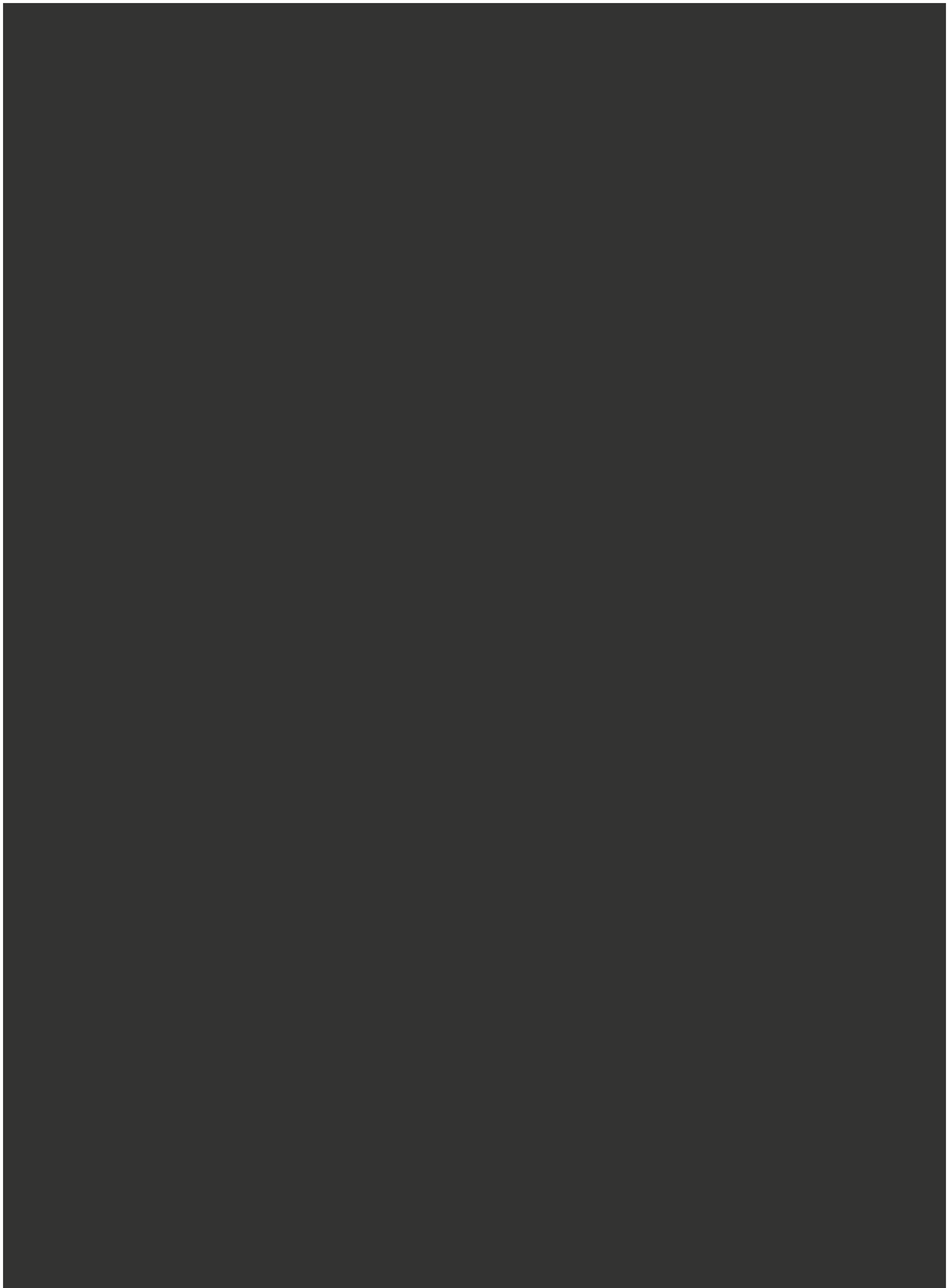
The CLI Book

Writing successful Command Line Clients with NODE.JS

ROBERT KOWALSKI



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.

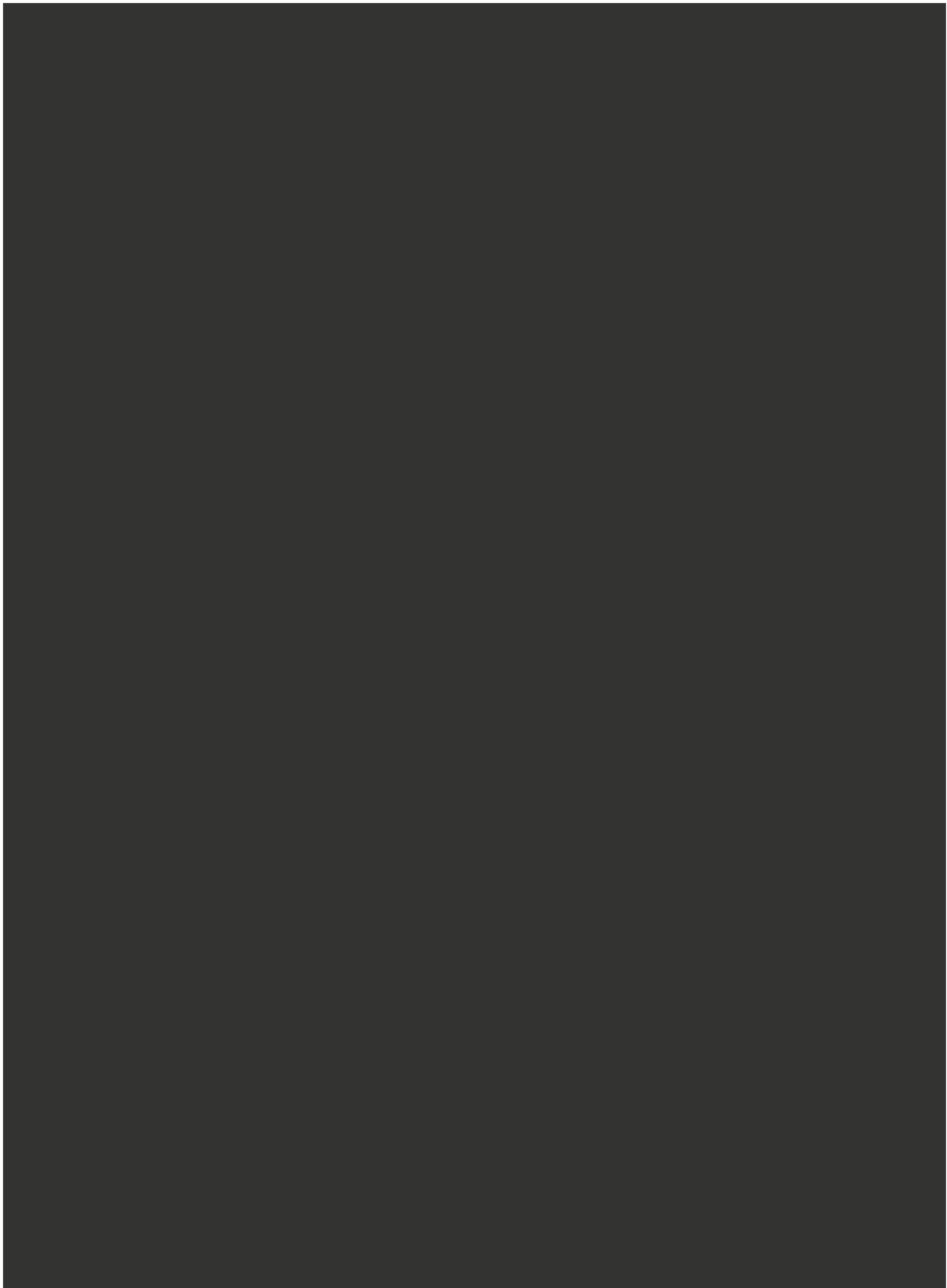
Every section has its own code examples. Before you run the code, you have to run `npm install` in the folder that belongs to the section.

You can download the code samples at http://theclibook.com/material/sourcecode.zip.

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:

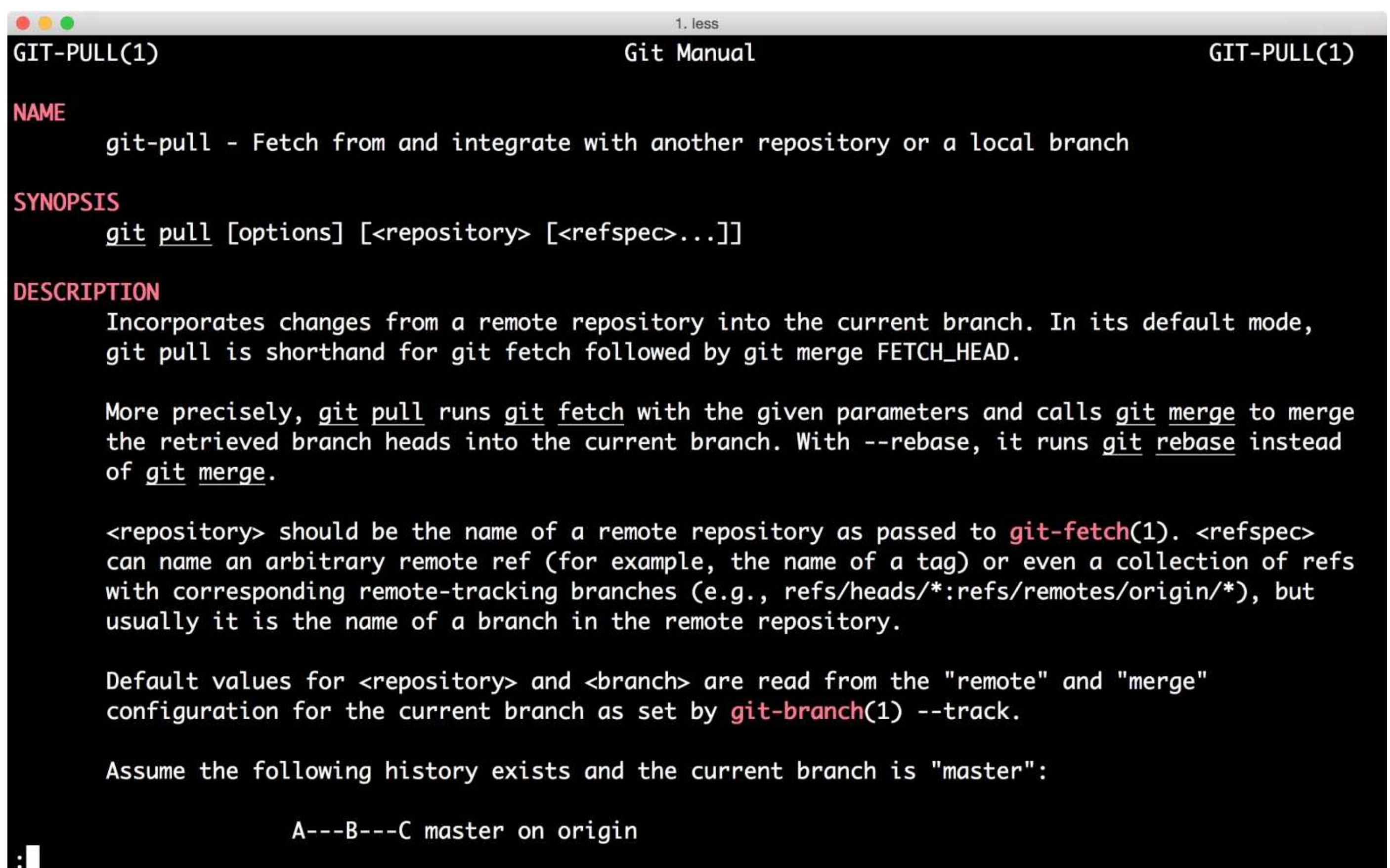
```
$ npm ragrragr
Usage: npm <command>

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!



```
1. less
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:

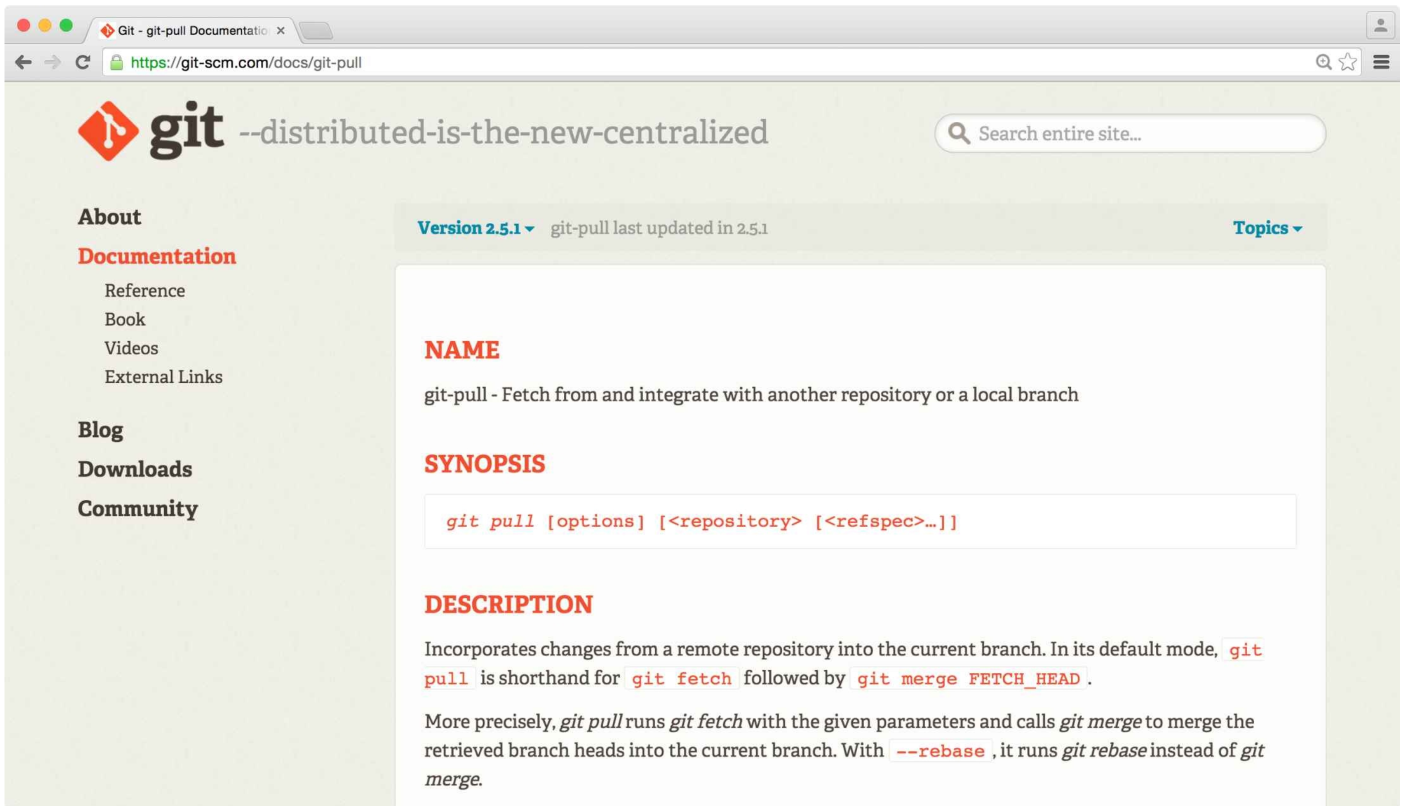


Figure 2. The documentation website of the git project

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

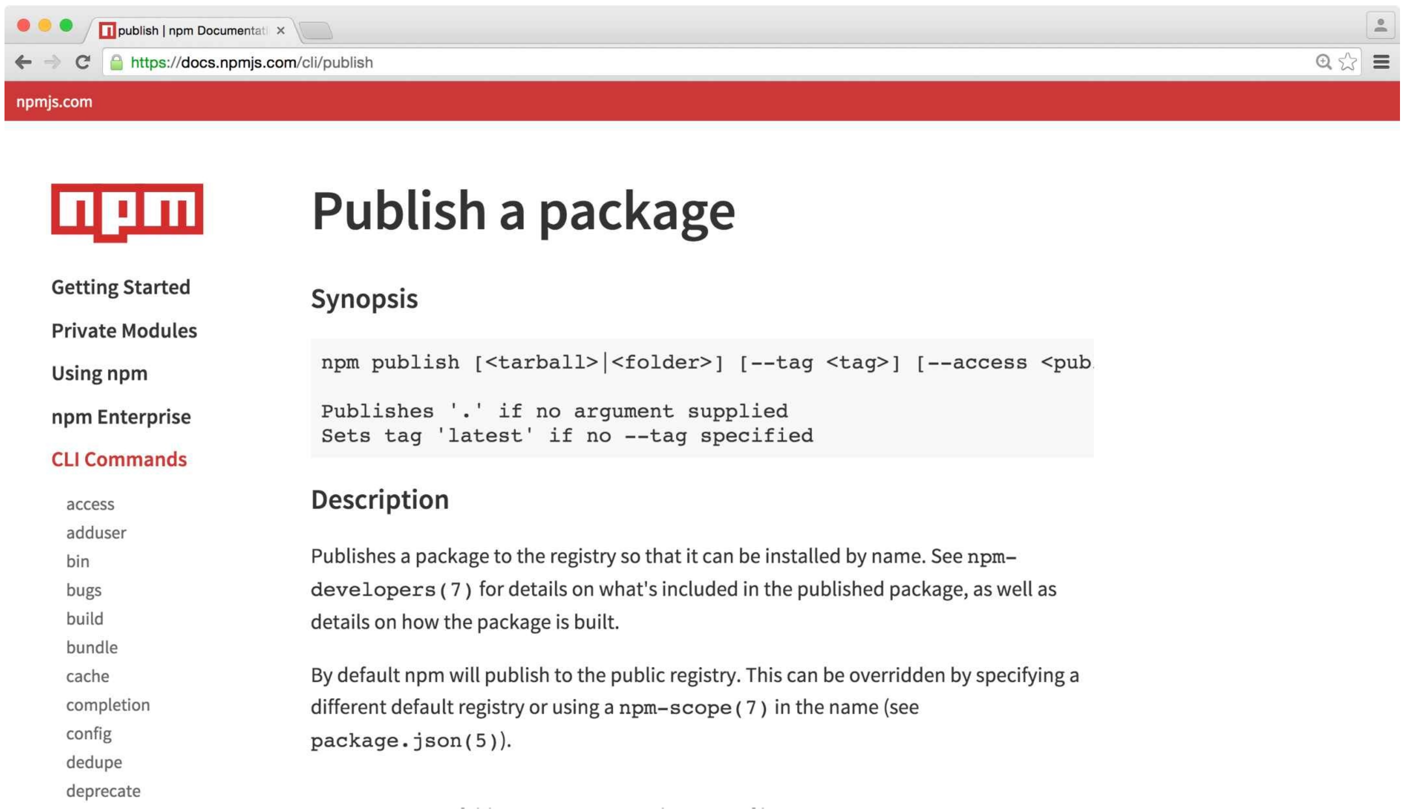


Figure 4. The documentation website of npm

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.

It supports powerusers

Powerusers are important for your CLI. They are users who will talk about your CLI and raise the overall adoption by spreading the word.

Shortcuts

Most power users are using your CLI multiple times every day. An easy way to support them is by providing shortcuts.

npm has lots of shortcuts. For instance, `npm i` is the short form for `npm install`. git lets you define your own shortcuts in the `.gitconfig` file. I use `git co` as a shortcut for `git checkout`, for example.

Scripting

At some point your command line client will get quite successful, people are loving it and start using your CLI in creative ways. The command line client will suddenly run on a Jenkins, as part of their deployment in a chef or Puppet run, or your users will use it in ways you never could have imagined!

Sooner or later not only humans will use your CLI, but also automated processes. To make your CLI even more successful it's a good idea to support scripting.

exit codes

Operating systems have different exit codes to signal if a command was successful. You will get back a `0` if your recent command was successful. `1` would be a general error.

Exit codes are very useful for users that want to wrap your command line client in a bash script.

Here is an example:

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

Did you mean this?
  pull
$ echo $?
1
```

git notifies me that something went wrong - I am getting back a `1` as exit code. With proper exit codes every writer of a bash script can handle the cases where a command is not successful.

JSON output

In nmo every command that gives back information supports json-formatted output:

```
$ nmo cluster get --json
{ anemone:
   { node1: 'http://node1.local',
     node2: 'http://node2.local',
     node3: 'http://node3.local' } }
```

JSON support enables user to process data easily in the programming language of their choice, as most languages support JSON. They spawn a child process in language x and

listen to `stdout` for the output. They can also directly pipe the output into a consumer on the shell:

```
$ nmo cluster get --json | consumer.py
```

JSON output gives a lot of flexibility to the users.

The API in the Command Line Client

There is another concept to make scripting easier: I call it the „The API in the Command Line Client“:

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

nmo.load({}).then(() => {
  nmo.commands.cluster
    .get('testcluster', 'node1@127.0.01')
    .then((res) => {
      console.log(res);
    });
});
```

In nmo every command is exposed on `nmo.commands`. If a user wants to use nmo as part of their node scripts, they are able to require it. The JavaScript API is documented like the CLI.

The JavaScript API enables the users to embed nmo in their Node.js scripts for complex processes. They could even fork nmo and embed it into their own command line client.

Configuration

Powerusers love configuration. Given they use a command line client a lot, maybe multiple times a day, it is no surprise that they would like to have some features enabled per default. But in rare cases, there is an exception and they don't need the default setting.

npm supports option arguments on the command line:

```
$ npm i hapi --registry=https://reg.kowalski.gd
/Users/robert
└── hapi@9.0.4
```

This command tries to download the package `hapi` from a private registry at `https://reg.kowalski.gd`.

But I can also set this private registry as the new default registry:

```
$ npm config set registry https://reg.kowalski.gd
```

npm writes the new registry into the config:

```
$ cat ~/.npmrc
loglevel=http
registry=https://reg.kowalski.gd
```

The next time I try to install a package, npm will use my new default registry, `https://reg.kowalski.gd`:

```
$ npm i hapi
```

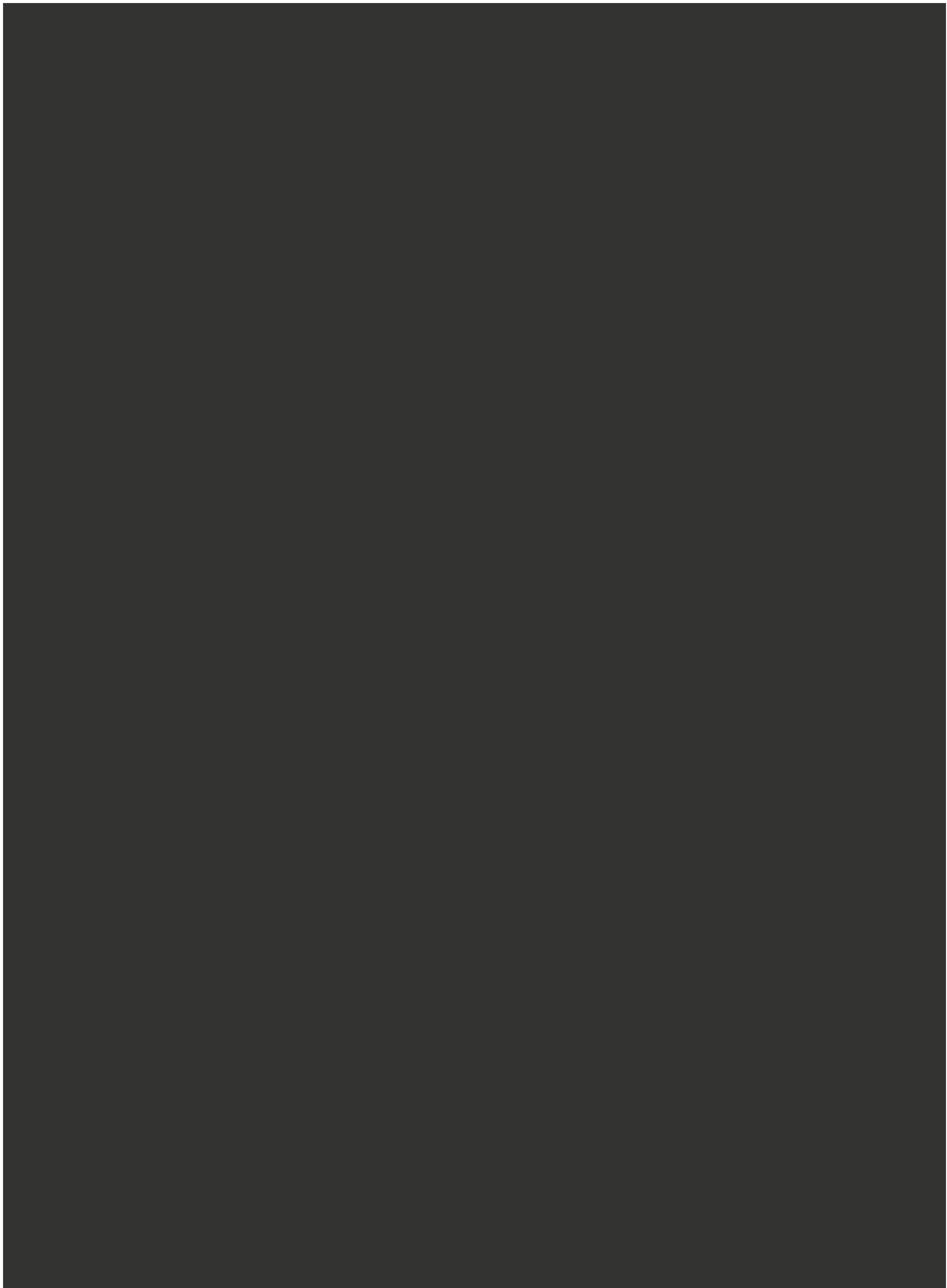
If I don't want to use this new default registry I can pass an argument to the CLI and it will use the alternate registry just for this call:

```
$ npm i hapi --registry=https://registry.npmjs.org
/Users/robert
└── hapi@9.0.4
```

That means we have different priorities between default configurations and command line arguments in npm and this combination is extremely powerful.

You can use it for all the things!

Let's take a look at the last principle, and the solution to it sounds very easy at first. Whenever I have to do a task multiple times and it fits into the domain of my command line client, I'll just add it as a new command. This habit turns into a win-win situation: You have to do less boring tasks and your users get happy because they get a new feature and they also have to do less monkey tasks - making your command line client even more successful. Sadly it can be quite hard to spot common pain points, especially if you work with multiple teams and/or a lot different people. Additionally most of us are suffering organisational blindness working on the same topic after a certain time. But if you identify a task to automate for you and your users, you will be hugely rewarded!



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.

```
1. bash
(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 1. 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.

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:

```
{
  "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!

The CLI part

It would be frustrating for the end user if that was the command line interface they have to use. The output is not easily readable and the functionality not easy to understand. The API function doesn't print to the console, which is perfect for an API, but not desirable for a CLI. It even requires some Node.js knowledge to run it. So let's add a nice CLI function to our file `isonline.js` and export it as `cli`:

```
function cli (url) {
  return new Promise((resolve, reject) => {

  });
}
exports.cli = cli;
```

The function returns a promise like all our other functions. We then call `isOnline` and print the result on `stdout` for the users that use the command line client on the terminal:

```
    isOnline(url).then((res) => {

      Object.keys(results).forEach((entry) => {
        let msg = 'seems to be offline';
        if (results[entry]) {
          msg = 'seems to be online';
        }

        // print on stdout for terminal users
        console.log(entry, msg);
        resolve(results);
      });
    });
```

```
function cli (url) {
  return new Promise((resolve, reject) => {

    isOnline(url).then((res) => {

      Object.keys(results).forEach((entry) => {
        let msg = 'seems to be offline';
        if (results[entry]) {
```

```
          msg = 'seems to be online';
        }

        // print on stdout for terminal users
        console.log(entry, msg);
        resolve(results);
      });
    });

  });
}
exports.cli = cli;
```

It is important to note: we export the command for the API as `exports.api`, while we export the CLI command under the `cli` property.

The code for this section can be found at `sourcecode/the-status-check`.

Booting the tool

Lounger still needs the code that makes it usable on the command line. It also lacks a comfortable way to run our API commands. Right now we just have one command, but we'll add more soon (see [_you_can_use_it_for_all_the_things]). To make all commands easy to use, we have to load all available commands into a namespace. The file `lib/lounger.js` will take care of it.

The file `lounger.js` is the heart of our command line client. We require the `fs` module as we have to list the files that could contain commmands:

```
'use strict';

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

We require the `package.json` and expose the current version of the module as a property on the lounger object, which will come in handy later. We also set `lounger.loaded` to `false`:

```
const pkg = require('../package.json');

const lounger = { loaded: false };
lounger.version = pkg.version;
```

We will need a place to store the API and the CLI commands. As the bootstrapping is async, we will throw an error if someone tries to access the exposed functions before the

bootstrap is finished:

```
const api = {}, cli = {};

Object.defineProperty(lounger, 'commands', {
  get: () => {
    if (lounger.loaded === false) {
      throw new Error('run lounger.load before');
    }
    return api;
  }
});

Object.defineProperty(lounger, 'cli', {
  get: () => {
    if (lounger.loaded === false) {
      throw new Error('run lounger.load before');
    }
    return cli;
  }
});
```

The custom getter for `Object.defineProperty` will throw if `lounger.loaded` is false and we try to access a property on `lounger.cli` and `lounger.api`.

It also works for the autocomplete in the Node.js REPL when you try to hit tab for completion. Just try it!

The last part of the file is the actual bootstrapping, where we get all files in `lib` and require them if they are JavaScript files and not `lounger.js`, the file we are currently working with.

The function to bootstrap lounger is called `lounger.load`. In this case we are using a named function. A named function can be very helpful in a stacktrace in case the app crashes:

```
lounger.load = function load () {
  return new Promise((resolve, reject) => {

  });
};
```

The function `fs.readdir` will list all files in a directory. We iterate over the list of files `fs.readdir` returns:

```
lounger.load = function load () {
  return new Promise((resolve, reject) => {
    fs.readdir(__dirname, (err, files) => {
      files.forEach((file) => {

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

If the file is not a JS file or is `lounger.js` itself, we ignore it by returning early:

```
        if (!/\.js$/.test(file) || file === 'lounger.js') {
          return;
        }
```

In all other cases we assume that we found a command for lounger. We take everything from the filename before the `.js` and save it as `cmd`:

```
const cmd = file.match(/(.*)\.js$/)[1];
```

We require the file:

```
const mod = require('./' + file);
```

If a file exports an API command as the `api` property, we expose it on `lounger.commands`. All CLI commands are available on `lounger.cli`:

```
        if (mod.cli) {
          cli[cmd] = mod.cli;
        }

        if (mod.api) {
          api[cmd] = mod.api;
        }
```

After the `forEach` loop is finished and all commands are loaded we can set `lounger.loaded` to `true`. This will prevent the checks we added previously from throwing:

```
      lounger.loaded = true;
```

As last step we resolve the Promise:

```
resolve(lounger);
```

The whole function `lounger.load`:

```
lounger.load = function load () {
  return new Promise((resolve, reject) => {
    fs.readdir(__dirname, (err, files) => {
      files.forEach((file) => {
        if (!/\.js$/.test(file) || file === 'lounger.js') {
          return;
        }

        const cmd = file.match(/(.*)\.js$/)[1];
        const mod = require('./' + file);
        if (mod.cli) {
          cli[cmd] = mod.cli;
        }

        if (mod.api) {
          api[cmd] = mod.api;
        }

      });
      lounger.loaded = true;
      resolve(lounger);
    });
  });
};
```

We almost forgot to export `lounger`, so we add a `module.exports` at the end of the file:

```
module.exports = lounger;
```

We can already use the code:

```
$ node
> const lounger = require('./lib/lounger.js');
lounger.load().then(console.log);
Promise { <pending> }
> { loaded: true, version: '1.0.0', load: [Function: load] }
> lounger.commands
{ isonline: [Function: isOnline] }
> lounger.cli
{ isonline: [Function: cli] }
```

```
> lounger.commands.isonline('http://localhost:5984').then(console.log);
Promise { <pending> }
> { 'http://localhost:5984': true }
```

The last step in this section is to make lounger usable on the terminal itself. For this task we've created the `bin` folder already. Time to put a file called `lounger-cli.js` into `bin`!

npm has a nice feature: if we add a JavaScript file to a property called `bin` in the package.json of a module, npm will add it to our `PATH` if we are installing the package globally (e.g. with `npm install -g lounger`).

If we have the bin-property defined and the lounger is globally installed, it gets available as `lounger` (which is our package name) on the terminal, quite comfortable! To inform the user that a package is intended to get installed globally, we can add `preferGlobal: true` to the `package.json` (see also: https://docs.npmjs.com/files/package.json#preferglobal).

To enable the two features we add these two lines to our `package.json`:

```
"bin": "./bin/lounger-cli",
"preferGlobal": true,
```

Additionally we have to make `bin/lounger-cli` executable, if we are on Linux or OSX:

```
$ chmod +x bin/lounger-cli
```

The first line of our `lounger-cli` file will be a "shebang line", it tells Linxux/Unix shell users that of Linux/Unix users that it must run our file with Node.js:

```
#!/usr/bin/env node
```

Afterwards we load `lib/lounger.js`, the core of our command line tool:

```
const lounger = require('../lib/lounger.js');
```

The next step is to parse our command line arguments. In this case we are using the module `nopt` to parse the command line arguments (install it with `$ npm i --save nopt`). It would be possible to try to parse the arguments on our own, but a battle proven

module like `nopt` offers a lot more features and is easier to use. We get the passed command by accessing `parsed.argv.remain`:

```
const nopt = require('nopt');
const parsed = nopt({}, {}, process.argv, 2);

const cmd = parsed.argv.remain.shift();
```

The next step is to boot the client by calling `lounger.load`, which will bootstrap the client and populate `lounger.commands` and `lounger.cli`. After the promise got resolved, we call the command that was passed on the command line:

```
lounger.load().then(() => {

  lounger.cli[cmd]
    .apply(null, parsed.argv.remain)
    .catch((err) => {
      console.error(err);
    });

}).catch((err) => {
  console.error(err);
});
```

As every command returns a Promise, we catch errors with `catch` and print them to the console.

We can now test our minimalistic command line client on the command line:

```
$ npm install -g .
$ lounger isonline http://couchdb.example.com
http://couchdb.example.com seems to be offline or no database
```

And given our PouchDB/CouchDB server is running:

```
$ lounger isonline http://localhost:5984
http://localhost:5984 seems to be online
```

Instead of running `npm install -g .` after each code change, you can also run `npm link` in your module directory. It will link the global installation to the current directory, which means that every change is immediately available, as long as the directory does not change.

Choose your own flavours

There are countless good argument parsing libraries on npm, some alternatives to `nopt` are: `commander`, `optimist` and `yargs`.

That was the first basic building block for our command line client, but we are still far away from a product that our users would really love and promote. We will fix those issues (error handling, help, documentation) in the next sections. The code for the current section is available at `sourcecode/client-bootstrap`.

Error handling

We already learned in [_you_never_get_stuck] that no user enjoys cryptic error messages and stack traces. Sadly that is still the case for our `lounger` application and making the application more accessible should be our first priority.

Right now lounger does not do anything about wrong input for the `isonline` command:

```
$ lounger isonline ragrragr
$
```

Another caveat to consider is, are we handling errors correctly so people can use the command line client in their bash scripts?

```
$ lounger isonline ragrragr
$ echo $?
0
```

Wouldn't it be nicer if the users would get a hint about the correct usage of the program right away, without opening any documentation? Nobody enjoys sitting in front of a black terminal having no idea what to do (see [_you_never_get_stuck]). While we are at it we can also fix the wrong exit code which is currently signalling a successful executed program (see also [_it_supports_powerusers_exit_codes]).

Why isn't our `console.error` call in `bin/lounger-cli` printing anything? Turns out we introduced a subtle bug: we forgot the `.catch` for our Promise returning call in `lib/isonline.js`. Given the API function `isOnline` rejects the Promise, we have no handler in the function `cli` to take care of it. No problem, we'll add the `.catch` right now:

```
function cli (url) {
  return new Promise((resolve, reject) => {

    isOnline(url).then((results) => {

      // print on stdout for terminal users
      Object.keys(results).forEach((entry) => {
        let msg = 'seems to be offline or no database server';
        if (results[entry]) {
          msg = 'seems to be online';
        }

        console.log(entry, msg);
        resolve(results);
      });
    }).catch(reject);  // add the missing catch
  });
}
exports.cli = cli;
```

Our next try is a bit more successful:

```
$ lounger isonline ragrragr
[Error: Invalid URI "ragrragr"]
```

Not sure if you are happy with it… I'm not! Just imagine someone that has never used Node.js or the Terminal. Maybe even someone that is completely new to computers. `Invalid URI` won't help them much to get their task done. Twenty years ago they would have had to get a book from the library in order to find out what an URI is, and today they would have to google for it. Useful time they could have fun with our CLI and get things done!

Gladly we can fix it by adding validations for the arguments in `isonline.js`.

If the user does not provide a url to the CLI we are creating a new error with the message `Usage: lounger isonline <url>` which describes how they have to use the command. We are setting the type of the error to `EUSAGE`, which will be important later. In lounger all errors that are thrown because the user made wrong input are getting the type `EUSAGE`. All other cases where we introduced bugs don't get the type `EUSAGE`:

```
function cli (url) {
  return new Promise((resolve, reject) => {

    if (!url) {
```

```
      const err = new Error('Usage: lounger isonline <url>');
      err.type = 'EUSAGE';
      return reject(err);
    }
```

The less-than sign and greater-than sign around the url indicate that `url` is a required argument and not optional. The last command in the block rejects the Promise with our error and returns, in order to prevent the execution of following code.

> Early returns are useful to reduce cyclomatic complexity. Cyclomatic complexity appears where if and else blocks are nested, which makes it harder for the human brain to reason about the flow of the program execution.

Additionally we have to check if the url is a valid url:

```
    if (!/^(http:|https:)/.test(url)) {
      const err = new Error([
        'invalid protocol, must be https or http',
        'Usage: lounger isonline <url>'
      ].join('\n'));
      err.type = 'EUSAGE';
      return reject(err);
    }
```

In this case we are setting the error type to `EUSAGE` again and reject the Promise. Additionally we are telling the user that we expect a valid url with a protocol that is usable for us.

On our next try we will get a slightly better result:

```
$ ./bin/lounger-cli isonline dsf
{ [Error: invalid protocol, must be https or http
Usage: lounger isonline <url>] type: 'EUSAGE' }
```

As we reject the promise, the `console.error` that we added in in `bin/lounger-cli` prints the error object. By adding a few lines of code we can format it, so humans can read it better. We will install the `npmlog` logger for it (hint: `npm i` is short for `npm install`):

```
$ npm i --save npmlog
```

We require it at the top of `bin/lounger-cli`, the file where we catch the rejected Promise:

```
const log = require('npmlog');
```

As a next step we add the function `errorHandler` to `bin/lounger-cli`. If the error is a usage error (of type `EUSAGE`), we log the message and exit with error code `1`. All other errors are logged using `log.error(err)` for now:

```
function errorHandler (err) {
  if (!err) {
    process.exit(1);
  }

  if (err.type === 'EUSAGE') {
    err.message && log.error(err.message);
    process.exit(1);
  }

  log.error(err);
  process.exit(1);
}
```

Now we have to switch from the old `console.error` call to our new error handling function:

```
lounger.load().then(() => {

  lounger.cli[cmd]
    .apply(null, parsed.argv.remain)
    .catch(errorHandler);

}).catch(errorHandler);
```

Cool, let’s see if it works:

```
$ lounger isonline
ERR! Usage: lounger isonline <url>
$ echo $?
1
```

That looks a lot better!

```
1. bash
(22:40:15) [robert@tequila-new] ~/sourcecode/lounger $ lounger isonline ragrragr
ERR! invalid protocol, must be https or http
Usage: lounger isonline <url>
(22:40:29) [robert@tequila-new] ~/sourcecode/lounger $ echo $?
1
(22:40:34) [robert@tequila-new] ~/sourcecode/lounger $
```

Figure 2. A usage error resulting from providing wrong input to the CLI

We still did not touch other errors: errors from dependencies we use, or evil bugs that sneak in, like reference errors. To simulate such an error we can add a call to a non-existing function in the cli function:

```
function cli (url) {
  return new Promise((resolve, reject) => {
    doesNotExist();
```

If we now run the command line client we get:

```
$ lounger isonline http://example.com
ERR! ReferenceError: doesNotExist is not defined
```

If I just downloaded the command line client and tried to use it, I would be quite puzzled. Maybe I got a new job and tried to use the same tool my coworkers use, but downloaded a newer release with bugs. I would be stuck, with no further notice how to continue. To be honest, if I hadn't programmed in JavaScript for years this stacktrace would really puzzle me! For most people these are rocks in their way where they just stop using our program and switch to an alternative. Very few go on the journey to find out where they can submit an issue or even write a PR. Usually computers are too frustrating and people don't really want to spend multiple hours trying to find someone to help them with a cryptic message. So what about making this process as easy as possible, reducing the friction where we can?

npm itself supports a `bugs` property in the `package.json`. If we add:

```
  "bugs": {
    "url": "http://example.com/lounger/issues"
  },
```

to the `package.json` of lounger, a call to `$ npm bugs` will open `http://example.com/lounger/issues` in a browser for us. Cool, we got a central place where we are storing the url to our issue tracker. We can also add the url to our stack traces, in order to make submitting bugs for our users easier. We need to require the `package.json` in `bin/lounger-cli`, the file where we print our errors anyway:

```
const pkg = require('./package.json');
```

By altering our `errorHandler` we make it print full stack traces. Additionally we add ask the user to open an issue as it is pretty clear right now that the error was not a usage error that was caught by our validations:

```
function errorHandler (err) {
  if (!err) {
    process.exit(1);
  }

  if (err.type === 'EUSAGE') {
    err.message && log.error(err.message);
    process.exit(1);
  }

  err.message && log.error(err.message);

  if (err.stack) {
    log.error('', err.stack);
    log.error('', '');
    log.error('', '');
    log.error('', 'lounger:', pkg.version, 'node:', process.version);
    log.error('', 'please open an issue including this log on ' +
pkg.bugs.url);
  }
  process.exit(1);
}
```

Ok, next try:

```
$ lounger isonline http://example.com
ERR! doesNotExist is not defined
ERR! ReferenceError: doesNotExist is not defined
ERR!     at /home/rocko/clibook/sourcecode/error-
handling/lib/isonline.js:35:7
ERR!     at cli (/home/rocko/clibook/sourcecode/error-
handling/lib/isonline.js:34:10)
ERR!     at /home/rocko/clibook/sourcecode/error-handling/bin/lounger-
cli:15:6
ERR!
ERR!
ERR! lounger: 1.0.0 node: v4.2.3
ERR! please open an issue including this log on
http://example.com/lounger/issues
```

Awesome! The stacktrace with line numbers is useful for us. The current version of the program and the Node.js environment help us, too. In case the command line client really hits a wall, we receive a lot of informations in order to debug the process. Even more important: the user get's all the information needed to create an issue. We remove a lot of friction from the process by directly pointing to the issue tracker and providing all information that is needed to describe the bug - no long back and forth about the current Node version or the missing logfile!

The code for this section can be found at `sourcecode/error-handling`.

JSON support and Shorthands

JSON support is useful for all users that want to take the output from the CLI and process it programmatically with their own tools (we talked about that in [_it_supports_powerusers_json] in the first part of the book). By adding a `--json` flag to our command `isonline` we can add this useful feature with a few lines of code. We have to tell our argument parser about it, in this case, we are telling `nopt` that we want to have `--json` handled as a boolean in `bin/lounger-cli`:

```
const parsed = nopt({
  'json': [Boolean]
}, {'j': '--json'}, process.argv, 2);
```

Based on the type `Boolean` `nopt` will automatically also add `--no-json` for us, which will come handy when we add additional configuration by file later. Additionally we register a shorthand for our power users, they can also use -j instead of —json.

We are then passing the result `parsed` into `lounger.load`:

```
const parsed = nopt({
  'json': [Boolean]
}, {'j': '--json'}, process.argv, 2);

const cmd = parsed.argv.remain.shift();

lounger.load(parsed).then(() => {

  lounger.cli[cmd]
    .apply(null, parsed.argv.remain)
    .catch(errorHandler);

}).catch(errorHandler);
```

`longer.load` adds a `lounger.config.get` command and makes it available for us as part of the bootstrap:

```
lounger.load = function load (opts) {
  return new Promise((resolve, reject) => {

    lounger.config = {
      get: (key) => {
        return opts[key];
      }
    };

    fs.readdir(__dirname, (err, files) => {
```

We require `lounger.js` in our file `isonline.js`:

```
const lounger = require('./lounger.js');
```

As last step we the check for the json-flag in our function `cli` after we got the results back:

```
    isOnline(url).then((results) => {

      if (lounger.config.get('json')) {
        console.log(results);
        resolve(results);
        return;
      }
```

That’s it! We can test the command:

```
$ lounger isonline http://example.com
http://example.com seems to be offline or no database server

$ lounger isonline http://example.com --json
{ 'http://example.com': false }

$ lounger isonline http://example.com -j
{ 'http://example.com': false }
```

Our users can now pipe the output on their terminals into other consumers and process the results. We also added our first command line flag to lounger to modify the execution of a command - great! The code and tests for this section are in `sourcecode/json-flags`.

Documentation

The last step to finish our command `isonline` is to add proper documentation. We will write the documentation in Markdown and need documentation for the API and CLI commands. The API docs will live in `doc/api` and the CLI commands will live in `doc/cli`. I know that most programmers hate writing documentation, but it will help us a lot: new users will be able to get up and running easier and we won't lose them before they had the chance to enjoy our product. Additionally, we make our lives easier by documenting the functionality once, so people don't have to open issues or ask in chat how they can use a command. Basically a win-win situation. We start with `doc/api/lounger-isonline.md`, which describes the API that is available at `lounger.commands`:

```
lounger-isonline(3)—check if a database is online
====================================================
```

The heading describes our command as `lounger-isonline(3)` and then adds a short explanation what the command is about. The number in brackets describes the type of the section. For a man-page a Library Function is noted by a `3` and a User Command would be a `1` (spoiler: our CLI command is a User Command).

The next section describes how our users can use the command:

```
## SYNOPSIS

    lounger.commands.isonline(url)
```

The last part is a detailed description of how the command works:

```
## DESCRIPTION

Check if a CouchDB / PouchDB database is available on the current
network.

url:
The url must be a `String` and must be a url using the http or https
protocol.

The command returns a promise. The promise returns an Object. The key
of the Object is the provided url and the values are of type `Boolean`.
`true` indicates an online CouchDB / PouchDB node.
```

That’s it for the API part, we can now add the text for `doc/cli/lounger-isonline.md`:

```
lounger-isonline(1)—check if a database is online
====================================================

    lounger isonline <url> [--json]

## DESCRIPTION

  <url>:
Check if a database node is currently online or available.

`isonline` prints the result as human readable output. JSON output is
also supported by passing the `--json` flag.
```

With the small `1` in `lounger-isonline(1)` we are signalling that this help section explains a User Command. The less-than and greater-than symbols for `<url>` show the user that `url` is a mandatory argument - without it the command won’t work. The square brackets of `[--json]` mean that the json-flag is an **optional** command.

As we got the sources for our documentation, we can start to build our documentation from our sources with `marked` and `marked-man`:

```
$ npm i --save-dev marked marked-man
```

A Makefile would be a great fit for generating the documentation from the source, but sadly it is hard to get Makefiles to work on Windows, so we will write our build steps in JavaScript. In the root directory of lounger, we create the file `build.js`. Additionally,

we have to install `mkdirp` and `rimraf`, `mkdirp` provides the functionality we know from the Linux command `mkdir -p` in a cross-platform compatible way: it creates a directories and subdirectories in a recursive way. The module `rimraf` brings us the equivalent of `rm -rf` to the Node.js platform: deleting directories in a recursive way. Additionally we will use the module `glob` to match all needed files for our documentation build.

```
$ npm i --save-dev mkdirp rimraf glob
```

Our first function will be a function to clean a fresh folder structure where we can save our man-pages:

```
'use strict';

const mkdirp = require('mkdirp');
const rimraf = require('rimraf');
const glob = require('glob');
const path = require('path');


function cleanUpMan () {
  rimraf.sync(__dirname + '/man/');
  // recreate the target directory
  mkdirp.sync(__dirname + '/man/');
}
```

We have to find out which markdown files are available for the compile. Our sources for documentation are at `doc/api` or at `doc/cli`. Additionally we will have some content in `doc/website` which is specific to the website, i.e. the content for the `index.html`.

The function `getSources` helps us to get the full path to the markdown files for each type of our sources (`api`, `doc`, `website`). It returns the relative path of the matching glob and then uses the function `path.resolve` to get the full path in a cross-platform compatible way.

```
function getSources (type) {
  const files = glob.sync('doc/' + type + '/*.md');

  return files.map(file => path.resolve(file));
}
```

The object `sources` stores an array of the found files for each type:

```
const sources = {
  api: getSources('api'),
  cli: getSources('cli'),
  websiteIndex: getSources('website'),
};
```

We are able to clean up our target directory now and to get a list of filenames that we want to convert. We still need to find out the target path and filename for the converted files. Man-pages have different file endings depending of the kind of functionality they describe. Our man-pages for the CLI would get the ending `.1` (User Commands) and our API function would get the ending `.3` (Library Functions). Additionally we have to change the `/doc/cli/` and `/doc/api/` in the path of the file to our target directory `/man/`. We have to take special care of the path-separators. On Windows the separators are a `\\`, instead of `/`. That means the path `doc/api` gets `doc\\api` on Windows. The good news is that we can access the current path- separator using `path.sep` in Node.js (the separator is provided by the core module `path`):

```
function getTargetForManpages (currentFile, type) {
  let target;
  // set the right section for the man page on unix systems

  if (type === 'cli') {
    target = currentFile.replace(/\.md$/, '.1');
  }

  if (type === 'api') {
    target = currentFile.replace(/\.md$/, '.3');
  }

  // replace the source dir with the target dir
  // do it for the windows path (doc\\api) and the unix path (doc/api)
  target = target
    .replace(['doc', 'cli'].join(path.sep), 'man')
    .replace(['doc', 'api'].join(path.sep), 'man');

  return target;
}
```

Right now we just want to create man-pages from our documents in the `api` and `cli` folder.

Based on these building blocks we can create the final function `buildMan` that will finally build our man-pages. It will make use of the functions we just created and

additionally spawn a child process which compiles the markdown files using `marked-man`. We will use the function `spawnSync` from Node core to spawn the processes. As we write the result to the filesystem, we have to require the `fs` module, too:

```
const fs = require('fs');
const spawnSync = require('child_process').spawnSync;
```

The first job is cleaning up to get a new target directory without any files from previous builds. We then iterate over our sources and get the target for our new generated file. The file is then written to the hard disk using `fs.writeFileSync`. In case of the website index we stop the execution as we don't want to do use the website index for a man-page right now. In the next iteration we could definitely add a main page for the lounger man-pages:

```
function buildMan () {
  cleanUpMan();

  Object.keys(sources).forEach(type => {

    sources[type].forEach(currentFile => {

      if (type === 'websiteIndex') {
        return;
      }

      // convert markdown to man-pages
      const out = spawnSync('node', [
        './node_modules/marked-man/bin/marked-man',
        currentFile
      ]);

      const target = getTargetForManpages(currentFile, type);

      // write output to target file
      fs.writeFileSync(target, out.stdout, 'utf8');
    })
  });
}
buildMan();
```

With `buildMan();` in the last line, we kick off the build process every time we run the script with Node. A few modifications to our `package.json` could make it a npm script and run the build before a publish, so our users don't have to compile anything on their

own as part of the installation. This ensures that every user really gets the same content of the package and makes installations faster.

In `package.json` we modify the `scripts` section and add entries for `build` and `prepublish`. The `prepublish` entry is a special hook for npm – it will run every time before we publish the package to the registry:

```
"scripts": {
  "test": "mocha -R spec",
  "docs": "node ./build",
  "prepublish": "npm run docs"
},
```

npm also offers a nice feature for man-pages: npm can install them for the user so they are available on the terminal with `man <command>` for Linux/Unix users. In order to do that, we have to add another entry to our `package.json`, the `man` entry in the `directory` section:

```
"directories": {
  "man": "./man"
},
```

The entry points to our local man-pages directory. If you are on Unix/Linux you can try if the man-pages work now:

```
$ npm run docs

> lounger@1.0.0 docs /home/rocko/clibook/sourcecode/documentation
> node ./build

$ npm install -g .


> lounger@1.0.0 prepublish /home/rocko/clibook/sourcecode/documentation
> npm run docs


> lounger@1.0.0 docs /home/rocko/clibook/sourcecode/documentation
> node ./build

/home/rocko/.nvm/versions/node/v4.2.3/bin/lounger ->
/home/rocko/.nvm/versions/node/v4.2.3/lib/node_modules/lounger/bin/lounger-
cli
/home/rocko/.nvm/versions/node/v4.2.3/lib
```

```
└── lounger@1.0.0

$ man lounger-isonline
```

```
LOUNGER-ISONLINE(1)                                              LOUNGER-ISONLINE(1)

NAME
       lounger-isonline - check if a database is online

         lounger isonline <url> [--json]

DESCRIPTION
       Check if a database is online and available.

       url: The url must be an url that is using the http or https protocol.

       isonline prints the result as human readable output. JSON output is also supported by passing the --json flag.

                              February 2016                      LOUNGER-ISONLINE(1)
~
~
~
~
~
~
~
~
~
(END)
```

Figure 3. The man-page for lounger, our command line client

It is also possible to select a specific section:

```
$ man 3 lounger-isonline
$ man 1 lounger-isonline
```

For our html-based documentation we have to add the html specific part now. We have create a folder called `website` in the doc folder of our module. It will contain the templates for the website that are used to “frame“ the document output.

Into the folder called `website` we put a file called `template.html` with some basic markup, and most importantly, placeholders!

```html
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>The lounger manual &amp; documentation</title>
</head>
<body>

  <div class="wrapper">
    <a href="./">lounger</a>

    <div class="content">
```

```
      __CONTENT__
    </div>
  </div>
  <nav class="toc-container">
    <div class="toc main-toc">
      __TOC__
    </div>
  </nav>

</body>
</html>
```

We need to create some content for the root of our website to welcome the user. I will just provide a very short example. In general the landing page should give the user an idea what the command line client is about and maybe also demo it. A screencast is a great way to demo some of the core features. For some inspiration what to put on the site, feel free to visit http://apache.github.io/couchdb-nmo, which is the page for the command line client I wrote last year. Next to our template in `doc/website` we add the bespoke index page as `index.md`:

````
# Welcome to Lounger


Lounger is a friendly administration tool for CouchDB and PouchDB.

```
# you will need Node.js > 4 for lounger
npm install -g lounger
```


Things you can do with lounger:

```
# check if a CouchDB / PouchDB instance is online

lounger isonline http://example.com
```
````

The page is minimalistic but gives a brief overview what lounger is about. It also shows how our visitors can install it and gives a hint that they need Node.js – not everyone has Node.js installed and some people may have never heard of npm. Remember: our goal is to make everything as easy as possible for new users – they never get stuck.

In order to build the website we have to add some code to the `build.js`. We start by adding another cleanup-function for our website files, so we can start with a blank slate every time:

```
function cleanUpWebsite () {
  rimraf.sync(__dirname + '/website/');
  mkdirp.sync(__dirname + '/website/');
}
```

Our website has different targets than the man-pages. We add a new function, `getTargetForWebsite`. Currently our markdown files are prefixed with `lounger-`, which comes handy for the man-pages, but is not very useful for our website. Instead we want to prefix them with their type, an API document would get prefixed by `api-`. This way we can put pages for the API next to the ones for the CLI, which makes linking easier. The first lines of the function `getTargetForWebsite` will take care of that task and replace the `lounger-` prefix with a prefix specific to the type of the document. After we replaced the prefix, we set the file ending from `.md` to `.html`. The final target folder for our compiled results will be `./website`, so we have to take care that the directories are set up right. The difference to `getTargetForManpages` is that we use the `website` folder instead of `man` and that we also take care of the path for the `doc/website/index.md` file:

```
function getTargetForWebsite (currentFile, type) {
  let target = currentFile;
  // modifiy the filename a bit for our html file:
  // prefix all cli functions with cli- instead of lounger-
  // prefix all api functions with api- instead of lounger-
  if (type === 'cli') {
    target = currentFile.replace(/lounger-/, 'cli-');
  }

  if (type === 'api') {
    target = currentFile.replace(/lounger-/, 'api-');
  }

  // set the file ending to html
  target = target.replace(/\.md$/, '.html');

  // replace the source dir with the target dir
  target = target
    .replace(['doc', 'cli'].join(path.sep), 'website')
    .replace(['doc', 'api'].join(path.sep), 'website')
```

```
    .replace(['doc', 'website'].join(path.sep), 'website');

  return target;
}
```

The last item which is missing for our website is something like a table of contents. `getTocForWebsite` will create a listing of our API and CLI functions. Later we will insert the table of contents into the *`TOC`* placeholder of the template.

The TOC itself gets a nested list of the files we compile. In the future it might make sense to replace the whole system with a template engine like `handlebars`, `jade` or `Nunjucks`. I could definitely write a second book about static website generation plus the different possible toolchains, but I try to keep it simple and minimalistic for now and just use plain ES6 templates.

After starting an unordered list with `<ul>` we iterate over the types of our sources again. The files of our `websiteIndex` type are unwanted and we return early in their case. For all others we take the type of each section and use it as the first list element. It shows the type of each section (API or CLI) and acts as a heading.

After we got a heading, we iterate over each file from the current section. As the linktext differs a bit from the actual hyperlink, we create a constant called `file` and an additional `linktext` constant. For both we must get rid of the `lounger-` prefix. To create the reference that is used as `href` we must change the `.md` to a `.html` ending. The link text should not have a file ending at all, so we remove the `.md` ending for it:

```
function getTocForWebsite () {
  let toc = '<ul>';

  Object.keys(sources).forEach(type => {

    // we don't want the index in our toc for now
    if (type === 'websiteIndex') {
      return;
    }

    toc += `<li><span>${type}</span><ul>`;

    sources[type].forEach(currentFile => {
      const prefix = type === 'cli' ? 'cli-' : 'api-';

      const file = path.basename(currentFile)
        .replace('lounger-', prefix)
```

```
        .replace(/\.md/, '.html');

      const linktext = path.basename(currentFile)
        .replace('lounger-', '')
        .replace(/\.md/, '');

      toc += `<li><a href="${file}">${linktext}</a></li>`;
    });
    toc += '</ul></li>';
  });

  toc += '</ul>';

  return toc;
}
```

We can now take the small functions we created and create the main build function `buildWebsite` from them. The constant `templateFile` describes where we can find our template file `template.html` which we created in the beginning:

```
const templateFile = __dirname + '/doc/website/template.html';
```

The next step is to replace the placeholders in the templates with the generated table of contents and the different content for each API or CLI method.

In `buildWebsite` we call `cleanUpWebsite` to remove any outdated files as a first step. We read the template with `fs.readFileSync` and get the table of contents. We then iterate over our sources. This time we don't spawn a child process with `marked-man`, we just use `marked`, which outputs html. After we got the target for the current file of the website we replace the *`CONTENT`* placeholder with it. The *`TOC`* gets replaced with the table of contents which is saved in `toc` at the top of the main function. Moving the read operation of the template and the creation of the TOC out of the loops has positive effects on the performance of our build, as we don't have to call them for every new file. The rendered content finally gets written to the disk using `fs.writeFileSync` again. The last line in the code snippet finally calls `buildWebsite` in order to build our website when we run `build.js`.

```
function buildWebsite () {
  cleanUpWebsite();

  const template = fs.readFileSync(templateFile, 'utf8');
  const toc = getTocForWebsite();
```

```
  Object.keys(sources).forEach(type => {

    sources[type].forEach(currentFile => {

        // convert markdown to website content
      const out = spawnSync('node', [
        './node_modules/marked/bin/marked',
        currentFile
      ]);

      const target = getTargetForWebsite(currentFile, type);

      const rendered = template
        .replace('__CONTENT__', out.stdout)
        .replace('__TOC__', toc);

      // write output to target file
      fs.writeFileSync(target, rendered, 'utf8');
    });
  });
}

buildWebsite();
```

We can now run our build again and take a look at our website, which appears in the `website` folder:

```
$ npm run docs
```

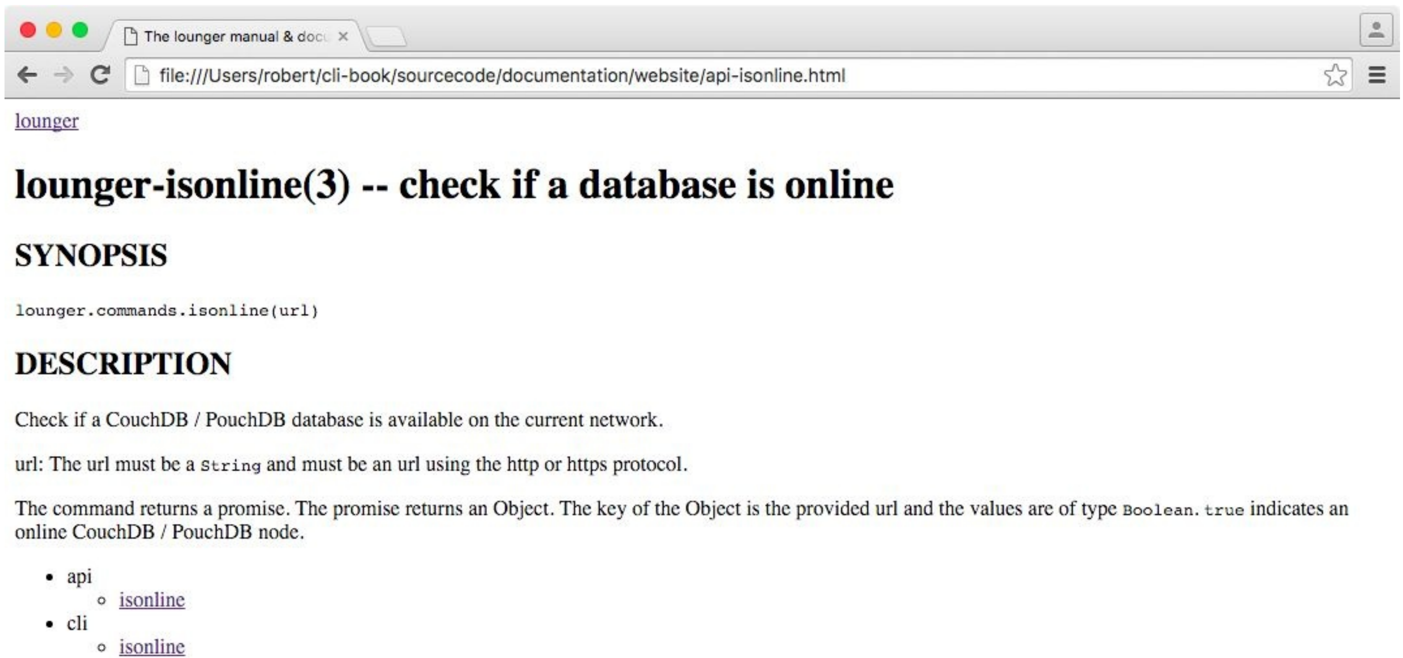


Figure 4. The minimal website for lounger generated from our markdown

The site still misses some content and also a nice stylesheet. As I already mentioned, building websites is a whole topic for a new book and I want to leave it like this for now. Feel free to add more content and styles to the website. Nevertheless we just created something that we can deploy to GitHub pages or any other hoster. The best thing about it is that we can ship it with our command line client as additional documentation – for everyone who can't use or doesn't like man-pages. We will use both types of documentation in our next section when we create a help system.

The code for this section can be found at `sourcecode/documentation`.

More Help

We got some documentation, but there are still some rough edges in lounger. If we try to access a command which does not exist, the user doesn't get any advice how to continue:

```
$ lounger foobar
```

If a user calls lounger with no arguments at all they also don't get support:

```
$ lounger
```

What's missing is a help page which explains how our users can get their task done. Additionally it would be really awesome if they could open our documentation (web-pages or man-pages) right from the terminal. Our users shouldn't have to care about man-pages or have to find out where we host our website. The desired behaviour of lounger would be:

1. The CLI prints a general help message if it was called without a command or if a passed command doesn't exist. It gives the user a hint how to proceed further to get their task done.
2. It is easy to get additional help for a command

We start by creating a new library function, `lib/help.js`. The first function will print a friendly help text to the user. The help text gets constructed in a helper function (no pun intended).

We can get all of our available commands by calling `Object.keys(lounger.cli)`. We chain a `.join(', ')` call to separate each command with a comma and a space:

```
function getGeneralHelpMessage () {
  const commands = Object.keys(lounger.cli).join(', ');
```

The next part is a template string which explains how to use lounger. We add all available commands which are exposed on the command line interface. We also explain that users without a clue can run `lounger help` together with the command they are interested in to get detailed help for a command. Additionally we provide an example how they would call the help.

The final line tells them which version of lounger they run, which comes handy in different situations. Usually people forget which version of a package they have installed. The reason is simple: nobody can remember the exact version of all software they have installed right now. Imagine a user who just read a blog article about lounger version 2.3 and a great new feature they want to try. They installed lounger some time ago, but sadly the command is just available since version 2.3 which got released three days ago. In this case they get immediate feedback that they run on an older version.

Here is the whole function `getGeneralHelpMessage`:

```
function getGeneralHelpMessage () {
  const commands = Object.keys(lounger.cli).join(', ');

  const message = `Usage: lounger <command>

The available commands for lounger are:

${commands}

You can get more help on each command with: lounger help <command>

Example:
lounger help isonline

lounger v${lounger.version} on Node.js ${process.version}`;

  return message;
}
```

The next function we'll build will try to open the man-page if possible. It will fallback to the website version for Windows users.

The module `opener` does a great job to open files on different operating systems. This also applies to html files, as we want to open them in the default browser of the current operating system. So we install `opener` as our next dependency:

```
$ npm install --save opener
```

We then require `opener` at the top of `help.js`:

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

We also have to spawn the `man` command later and find out the absolute path for our website files:

```
const spawnSync = require('child_process').spawnSync;
const path = require('path');
```

The core module `os` can help us to find out if we are running on Windows:

```
const isWindows = require('os').platform() === 'win32';
```

We then have to spawn the `man` command or open the default browser for the desired functionality. With `stdio: inherit` for the spawn command we can see and interact

with the output of the spawned process:

```
function openDocumentation (command) {

  if (isWindows) {
    const htmlFile = path.resolve(__dirname + '/../website/cli-' + command +
'.html');
    return opener('file:///' + htmlFile);
 }

  spawnSync('man', ['lounger-' + command], {stdio: 'inherit'});
}
```

The last task is putting all our helper functions together, if a command is not available, we print the general help. If the command exists, we are opening the man-page or the browser, depending on the Operating System:

```
exports.cli = help;
function help (command) {
  return new Promise((resolve, reject) => {

    if (!lounger.cli[command]) {
      console.log(getGeneralHelpMessage());
    } else {
      openDocumentation(command);
    }

    resolve();
  });
}
```

If you want you can also add some code to make it configurable for the user which type of documentation is opened for them. Maybe Linux users prefer the Website version of the docs. We can already try the new help command, as it gets picked up by our `lounger.js` file:

```
1. bash
(09:55:34) [robert@tequila-new] ~/cli-book/sourcecode/help-system (master *+) $ ./bin/lounger-cli help
Usage: lounger <command>

The available commands for lounger are:

help, isonline

You can get more help on each command with: lounger help <command>

Example:
lounger help isonline

lounger v1.0.0 on Node.js v4.2.4
(09:55:50) [robert@tequila-new] ~/cli-book/sourcecode/help-system (master *+) $
```

Figure 5. Our help command in action

Our final task in this section is that we have to make sure to print the general help in cases where the user does not enter a command at all. In `bin/lounger-cli` we require the `help.js` file and modify the `lounger.load` call. In case we don't find the command in `lounger.cli`, we print the general help:

```
const help = require('../lib/help.js');
lounger.load(parsed).then(() => {

  if (!lounger.cli[cmd]) {
    return help.cli();
  }

  lounger.cli[cmd]
    .apply(null, parsed.argv.remain)
    .catch(errorHandler);

}).catch(errorHandler);
```

Congrats! We are finished with the help system now and made significant progress! Feel free to play around with our new help system by entering these commands:

```
$ lounger blerg
$ lounger
$ lounger help
$ lounger help isonline
```

The code for this section is located at: `sourcecode/help-system`.

Configuration

Requiring our users to add their favourite settings as flags by hand every time they use lounger can be cumbersome. A configuration file enables our users to save the settings they need every day . It would be nice if we could support this feature that we covered in [_it_supports_powerusers_configuration] to make our power users happier.

It is common practice that command line tools put their configuration files into the home directory of the user. The home-directory is different for every Operating System, but the module `osenv` can help us to find the current home directory:

```
$ npm install --save osenv
```

We basically have to see if a `loungerrc` configuration exists in our home directory, and if not, we have to create an empty one. We do that in `lounger-cli`, in order to keep the API functions free of the side effect. This is the changed `lounger-cli` file where we added a `require`-call for `osenv` and create a config file right after the argument parsing. We also add the path to the config file to our parsed arguments:

```
#!/usr/bin/env node

const lounger = require('../lib/lounger.js');
const pkg = require('../package.json');
const log = require('npmlog');
const nopt = require('nopt');
const help = require('../lib/help.js');
const osenv = require('osenv');
const fs = require('fs');

const parsed = nopt({
  'json': [Boolean]
}, {'j': '--json'}, process.argv, 2);

const home = osenv.home();
parsed.loungerconf = home + '/' + '.loungerrc';

if (!fs.existsSync(parsed.loungerconf)) {
  fs.writeFileSync(parsed.loungerconf, '');
}
```

The config itself will use ini-formatted config files in order to store and read settings. `config-chain` is a module to load configurations with different priorities based on the order we load them. It also supports ini-formatted files.

Let's create a file `lib/config.js` and require `config-chain`:

```
$ npm i --save config-chain
```

```
'use strict';

const cc = require('config-chain');
```

`config-chain` is able to manage multiple configurations. It will override configuration settings according to the order we load them.

For our use case we want the options provided as arguments on the command line to have the highest priority. Meaning, they override the ones from the config file. As the loading of the file is done async, we have to listen to the `load` event emitted by `config-chain` once it is finished. For the use cases where no config file was set, we don't try to load add it as file. On all errors we reject our promise and pass the error object:

```
exports.loadConfig = loadConfig;
function loadConfig (nopts) {
  return new Promise((resolve, reject) => {
    let cfg;

    if (!nopts.loungerconf) {
      cfg = cc(nopts)
        .on('load', () => {
          resolve(cfg);
        }).on('error', reject);
    } else {
      cfg = cc(nopts)
        .addFile(nopts.loungerconf, 'ini', 'config')
        .on('load', () => {
          resolve(cfg);
        }).on('error', reject);
    }
  });
};
```

The config object that is returned from config chain has nice `get` and `set` methods. We can even save the config back to the configuration file after changing the config using the

`save` method. For now we have to integrate `loadConfig` into the bootstrap of lounger. Do you remember the `config.get` method in `lounger.js` from the previous chapter? We will replace it with the config object that is returned by our `load` function.

As a first step we have to load our config in `lounger.js`:

```
const config = require('./config.js');
```

In `lounger.load` we are going to load the config:

```
lounger.load = function load (opts) {
  return new Promise((resolve, reject) => {
    config
      .loadConfig(opts)
      .then((cfg) => {

      });
```

The other content of `lounger.load` including the `fs.readdir` call is moved into the callback of the chained `then` function:

```
lounger.load = function load (opts) {
  return new Promise((resolve, reject) => {

    config.loadConfig(opts)
      .then((cfg) => {

        lounger.config = cfg;

        fs.readdir(__dirname, (err, files) => {
          files.forEach((file) => {
            if (!/\.js$/.test(file) || file === 'lounger.js') {
              return;
            }

            const cmd = file.match(/(.*)\.js$/)[1];
            const mod = require('./' + file);
            if (mod.cli) {
              cli[cmd] = mod.cli;
            }

            if (mod.api) {
              api[cmd] = mod.api;
            }

          });
```

```
          lounger.loaded = true;
          resolve(lounger);
        });
      }).catch(reject);

  });
};
```

If we now run lounger it will create a config file for us in our home directory. On OSX my config is at `~/.loungerrc`.

After the file got created (don't forget to run lounger at least one time) we can set JSON output to true in `~/.loungerrc`:

```
json = true
```

Just try out `$ lounger isonline http://example.com`, lounger will print JSON now. We can still override the config on the command line:

```
$ lounger isonline --no-json
```

As manually editing the `~/.loungerrc` is not very user friendly, we will build a `lounger config` command. The config command should have the abilities to show the config and its values and to set config values.

Here is the proposed CLI:

```
$ lounger config set json true
$ lounger config get json
```

The API could look like this:

```
lounger.commands.set('json', true)
lounger.commands.get('json')
```

`config-chain` offers the data provided in the loaded config file as `.sources.config.data`. For JSON formatted output we return the whole config if no key was provided:

```
    const data = lounger.config.sources.config.data;

    if (lounger.config.get('json') && !key) {
```

```
      resolve(data);
      return;
    }
```

If a key was provided we build a JSON object that just contains the value for our key:

```
    if (lounger.config.get('json') && key) {
      resolve({[key]: data[key]});
      return;
    }
```

Given we don’t want JSON formatted output and provided a key we return the value:

```
    if (key) {
      resolve(lounger.config.sources.config.data[key]);
      return;
    }
```

In the last case where the json setting is set to false, and no key was provided we simply read the unparsed ini-file:

```
    resolve(fs.readFileSync(lounger.config.sources));
```

Here is the whole `get` function:

```
function get (key) {
  return new Promise((resolve, reject) => {
    const data = lounger.config.sources.config.data;

    if (lounger.config.get('json') && !key) {
      resolve(data);
      return;
    }
    if (lounger.config.get('json') && key) {
      resolve({[key]: data[key]});
      return;
    }

    if (key) {
      resolve(lounger.config.sources.config.data[key]);
      return;
    }

    resolve(fs.readFileSync(lounger.config.sources));
  });
}
```

Modifying the config is done by the set function. It takes a key and a value, calls `set` on the config-chain object and resolves the promise after the values are written to the disk:

```
function set (key, value) {
  return new Promise((resolve, reject) => {
    if (!key && !value) {
      reject(new Error('key and value required'));
      return;
    }

    lounger.config.set(key, value, 'config');
    lounger.config.on('save', () => {
      resolve();
    });

    lounger.config.save('config');
  });
}
```

As a last step we have to expose both commands:

```
exports.api = {
  get: get,
  set: set
};
```

We build the CLI functionality on top of our API functions. The main difference between the API and CLI function is that the CLI function has side effects for the `get` command: it prints the result to the console. We also add some nice error messages to make it easier to use for our users. They get instructions how to run the command if they don't use it properly:

```
exports.cli = cli;
function cli (cmd, key, value) {
  return new Promise((resolve, reject) => {

    function getUsageError () {
      const err = new Error([
        'Usage:',
        '',
        'lounger config get [<key>]',
        'lounger config set <key> <value>',
      ].join('\n'));
      err.type = 'EUSAGE';
      return err;
    }
```

```
    if (!cmd || (cmd !== 'get' && cmd !== 'set')) {
      const err = getUsageError();
      return reject(err);
    }

    if (cmd === 'get') {
      return get(key).then((result) => {
        console.log(result);
      }).catch(reject);
    }

    if (cmd === 'set') {
      if (!key && !value) {
        const err = getUsageError();
        return reject(err);
      }
      return set(key, value).catch(reject);
    }
  });
}
```

Great! We have a config command now! The code for this section is at `sourcecode/config`. We fulfilled all points from [_it_supports_powerusers] and [_you_never_get_stuck] and are ready for an initial release!

Our first release & release tips

You can publish Open Source modules after registering an account at the npm registry. After registering the account you basically just have to type `npm publish` to publish your module to the registry. Before we publish lounger I want to mention that there are some nice ways to optimise the published packages. In this section I will explain how to optimise your package regarding size and installation time.

One way to keep the installation size small is to add a `.npmignore` file to the root directory. It works similar to a `.gitignore` file and npm won't include the listed files and directories in the published package.

Depending on the type of the files which aren't needed when using the module we are able to save a lot of space on the hard disks of our users. We will save them a lot of bandwidth, too.

We can save some space if we exclude our unit and integration tests and also the source for the compiled documentation:

```
/test/
/docs/
build.js

.DS_Store
npm-debug.log
```

Another great way to speedup installation time is to include all production dependencies in the published module. We can tell npm to bundle them with our module by adding them as `bundleDependencies` to our `package.json`:

```
  "bundleDependencies": [
    "config-chain",
    "nopt",
    "npmlog",
    "opener",
    "osenv",
    "request"
  ],
```

Bundling the dependencies reduces the installation time a lot, as we omit all the small HTTP requests for each dependency and their dependencies during installation. With the current npm 3 it reduces the installation time from 20 seconds to 5 seconds for a broadband connection. Bundling the dependencies also makes sure that our package is still installable even if a module was unpublished.

When we now run `npm publish` we will publish a highly optimised version of our package. The code for this section is at `sourcecode/first-release`.

Migration of large amounts of data using Streams

Sometimes we want to process large amounts of data. With traditional buffering we run into memory problems very soon. The whole data just doesn't fit into the memory of the computer. Streams enable us to process data in small slices. Node.js streams work like Unix streams on the terminal, where you pipe data from a producer into a consumer using the pipe symbol `|`.

The `cat` program prints the content of files to `stdout`. In this example I am piping the output of the `cat` program, into `tr` to change all letters from our `package.json` to upper case letters:

```
$ cat sourcecode/first-release/package.json | tr 'a-z' 'A-Z'
```

It works with all output on `stdout`:

```
$ echo "i shout?" | tr 'a-z' 'A-Z'
```

Streams in Unix and in Node.js enable us to compose small programs or modules that do one thing well to get our task done. They handle backpressure, which means that a fast producer will automatically slow down if it is piped into a slow consumer.

The most used streams in Node are the `Readable`, `Writeable` and `Transform` streams. They are base classes that can be used to build your own custom streams. There are also other stream types, like the `Duplex` stream and the `Passthrough` stream which aren't covered in the book. The `Readable` stream is used to read input data, the `Transform` stream is usually used to modify chunks of data and the `Writeable` streams accepts data to write it somewhere (e.g. into a file).

Today we will write a command that will use streams for piping data from CSV files into CouchDB/PouchDB. We could also write an importer to migrate data from a database, e.g. a Postgres or MongoDB, but with plain CSV files we don't have to install a new database. The principle applies to both source types, files and databases. At the end of the chapter I will provide a link to an example for a Node.js stream pipeline that migrates data from MongoDB to CouchDB/PouchDB.

The first stream

Building streams can be a bit tricky sometimes. We start by creating the file `stream-example.js` in the root dir of our module.

Our first iteration will output our CSV contents to `stdout`. We will use it to learn how streams work and build our importer on top of it later. To develop we need a CSV file, we can save it to `test/fixtures/test.csv`:

```
time;location
march;austin,us
april;boston,us
october;bristol,uk
february;hermigua,es
march;hermigua,es
april;havana,cu
```

Luckily we don't have to write our own streaming CSV parser:

```
$ npm install --save csv-parse
```

We require the `fs`, and `util` module. The `util` module is used to inherit from the base object Transform. The `fs` module is needed to read the CSV file from disk. We also need to require the CSV parser:

```
const parse = require('csv-parse');
const fs = require('fs');
const Transform = require('stream').Transform;
const util = require('util');
```

Our custom stream called `MyTransformStream` inherits from `stream.Transform`. We set the stream into `objectMode` to be able to process the JSON input from the CSV parser:

```
function MyTransformStream () {

  Transform.call(this, {
    objectMode: true
  });
}

util.inherits(MyTransformStream, Transform);
```

A Transform stream has to implement one method: `_transform`. The method is called for every chunk of data that we are processing. In the `_transform` method we can transform the chunks to something new. The transformed data is then pushed to the next consumer using `this.push`. Once we are finished we call the `done` callback to signal that we are finished with this chunk. Right now we just want to take a look how a chunk looks like:

```
MyTransformStream.prototype._transform = transform;
function transform (chunk, encoding, done) {

  console.log('chunk: ', chunk);

  this.push(chunk);

  done();
}
```

As last step we have to pipe the CSV file into the CSV parser and the parsed output into our custom stream:

```
const opts = {comment: '#', delimiter: ';', columns: true};
const parser = parse(opts);
const input = fs.createReadStream(__dirname + '/test/fixtures/test.csv');

input
  .pipe(parser)
  .pipe(new MyTransformStream());
```

When we now run `node streams-example.js` we get this output:

```
$ node streams-example.js
chunk:  { time: 'march', location: 'austin,us' }
chunk:  { time: 'april', location: 'boston,us' }
chunk:  { time: 'october', location: 'bristol,uk' }
chunk:  { time: 'february', location: 'hermigua,es' }
chunk:  { time: 'march', location: 'hermigua,es' }
chunk:  { time: 'april', location: 'havana,cu' }
```

Every chunk is a JSON object and every time we call `done` and if there is still input produced, `_transform` is called with the next chunk. We could take every chunk and post it against the CouchDB / HTTP API now. We would keep our memory footprint super low, but we would also send a lot of HTTP requests and the whole migration would

take very long. A healthy compromise is to buffer a few chunks and post them against the bulk APIs of CouchDB/PouchDB. This way we don't buffer all existing data and run out of memory and we are finished earlier with our import, as we don't have to send so many HTTP requests.

The code for this section can be found at `sourcecode/streams/streams-example.js`.

The Transform and Writeable stream

For our next stream we will create the file `streams-bulk-example.js` in the root directory of lounger. It will take the objects from the CSV parsing stream and buffer them. At a given limit it will pass the buffered objects to the next consumer. The result passed to the next consumer is ready to get posted against the CouchDB/PouchDB bulk docs API endpoint. The CouchDB/PouchDB bulk API accepts an array of objects wrapped with `{"docs": []}`.

We start with the same set of modules:

```
const parse = require('csv-parse');
const fs = require('fs');
const Transform = require('stream').Transform;
const util = require('util');
```

The name of our stream will be `TransformToBulkDocs` and it will take options as an object. Using the options we can specify the amount of documents to buffer:

```
function TransformToBulkDocs (options) {
  if (!options) {
    options = {};
  }

  if (!options.bufferedDocCount) {
    options.bufferedDocCount = 200;
  }
```

The empty array for `this.buffer` will be our buffer:

```
  Transform.call(this, {
    objectMode: true
  });

  this.buffer = [];
```

```
  this.bufferedDocCount = options.bufferedDocCount;
}

util.inherits(TransformToBulkDocs, Transform);
```

Here is the whole constructor function:

```
function TransformToBulkDocs (options) {
  if (!options) {
    options = {};
  }

  if (!options.bufferedDocCount) {
    options.bufferedDocCount = 200;
  }

  Transform.call(this, {
    objectMode: true
  });

  this.buffer = [];
  this.bufferedDocCount = options.bufferedDocCount;
}

util.inherits(TransformToBulkDocs, Transform);
```

In the `_transform` method we add every chunk to our buffer:

```
TransformToBulkDocs.prototype._transform = transform;
function transform (chunk, encoding, done) {

  this.buffer.push(chunk);
```

If the buffer has grown big enough we call the method `this.push`, which we inherited from the base Transform stream. `this.push(args)` tells Node that we want to pass `args` to the next consumer in our stream pipeline. We then empty the buffer for new data that might arrive:

```
  if (this.buffer.length >= this.bufferedDocCount) {
    this.push({docs: this.buffer});
    this.buffer = [];
  }

  done();
}
```

Here is the whole `_transform` method:

```
TransformToBulkDocs.prototype._transform = transform;
function transform (chunk, encoding, done) {

  this.buffer.push(chunk);
  if (this.buffer.length >= this.bufferedDocCount) {
    this.push({docs: this.buffer});
    this.buffer = [];
  }

  done();
}
```

The last part of our file is almost identical to our first streams example:

```
const opts = {comment: '#', delimiter: ';', columns: true};
const parser = parse(opts);
const input = fs.createReadStream(__dirname + '/test/fixtures/test.csv');

input
  .pipe(parser)
  .pipe(new TransformToBulkDocs());
```

We add a temporary `console.log` to our code to see if it works:

```
  if (this.buffer.length >= this.bufferedDocCount) {
    this.push({docs: this.buffer});
    console.log({docs: this.buffer});
    this.buffer = [];
  }
```

When we run `node streams-bulk-example.js` we see that… it doesn't work! Why?

The problem is that we don't have enough documents to reach the default document count of 200. The same applies to the remaining documents of a set. If we have 250 initial documents as input, the first 200 hundreds are pushed to the next consumer, but the remaining 50 are lost. Luckily the Node.js developers were aware of the problem and provided the `_flush` method. The method doesn't have to be implemented to make a Transform stream work, like the `_transform` method. Instead we can chose to implement it, given we need it.

The `_flush` method will get called at the very end after all data was consumed by the stream, but before the stream emits the `end` event which signals the end of the stream. `_flush` will get called at the very end and given we still have a few buffered documents, we push them to the next consumer:

```
TransformToBulkDocs.prototype._flush = flush;
function flush (done) {

  this.buffer.length && this.push({docs: this.buffer});
  done();
}
```

That's our first custom stream! Don't forget to remove the `console.log` call we added!

The next consumer in our pipeline will take the collected documents and post them against the CouchDB / PouchDB bulk docs endpoint. As the streams are able to handle backpressure, the other streams will wait until we successfully added the documents to CouchDB / PouchDB. They will continue to pass us data down the pipeline once we are able to pull in the next collection of documents.

The next stream we will build accepts the data from the Transform stream and writes it into the database. It is a `Writeable` stream and we will add it to our `streams-bulk-example.js` file:

```
const Writable = require('stream').Writable;
```

Our Writable stream needs to know where to put the data, so we will need to pass it the database url. As the methods of the stream are called for each chunk we have to store the passed url as `this.url`:

```
function CouchBulkImporter (options) {
  if (!options) {
    options = {};
  }

  if (!options.url) {
    const msg = [
      'options.url must be set',
      'example:',
      "new CouchBulkImporter({url: 'http://localhost:5984/baseball'})"
    ].join('\n')
    throw new Error(msg);
```

```
  }

  Writable.call(this, {
    objectMode: true
  })

  // sanitise url, remove trailing slash
  this.url = options.url.replace(/\/$/, '');
}
util.inherits(CouchBulkImporter, Writable);
```

To implement a child of a Writable stream, we have to implement the `_write` method. Like the `_transform` method of the Transform stream, the `_write` method is called for every chunk that is passed to the stream from the previous producer. In our case we send the JSON chunks as JSON to the database using request. After we sent the data successful to the `/_bulk_docs` API endpoint we call the `done` callback to signal that we are ready for a new chunk:

```
CouchBulkImporter.prototype._write = write;
function write (chunk, enc, done) {
  request({
    json: true,
    uri: this.url + '/_bulk_docs',
    method: 'POST',
    body: chunk
  }, function (err, res, body) {
    if (err) {
      return done(err);
    }

    if (!/^2../.test(res.statusCode)) {
      const msg = 'CouchDB server answered: \n Status: ' +
        res.statusCode + '\n Body: ' + JSON.stringify(body);
      return done(new Error(msg));
    }

    done();
  });
}
```

We also have to require `request` in our file:

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

To use the stream we have to pipe the data into it. We update the last section of `streams-bulk-example.js`:

```
input
  .pipe(parser)
  .pipe(new TransformToBulkDocs())
  .pipe(new CouchBulkImporter({url: 'http://127.0.0.1:5984/travel'}));
```

After we created the database `travel` and run our script, we have imported the CSV:

```
$ curl -X PUT http://localhost:5984/travel
{"ok":true}
$ node streams-bulk-example.js
$ curl http://localhost:5984/travel/_all_docs
{"total_rows":6,"offset":0,"rows":[{"id":"3444bf7c-65c0-438f-f8e8-
7f55124f1736","key":"3444bf7c-65c0-438f-f8e8-7f55124f1736","value":{"rev":"1-
37fcd2e5b40939805b8e043da44f9b1d"}},{"id":"568c3b0f-78fe-43a2-9eac-
06620cfaa595","key":"568c3b0f-78fe-43a2-9eac-06620cfaa595","value":{"rev":"1-
e047788bac9aada0564fc928642d3960"}},{"id":"5d170e63-d845-4e45-f760-
97e30cbc4b21","key":"5d170e63-d845-4e45-f760-97e30cbc4b21","value":{"rev":"1-
6f1794e4eb24b60665fe02c2624d53eb"}},{"id":"9cd34a61-8a48-4b88-afbf-
7fd6e3c9cf42","key":"9cd34a61-8a48-4b88-afbf-7fd6e3c9cf42","value":{"rev":"1-
747b11a103cc0fe31d1c546ef70d69c0"}},{"id":"cc7da504-438a-40c1-842b-
4722b63c9a37","key":"cc7da504-438a-40c1-842b-4722b63c9a37","value":{"rev":"1-
0f9e7add8b7fbb773dc7d3081475d855"}},{"id":"f09811c5-43ec-4a6d-bd3b-
b34b3674676d","key":"f09811c5-43ec-4a6d-bd3b-b34b3674676d","value":{"rev":"1-
d677c5bbbee1bbbe45f6128a9cf1fe8d"}}]}
```

We can access a single document using the `id`:

```
$ curl http://localhost:5984/travel/3444bf7c-65c0-438f-f8e8-7f55124f1736
{"time":"february","location":"hermigua,es","_id":"3444bf7c-65c0-438f-f8e8-
7f55124f1736","_rev":"1-37fcd2e5b40939805b8e043da44f9b1d"}
```

Looks great! Seems we have everything in place to use our low-level streaming functions in the command line client. The code for this section is located at `sourcecode/streams/streams-bulk-example.js`.

The streaming import command

For the last part of this chapter we will reuse the custom stream implementation that we created in The Transform and Writeable stream. In the real world I would create two modules for our two streams to make them reusable across multiple projects, but for now we can copy the code for the `CouchBulkImporter` and the

`TransformToBulkDocs` streams into `lib/csv.js` which will be the home of our import command:

```
const parse = require('csv-parse');
const fs = require('fs');
const Transform = require('stream').Transform;
const util = require('util');
const Writable = require('stream').Writable;
const request = require('request');
const lounger = require('./lounger.js');

function TransformToBulkDocs (options) {
  if (!options) {
    options = {};
  }

  if (!options.bufferedDocCount) {
    options.bufferedDocCount = 200;
  }

  Transform.call(this, {
    objectMode: true
  });

  this.buffer = [];
  this.bufferedDocCount = options.bufferedDocCount;
}

util.inherits(TransformToBulkDocs, Transform);

TransformToBulkDocs.prototype._transform = transform;
function transform (chunk, encoding, done) {

  this.buffer.push(chunk);
  if (this.buffer.length >= this.bufferedDocCount) {
    this.push({docs: this.buffer});
    this.buffer = [];
  }

  done();
}

TransformToBulkDocs.prototype._flush = flush;
function flush (done) {

  this.buffer.length && this.push({docs: this.buffer});
  done();
}
```

```
function CouchBulkImporter (options) {
  if (!options) {
    options = {};
  }

  if (!options.url) {
    const msg = [
      'options.url must be set',
      'example:',
      "new CouchBulkImporter({url: 'http://localhost:5984/baseball'})"
    ].join('\n')
    throw new Error(msg);
  }

  Writable.call(this, {
    objectMode: true
  })

  // sanitise url, remove trailing slash
  this.url = options.url.replace(/\/$/, '');
}
util.inherits(CouchBulkImporter, Writable);


CouchBulkImporter.prototype._write = write;
function write (chunk, enc, done) {
  request({
    json: true,
    uri: this.url + '/_bulk_docs',
    method: 'POST',
    body: chunk
  }, function (err, res, body) {
    if (err) {
      return done(err);
    }

    if (!/^2../.test(res.statusCode)) {
      const msg = 'CouchDB server answered: \n Status: ' +
        res.statusCode + '\n Body: ' + JSON.stringify(body);
      return done(new Error(msg));
    }

    done();
  });
}
```

Let’s think a bit about the command we are going to build. A CSV can have different delimiters, some use semicolons as a delimiter, others are using commas or tabs. The

symbols to denote a comment can also change. In our previous implementations we used fixed values:

```
const opts = {comment: '#', delimiter: ';', columns: true};
```

For a real world use case the symbols for the delimiter and comment must be configurable. The CSV input is usually a file.

Here is a possible CLI:

```
$ lounger csv transfer <file> <database> [--delimiter=;] [--comment=#] [--
chunksize=200]
```

The command `csv` is open to extension and can host all CSV related commands in the future. The command reads quite nicely and is easy to remember: `lounger csv transfer <file> <database>` reads almost as `lounger [do] csv transfer [from] <file> [to] <database>`. Sane defaults help us to avoid passing optional modifiers at all, but in case we need to modify them we can change every important aspect of our import.

I’m not sure if you noticed it, but when we played with our streams we would have had to create the target database using curl in advance. It would be handy if our CLI users don’t have to create the target database on their own. Our goal is to help them to solve their task as quickly and easily as possible, so we should automatically create databases as necessary.

The function `createTargetDatabase` is a helper function which wraps `request` into a Promise. If the database was created (HTTP code 201 or 200) or the database exists already (HTTP code 412) we resolve, all other states lead to rejection of the Promise:

```
function createTargetDatabase (url) {
  return new Promise((resolve, reject) => {
    request({
      json: true,
      uri: url,
      method: 'PUT',
      body: {}
    }, function (er, res, body) {

      if (er && (er.code === 'ECONNREFUSED' || er.code === 'ENOTFOUND')) {
        const err = new Error(
```

```
          'Could not connect to ' + url + '. Please check if the database is
offline'
        );
        err.type = 'EUSAGE';
        return reject(err);
      }

      if (er) {
        return reject(er);
      }

      const code = res.statusCode;

      if (code !== 200 && code !== 201 && code !== 412) {
        const msg = 'CouchDB server answered: \n Status: ' +
          res.statusCode + '\n Body: ' + JSON.stringify(body);
        return reject(new Error(msg));
      }
      resolve();
    });
  });
}
```

In case of an `ECONNREFUSED` or `ENOTFOUND` error we can safely assume that the database is currently offline and ask the user to take a look if the database is available. I can't stress enough how important proper error handling is. Take this example, where we are getting back `ECONNREFUSED`:

```
$ ./bin/lounger-cli csv transfer test/fixtures/test.csv
http://127.0.0.1:1337/testimport
ERR! connect ECONNREFUSED 127.0.0.1:5984
ERR! Error: connect ECONNREFUSED 127.0.0.1:5984
ERR!     at Object.exports._errnoException (util.js:870:11)
ERR!     at exports._exceptionWithHostPort (util.js:893:20)
ERR!     at TCPConnectWrap.afterConnect [as oncomplete] (net.js:1063:14)
ERR!
ERR!
ERR! lounger: 1.0.0 node: v4.2.4
ERR! please open an issue including this log on
http://example.com/lounger/issues
```

Depending on how much our users used Node.js before they would be very puzzled I guess. The only way to continue for them would be to ask a search engine or to open an issue. After receiving the issue our boring job would be to close the issue and tell them that they probably had a typo in their url.

After writing the `createTargetDatabase` function we should have all our supporting functions in place. As usual we are starting to implement the main CLI functions by implementing the API command which we will then wrap with our CLI function.

The delimiter and comment options are defined in the config file or are passed on the command line. To know what their values are, we have to interact with `lounger.config`. To access `lounger.config` we have to require it:

```
const lounger = require('./lounger.js');
```

The main API function checks if all necessary arguments were provided and applies defaults if no configuration was passed in from the config file or on the command line. We create the database in case it does not exist yet and delegate to the helper function `importFromCsvFile`:

```
exports.api = {
  transfer: bulkdocsImport
};

function bulkdocsImport (file, targetDb) {
  return new Promise((resolve, reject) => {
    const opts = {};

    if (!file && !targetDb) {
      return reject(new Error('file and/or targetDb argument missing'));
    }

    opts.delimiter = lounger.config.get('delimiter') || ';';
    opts.comment = lounger.config.get('comment') || '#';
    opts.chunksize = lounger.config.get('chunksize') || 200;

    createTargetDatabase(targetDb)
      .then(() => {
        return importFromCsvFile(file, targetDb, opts);
      }).catch(reject);
  });
}
```

`importFromCsvFile` accepts the source CSV file, url and options and creates the stream pipeline. The main difference to our previous code in `streams-example.js` is that we have proper error handling in place to catch all errors.

```
function importFromCsvFile (file, url, opts) {
  return new Promise((resolve, reject) => {
    const options = {comment: opts.comment, delimiter: opts.delimiter,
columns: true};
    const parser = parse(options);
    const input = fs.createReadStream(file);

    input
      .pipe(parser)
      .on('error', reject)
      .pipe(new TransformToBulkDocs({bufferedDocCount: opts.chunksize}))
      .on('error', reject)
      .pipe(new CouchBulkImporter({url: url}))
      .on('error', reject);
  });
}
```

The CLI functions finally wraps our API method and adds friendly error messages:

```
exports.cli = importCli;
function importCli (cmd, file, target) {
  return new Promise((resolve, reject) => {
    if (!cmd || cmd !== 'transfer' || !file || !target) {
      const err = new Error(
        'Usage: lounger csv transfer <file> <database> [--delimiter=;] [--
comment=#] [--chunksize=200]'
      );
      err.type = 'EUSAGE';
      return reject(err);
    }

    return bulkdocsImport(file, target).catch(reject);
  });
}
```

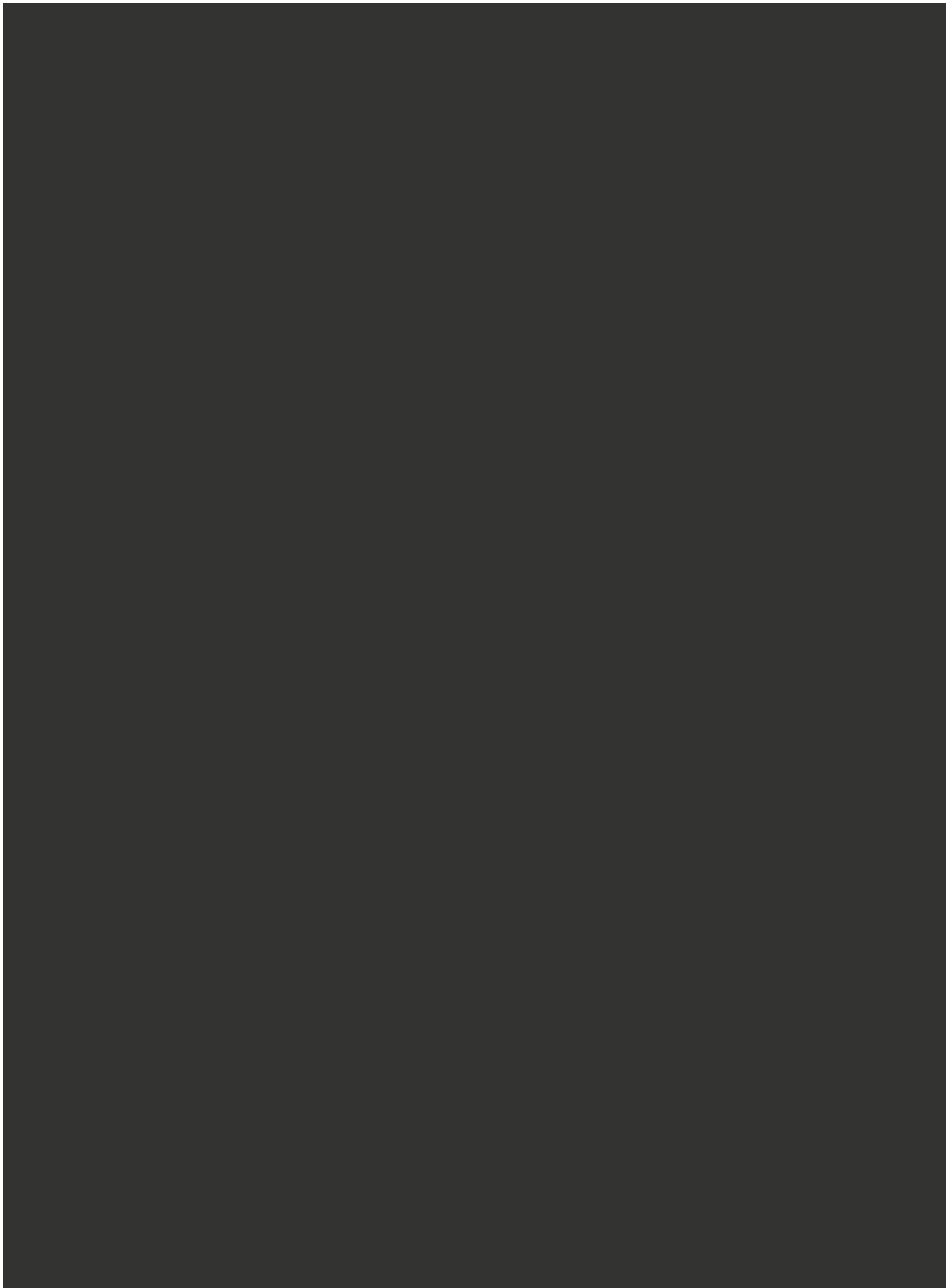
We introduced three new options, `delimiter`, `comment` and `chunksize`. `lounger.config` enables our users to set default values using the config file. In addition we have to take care that the options are parsed on the command line. In `bin/lounger-cli` we have to register our optional arguments to `nopt`:

```
const parsed = nopt({
  'json': [Boolean],
  'delimiter': [String],
  'comment': [String],
  'chunksize': [Number]
}, {'j': '--json'}, process.argv, 2);
```

That’s it! We created a command that is able to stream large amounts of data into our database.

If you are interested in a stream pipeline which would stream data from MongoDB to CouchDB you can take a look at https://github.com/robertkowalski/couchbulkimporter/blob/master/examples/mongo.js.

The code for this section is at `sourcecode/streams`, enjoy!



TIPS & TRICKS

This section collects some tips and tricks regarding Node.js development in general.

Testing

With proper unit and integration tests in place, ensure new features or bugfixes don't introduce regressions. There are a lot of great services that provide a hosted CI environment. They can test every Pull Request before it is merged, which makes reviewing code a lot easier. A popular service for hosted CI is Travis CI. Travis CI is free for Open Source projects.

Semantic Versioning with SemVer

I would recommend to follow Semantic Versioning with SemVer (http://semver.org/). SemVer divides the version number of a release into three areas: MAJOR.MINOR.PATCH. The version 3.5.8 would have 3 as MAJOR version level, 5 as MINOR version level and 8 as patchlevel. Given a new release includes a breaking change, the MAJOR version number is bumped. A new feature would just need a minor version bump and bugfixes would just require a bump of the PATCH section.

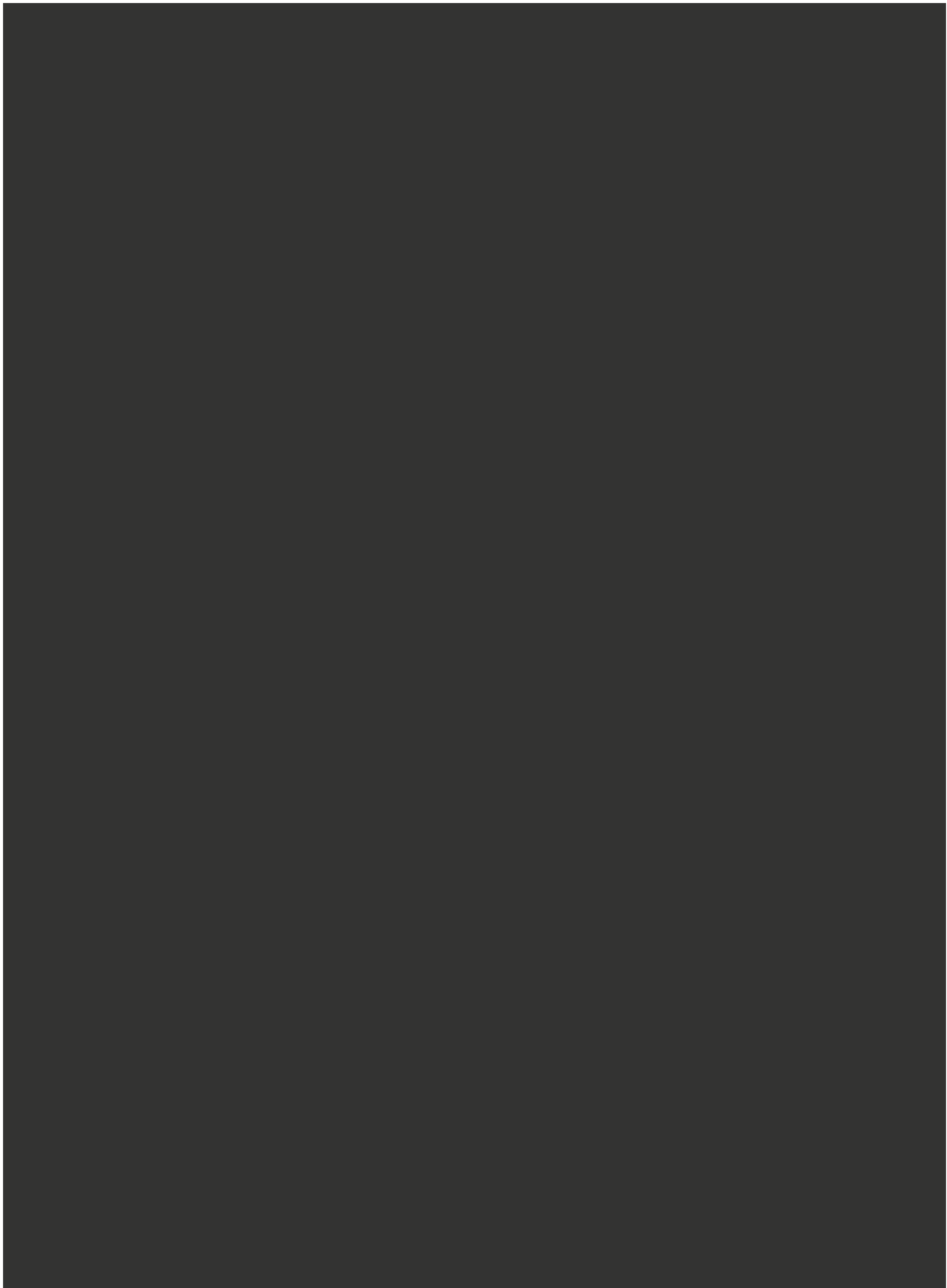
Example: My package has version 3.5.7 and I add a new feature which does not break backwards compatibility. My next release would be 3.6.0.

This way your users get an idea if a release might break their production code (MAJOR), it contains a new feature (MINOR) or a bug fix (PATCH). A great tool to help you to make the right decision for the next version bump is semantic-release (https://www.npmjs.com/package/semantic-release).

Greenkeeper

Keeping track which dependencies of your project got a new version and need to get updated can be tedious. The update itself (bumping the version number in the `package.json`) is not the most interesting task on earth, too. A new and exciting service is http://greenkeeper.io. Once you registered it for your project it will send you pull requests with updated versions of your dependencies. If you have a testsuite in place and everything is “green“ you just have to merge the Pull Request from the Greenkeeper

bot. Testing and a CI Service that automatically runs the tests, SemVer and Greenkeeper really show their strengths when combined together.



THE END

I hope you enjoyed *The CLI Book*!

Our journey to successful CLIs doesn't end here, it just begins.

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 – please recommend the book to others in case you like it.