdb administration",
  "main": "index.js",
  "scripts": {
    "test": "mocha -R spec"
  },
  "keywords": [
    "couchdb",
    "pouchdb"
  ],
  "author": "Robert Kowalski <rok@kowalski.gd>",
  "license": "ISC"
}


Is this ok? (yes) yes
(22:36:37) [robert@tequila-new] ~/sourcecode/lounger $
```

*Figure 5. The assistant from npm init to create a package.json*

Additionally we have to create three folders: `test`, `lib` and `bin`. `test` will contain our unit and integration tests, `lib` will contain the core of our command line client. The `bin` folder will contain a small wrapper that will boot up the core of our client.

CouchDB and PouchDB both return a welcome message when we access the root url at http://localhost:5984

```
$ curl localhost:5984
```

CouchDB returns:

```
{"couchdb":"Welcome","uuid":"17ed4b2d8923975cf658e20e219faf95","version":"1.5.0","vendor"
:{"version":"14.04","name":"Ubuntu"}}
```

PouchDB returns:

```
{"express-pouchdb":"Welcome!","version":"1.0.1","vendor":{"name":"PouchDB
authors","version":"1.0.1"},"uuid":"4fad2c01-ba32-4249-8278-8786e877c397"}
```

We will make use of this behaviour to check if the database is online.

As already mentioned in Why use Node.js? Node.js has a great ecosystem. There are many battle proven modules that help us to solve our tasks.

For our status check we will use the module `request` to handle our HTTP requests. `mocha` will run our testsuite and `nock` helps us to mock HTTP responses without the need to boot a database instance for the testsuite.

The arguments `--save` and `--save-dev` will add the packages to the `dependencies` and `devDependencies` section of our `package.json`. `devDependencies` are needed just for development, not for running the package in production:

```
$ npm i --save request
$ npm i --save-dev mocha nock
```

After running the commands we should have everything we will need for now.

**TIP**

*Choose your own flavours*

There are many good test runners for Node.js, some alternatives to `mocha` are the npm modules `tap`, `tape` or `lab`

My `package.json` looks like this now:

```json
{
  "name": "lounger",
  "version": "1.0.0",
  "description": "a tool for couchdb/pouchdb administration",
  "main": "lib/lounger.js",
  "directories": {
    "test": "test"
  },
  "dependencies": {
    "request": "^2.67.0"
  },
  "devDependencies": {
    "mocha": "^2.3.4",
    "nock": "^5.2.1"
  },
  "scripts": {
    "test": "mocha -R spec"
  },
  "keywords": [
    "couchdb",
    "pouchdb"
  ],
  "author": "Robert Kowalski <rok@kowalski.gd>"
}
```

This book is not focussed on different development techniques like TDD, but if you are really into Test-Driven-Development, you can write failing tests with mocha before we implement the actual code. A few suggestions:

1. it detects if the database is online

2. it detects offline databases

3. it detects if something is online, but not a CouchDB/PouchDB

4. it only accepts valid urls

Written in mocha and ES6 we get a few failing tests in `test/isonline.js`:

```
'use strict';

const assert = require('assert');
const nock = require('nock');

describe('isonline', () => {

  it('detects if the database is online', () => {

    assert.equal('foo', 'to implement');
  });

  it('detects offline databases', () => {

    assert.equal('foo', 'to implement');
  });

  it('detects if something is online, but not a CouchDB/PouchDB', () => {

    assert.equal('foo', 'to implement');
  });

  it('just accepts valid urls', () => {

    assert.equal('foo', 'to implement');
  });
});
```

To run the testsuite, we have to type either `npm test` or `npm t` on the terminal. The code of this section can be found in `sourcecode/client-boilerplate`.

## The internals of the command

Let's create and edit the file `lib/isonline.js`. The filename is important, as we will use the name of the file later during the bootstrap of the client. As a first step, we have to require our dependency `request`:

```
'use strict';

const request = require('request');
```

To make a request we create the function `isOnline` which will take an url and send the request:

```
function isOnline (url) {
  return new Promise((resolve, reject) => {
    request({
      uri: url,
      json: true
    }, (err, res, body) => {
```

If there is no HTTP service at all listening on the specified url we resolve the Promise with an object with contains the url as a key, and `false` as a value:

```
      if (err && (err.code === 'ECONNREFUSED' || err.code === 'ENOTFOUND')) {
        return resolve({[url]: false});
      }
```

For all other errors we reject the Promise:

```
      // any other error
      if (err) {
        return reject(err);
      }
```

If we get a `Welcome` from CouchDB or PouchDB, we can safely assume that the database server is online:

```
      // maybe we got a welcome from CouchDB / PouchDB
      const isDatabase = (body.couchdb === 'Welcome' ||
        body['express-pouchdb'] === 'Welcome!');

      return resolve({[url]: isDatabase});
```

As a last step we have to export the function:

```
exports.api = isOnline;
```

Here is the whole function:

```
function isOnline (url) {
  return new Promise((resolve, reject) => {
    request({
      uri: url,
      json: true
    }, (err, res, body) => {

      // db is down
      if (err && (err.code === 'ECONNREFUSED' || err.code === 'ENOTFOUND')) {
        return resolve({[url]: false});
      }

      // any other error
      if (err) {
        return reject(err);
      }

      // maybe we got a welcome from CouchDB / PouchDB
      const isDatabase = (body.couchdb === 'Welcome' ||
        body['express-pouchdb'] === 'Welcome!');

      return resolve({[url]: isDatabase});
    });
  });
}
exports.api = isOnline;
```

We can try our function on the Node.js REPL:

```
$ node
> const isonline = require('./lib/isonline.js');
undefined
> isonline('http://example.com').then(console.log);
Promise { <pending> }
> { 'http://example.com': false }
> isonline('http://doesnotexist.example.com').then(console.log);
Promise { <pending> }
> { 'http://doesnotexist.example.com': false }
> isonline('http://localhost:5984').then(console.log);
Promise { <pending> }
> { 'http://localhost:5984': true }
```

Congratulations! We just finished the first part, the API part of our new command!