6. Command-line interfaces

Node has become a popular platform for building command-line interfaces, or CLIs. For example, the CoffeeScript[1] compiler and the Grunt[2] task framework are both Node programs. In this chapter, we'll examine how we can build Node CLI apps that are easy to test. We'll start by treating a CLI as a black box, and move onto manipulating its internal workings using unit tests.

6.1. A black-box program

To get us started, let's write a simple little program that prints the sum of all the arguments passed to it. Here it is:

Figure 6.1. `node/blackbox_cli/bin/add`

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

var args = process.argv.slice(2)

var sum = args.reduce(function(s, n) {
  return s + Number(n)
}, 0)

console.log(sum)
```

Executable files, or just *executables*, are usually placed in the `bin` directory of a project. 'bin' is short for *binary*; executables are also referred to as binaries even if they contain source code in a scripting language.

The first line, `#!/usr/bin/env node`, is colloquially called a shebang[3]. It tells the operating system how to run the file when it's executed directly, i.e. when we run `./bin/add` in the terminal, the operating system turns this into "`/usr/bin/env node ./bin/add`". We use `/usr/bin/env` because the shebang must use an absolute path, and we don't know where the user will have Node installed; `env` is a standard Unix program[4] with a predictable location, and all it does run the arguments passed to it as a command in the current environment. That is, running "`/usr/bin/env node`" runs `node` by finding it in your `PATH`, exactly as the shell does when you type `node` at the prompt.

In Node, the program's arguments are exposed via the `process.argv` array. The first two elements are usually the string `"node"`, followed by the path of the program being run, followed by the arguments given to the program via the command-line. So, if we were to run

```
$ ./node/blackbox_cli/bin/add foo 42
```

then `process.argv` would be

```
["node", "/absolute/path/to/node/blackbox_cli/bin/add", "foo", "42"]
```

So, the program grabs the arguments using `process.argv.slice(2)`, and uses `Array.reduce()`[5] to sum the given values. It then uses `console.log()` to print the result to stdout.

Writing a test for this is simple: we can call the program with some arguments and see what comes out. Node has a built-in module called `child_process`[6] that makes running external programs quite easy.

[1] http://coffeescript.org/
[2] http://gruntjs.com/
[3] http://en.wikipedia.org/wiki/Shebang_(Unix)
[4] http://en.wikipedia.org/wiki/Env
[5] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce
[6] http://nodejs.org/api/child_process.html

In these tests, we use `child_process.spawn()` to run the program directly. This function takes a path to an executable (or just the name of the executable if it's on the current process's `PATH`), and an array containing the arguments to pass to the executable. It returns an object representing the new process, with methods to control its execution and read and write data to and from it. Just like the Node `process` object, this subprocess has three streams attached to it called `stdin`, `stdout` and `stderr` that provide access to the standard I/O streams for the process. As we discussed in Section 3.4.2, "Checking a readable stream's output", we can use the concat-stream[7] module to collect what the child process writes to stdout and check it.

This test just runs the program with a couple of example inputs and checks the output.

Figure 6.2. `node/blackbox_cli/spec/add_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add")
  }})

  it("prints the sum of some integers", function(resume) { with(this) {
    var proc = child.spawn(executable, ["1", "2", "3"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})

  it("prints the sum of some floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "5.86\n", sum ) })
    }))
  }})
}})
```

There's one important thing about these tests: they treat the program under test as a black box, that is, they can't see what the program's internals are like. It could be a compiled C binary and these tests would work just fine. Handy to know you can write tests in your favourite scripting language if you're building such a program.

Command-line arguments are just one source of input a program can use. It can also use its standard input stream, environment variables, config files — both system-wide and user-specific — and the contents of files it's asked to operate on. It can even get information from the internet or from a database. When we write tests, we want to manipulate these inputs and see how the program responds; what it prints to stdout, what its exit status is, what file changes it makes, and so on. These all pose different challenges, especially when used in combination, and you need to pick the right approach to make sure your tests are robust. Let's consider some input sources for our black-box program.

[7] https://npmjs.org/package/concat-stream

6.1.1. Command-line arguments

Our original example already uses command-line arguments; those are the values passed in the command following the name of the program. But the program doesn't do anything fancy with them aside from turning them from strings into numbers. Most command-line programs have some system of flags and switches — option names that are prefixed with two dashes and control how the program behaves.

Let's change our program so that it takes a flag called `--round`, which if set makes it round off any fractional numbers it receives. To do this, we need to differentiate this `--round` flag from normal input values, and we can do this with an argument parser like nopt[8].

Figure 6.3. `node/blackbox_cli/bin/add_args`

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

var nopt = require("nopt")

var params = nopt({round: Boolean}, {}, process.argv),
    round  = params.round,
    args   = params.argv.remain

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

Just like the original program, this can be easily tested using a combination of `child_process.spawn()` and `concat()`. To test the flag, we have one example that uses it and one that does not, to check the change in behaviour that the flag causes.

[8] https://npmjs.org/package/nopt

Figure 6.4. node/blackbox_cli/spec/add_args_spec.js

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_args", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_args")
  }})

  it("prints the sum of some floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "5.86\n", sum ) })
    }))
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    var proc = child.spawn(executable, ["--round", "3.14", "2.72"])

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})
}})
```

You can get much fancier than this with argument parsing, but this pattern remains useful: use nopt or posix-argv-parser[9] to parse the program arguments, and use `child_process.spawn()` and `concat()` to launch a process and check its output. Since you're probably going to come up with quite a lot of examples of what the program can do, it might help to extract all the boilerplate into a function to slim the tests down a bit:

[9] https://npmjs.org/package/posix-argv-parser

Figure 6.5. node/blackbox_cli/spec/add_args_optimised_spec.js

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

var executable = path.join(__dirname, "..", "bin", "add_args")

var execute = function(args, callback) {
  var proc = child.spawn(executable, args)
  proc.stdout.pipe(concat(function(output) {
    callback(output.toString("utf8"))
  }))
}

JS.Test.describe("add_args 2", function() { with(this) {
  it("prints the sum of some floats", function(resume) { with(this) {
    execute(["3.14", "2.72"], function(output) {
      resume(function() { assertEqual( "5.86\n", output ) })
    })
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    execute(["--round", "3.14", "2.72"], function(output) {
      resume(function() { assertEqual( "6\n", output ) })
    })
  }})
}})
```

You might notice that running these tests is much slower than it would be if we tested the addition and rounding logic directly. This is because of the overhead of spinning up a new Node process for each test, and piping output into our test suite. This suggests that testing the internal functions might be preferable to executing the program from outside, but there will be even stronger motivations for doing this later.

6.1.2. Environment variables

Sometimes, you want to make an aspect of a program's behaviour controlled via an implicit setting, rather than having to explicitly use a flag every time you run it. This is what environment variables do: every Unix program runs in an *environment*, which roughly speaking is a bag of name-value pairs that contain settings. If you run `env` in your terminal you'll see the environment your shell is running with; here's a few of my environment variables:

Figure 6.6. Common Unix environment variables

```
$ env | sort
EDITOR=vim
HOME=/home/jcoglan
LANG=en_GB.UTF-8
LOGNAME=jcoglan
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/home/jcoglan/projects/javascript-testing-recipes
SHELL=/bin/bash
TERM=xterm-256color
```

When a Unix process spawns a child, that child process inherits its parent's environment, and can make changes to it that are only visible to the child process. So, when you run a program from the terminal, it can see all those settings that `env` shows you. Some of these settings are standardised, for example `HOME` is the current user's home directory, `PATH` tells the shell which directories to find programs in, and `EDITOR` tells programs what to launch if they want the user to edit some text; for example when I run `git-commit` in this environment it will launch `vim` so I can enter a commit message.

Environment variables can be set by the user, either for the duration of a shell session by typing

Figure 6.7. Exporting an environment variable

```
$ export FOO=something
```

or by setting the variable just for one command, for example the `FORMAT` variable we use for running tests:

Figure 6.8. Setting a variable for one command

```
$ FORMAT=tap node node/hello_world/test.js
1..1
ok 1 - Hello, world! runs a test
```

So you can invent and use environment variables to control your programs. Let's replace the command-line argument `--round` to our `add` program with a `ROUND` environment variable, which Node exposes through `process.env`:

Figure 6.9. `node/blackbox_cli/bin/add_env`

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

var round = "ROUND" in process.env,
    args  = process.argv.slice(2)

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

Just as you can pass command-line arguments when using `child_process.spawn()`, you can also set the environment that you want the child process to run in. This lets you try running the program with different environment variables to check what happens. To do this, we make a new object that inherits from the current process's environment, and change some of its settings before handing it to `child_process.spawn()`.

Figure 6.10. `node/blackbox_cli/spec/add_env_spec.js`

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_env", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_env")
  }})

  it("prints the sum of some rounded floats", function(resume) { with(this) {
    var env = Object.create(process.env)
    env.ROUND = "1"

    var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

    proc.stdout.pipe(concat(function(output) {
      var sum = output.toString("utf8")
      resume(function() { assertEqual( "6\n", sum ) })
    }))
  }})
}})
```

Note that when you use the `env` parameter to `child_process.spawn()`, that sets the entire environment for the child process. i.e. if you only pass `{env: {ROUND: "1"}}` then the child process will *only* see that one environment variable. By saying `env = Object.create(process.env)`, we make an environment that inherits from the current process[10], as environments usually work. Whether you want to inherit the current environment or start with a blank slate depends on the kind of program you're testing, but in this case the child process needs to *at least* see the same `PATH` as the test process, so that the shebang `#!/usr/bin/env node` resolves to the Node you're currently running.

6.1.3. Configuration files

Programs with more complex configuration, or programs that want to be able to save changes to their configuration, often use configuration files rather than environment variables. On most Unix systems, system-wide configuration lives in the `/etc/program-name` directory and user-specific configuration lives in `$HOME/.config/program-name` or `$HOME/.program-name`.

But before you hard-code those locations into your program, you should see a big red flag. In order to test the program, we'd need to poke configuration data into a shared system directory or into the user's home directory. Manipulating the state of the host system is dangerous, since it may clobber the user's existing config and might require root privileges. Ideally, we should isolate the effect of the tests so they don't manipulate any files outside the current project.

We can do this by allowing an environment variable to override the default config file location, and this has the added benefit that it allows the end user to change the location if they want their config stored somewhere else. Here's a version of our `add` program that looks in a config file for the rounding setting. It uses the environment variable `ADD_CONFIG_PATH` to let the user override the config path.

Figure 6.11. `node/blackbox_cli/bin/add_config`

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

var DEFAULT_CONFIG_PATH = "/etc/add/config.json"

var fs     = require("fs"),
    args   = process.argv.slice(2),
    path   = process.env.ADD_CONFIG_PATH || DEFAULT_CONFIG_PATH,
    config = fs.readFileSync(path)

config = JSON.parse(config.toString("utf8"))

var sum = args.reduce(function(s, n) {
  n = Number(n)
  if (config.round) n = Math.round(n)
  return s + n
}, 0)

console.log(sum)
```

We can test this using the technique we discussed above for passing an environment variable into the child process. Before each test, we make a new env using `Object.create(process.env)`, with `ADD_CONFIG_PATH` set to a path inside the project's test directory. Each test writes a different config file to that location using `fs.writeFile()` before running the program, in the modified environment, and checking its output. After each test, we need to make sure to delete our fake config file using `fs.unlink()`.

Notice how the inner `before()` blocks express the name of the containing `describe()` block in code; the `describe()` expresses something abstract like "with rounding enabled", and the `before()` block

[10] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create

contains the code to make that the case. Using the test names to express intention helps keep your test suite easy to understand.

Figure 6.12. node/blackbox_cli/spec/add_config_spec.js

```
var JS     = require("jstest"),
    child  = require("child_process"),
    fs     = require("fs"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("add_config", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "add_config")

    this.env = Object.create(process.env)
    env.ADD_CONFIG_PATH = path.join(__dirname, ".config.json")
  }})

  after(function(resume) { with(this) {
    fs.unlink(env.ADD_CONFIG_PATH, resume)
  }})

  describe("with rounding enabled", function() { with(this) {
    before(function(resume) { with(this) {
      fs.writeFile(env.ADD_CONFIG_PATH, '{"round": true}', resume)
    }})

    it("prints the sum of some rounded floats", function(resume) { with(this) {
      var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

      proc.stdout.pipe(concat(function(output) {
        var sum = output.toString("utf8")
        resume(function() { assertEqual( "6\n", sum ) })
      }))
    }})
  }})

  describe("with rounding disabled", function() { with(this) {
    before(function(resume) { with(this) {
      fs.writeFile(env.ADD_CONFIG_PATH, '{"round": false}', resume)
    }})

    it("prints the sum of some floats", function(resume) { with(this) {
      var proc = child.spawn(executable, ["3.14", "2.72"], {env: env})

      proc.stdout.pipe(concat(function(output) {
        var sum = output.toString("utf8")
        resume(function() { assertEqual( "5.86\n", sum ) })
      }))
    }})
  }})
}})
```

6.1.4. Standard input

Many CLIs read from stdin and process the input before emitting it to stdout. Such programs can be quite easily tested by writing to the child process's `stdin` stream and reading from its `stdout` stream, both of which are exposed by the object returned by `child_process.spawn()`. Also, as covered in Section 3.4.5, "Decoupled streams", it's not usually necessary to test such programs as a black box since you can implement the logic as a standalone stream that's not coupled to standard I/O.

But some CLIs are interactive; they ask the user to confirm things, or enter their password or other personal details. Some programs ask for their configuration interactively, `ssh-keygen` is a good example of this. This is typically done using the readline[11] library; here's a small example that asks the user a silly question and prints a response based on the user's input. If the input is not what the program expects, the response is written to stderr and it also changes the exit status to indicate success or failure.

Figure 6.13. `node/blackbox_cli/bin/doge`

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

var readline = require("readline")

var interface = readline.createInterface({
  input:  process.stdin,
  output: process.stderr
})

interface.question("Who's a good doge? ", function(answer) {
  if (answer === "such question, wow") {
    console.log("Good doge!")
    process.exit(0)
  } else {
    console.error("Nope")
    process.exit(1)
  }
})
```

Notice how the readline interface writes its question prompts to standard *error*. This is because it's part of the user interface, not the logical output of the program, and since Unix only gives us two possible streams to send program output to, stderr is sometimes used for this.

Let's try to write a test for this. We'll spawn a child process, write the user's input to stdin, and read from stdout or stderr depending on which response we're expecting. These tests use concat-stream to bundle up the entire output of each stream before checking it.

[11] http://nodejs.org/api/readline.html

Figure 6.14. node/blackbox_cli/spec/doge_quick_spec.js

```
var JS     = require("jstest"),
    child  = require("child_process"),
    path   = require("path"),
    concat = require("concat-stream")

JS.Test.describe("doge", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "doge")
    this.proc       = child.spawn(executable)
  }})

  it("rewards a good doge", function(resume) { with(this) {
    proc.stdin.write("such question, wow\n")

    proc.stdout.pipe(concat(function(output) {
      output = output.toString("utf8")
      resume(function() { assertEqual( "Good doge!\n", output ) })
    }))
  }})

  it("punishes a bad doge", function(resume) { with(this) {
    proc.stdin.write("WOOF!\n")

    proc.stderr.pipe(concat(function(output) {
      output = output.toString("utf8")
      resume(function() {
        assertEqual( "Who's a good doge? Nope\n", output )
      })
    }))
  }})
}})
```

These seem simple enough but there are some subtleties that could cause problems later. Notice how the second test's output contains the question we asked the user, but the first test's output doesn't. This is because we write the UI to stderr, and the second test's output goes to stderr, so when we read from stderr we get the UI mixed in with the output. In more complex programs this becomes a big problem.

Also, we haven't really written the test in a way that *interacts* with the program being tested. We just throw user input to stdin immediately, without checking which question we've been asked, or even waiting for the question to be asked at all. This might affect the behaviour of the program, and makes the test quite unexpressive: it doesn't really show how the program works when a user interacts with it for real.

To fix these problems, as well as to fix the fact we're not checking the program's exit status, we need to listen out for individual prompts from the program and send responses to them at appropriate times. We need to differentiate user interface from logical output and act accordingly. The following test rewrites our previous attempt to be more explicit about the data flow of the program and how it should be interacted with.

Notice in particular how the first test does not have to check what `stdout` is emitting, but the second test does: the UI and output are interleaved in the second test but not in the first. If the program had much more conditional interface flow, however, the tests would need to reflect this.

Figure 6.15. node/blackbox_cli/spec/doge_spec.js

```
var JS    = require("jstest"),
    child = require("child_process"),
    path  = require("path")

JS.Test.describe("doge", function() { with(this) {
  before(function() { with(this) {
    this.executable = path.join(__dirname, "..", "bin", "doge")
    this.proc       = child.spawn(executable)
    this.output     = null
  }})

  it("rewards a good doge", function(resume) { with(this) {
    proc.stdout.on("data", function(chunk) {
      output = chunk.toString("utf8")
    })
    proc.stderr.on("data", function(chunk) {
      proc.stdin.write("such question, wow\n")
    })
    proc.on("exit", function(status) {
      resume(function() {
        assertEqual( "Good doge!\n", output )
        assertEqual( 0, status )
      })
    })
  }})

  it("punishes a bad doge", function(resume) { with(this) {
    proc.stderr.on("data", function(chunk) {
      var response = chunk.toString("utf8")
      if (response === "Who's a good doge? ") {
        proc.stdin.write("WOOF!\n")
      } else {
        output = response
      }
    })
    proc.on("exit", function(status) {
      resume(function() {
        assertEqual( "Nope\n", output )
        assertEqual( 1, status )
      })
    })
  }})
}})
```

This is more explicit but I hope you can see how quickly this approach becomes complicated and cumbersome. In addition, some libraries for consuming user input, such as the password prompting `pw`[12] module, only work if stdin is a TTY[13], which severely restricts our ability to test it as a child process. Even if the test process is running attached to a TTY, it's not a good idea to make the child process inherit this stream since its output will then be intermingled with the output of your tests.

We've encountered several problems that indicate that testing CLIs as black boxes can be tricky, and in cases where the program talks to an external system like a database or the internet it can be almost impossible. Let's see what we can do to improve matters.

[12] https://npmjs.org/package/pw
[13] http://en.wikipedia.org/wiki/TTY

6.2. Breaking open the black box

Testing CLIs as a black box can be awkward at best and impossible at worst. Controlling the environment, collecting stream output, dealing with config files and standard I/O all introduce complexity, but that complexity is manageable. For some tasks though, we really need control of the inner workings of the program so we can stub things out. Let's look at a more realistic example than those we've considered so far: a program that talks to an API over the internet to gather and display data for us.

6.2.1. Talking to the internet

Suppose we want a program to help us monitor issues on our GitHub projects. It should request the list of issues for a project and display a summary for us. For example, here are the current open issues on one of my repositories:

Figure 6.16. Displaying open issues for a GitHub repo

```
$ ./node/github_issues/bin/blackbox faye faye-websocket-node
3 open issues:

#27: Forward networking error events to the websocket event listeners.
https://github.com/faye/faye-websocket-node/pull/27
[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT

#26: Allow server to limit max frame length (avoid DOS and crash)
https://github.com/faye/faye-websocket-node/issues/26
[glasser] Sat, 12 Oct 2013 02:12:05 GMT

#16: Compression support with "deflate-frame" protocol extension
https://github.com/faye/faye-websocket-node/issues/16
[nilya] Fri, 14 Sep 2012 00:37:40 GMT
```

I've written the first version of this program as a simple script, hence the name `blackbox`. Its source is printed below. The program takes the repo owner and project name via `process.argv`, makes an HTTPS[14] request to `api.github.com` to get the open issues for that project, checks the response was successful, then prints the title, URL, author and creation date for each issue. If there's an error sending the request or any non-successful response is received, the error is printed and the program exits with a non-zero status.

One thing should jump out about this program, even more so than when we were testing configuration files. This program talks to the internet, so we'd need to provide stub responses in our tests. Otherwise, the tests will fail every time anyone opens, closes or edits an issue on the repo we're testing against. But the way the program is written, this is almost impossible to achieve without heavily manipulating the system: you'd need to spin up a local server that returns canned GitHub API responses, put an entry in `/etc/hosts` to route `api.github.com` to the local machine, and get an SSL certificate that Node will validate successfully[15].

Here's the full program:

[14] http://nodejs.org/api/https.html

[15] This would involve changing the certificate chain in `libcurl` so that it trusts a self-signed certificate, or getting an actual SSL certificate for `api.github.com`. One of those is a lot of hassle and the other ought not to be possible.

Figure 6.17. node/github_issues/bin/blackbox

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

var https  = require("https"),
    concat = require("concat-stream"),
    owner  = process.argv[2],
    repo   = process.argv[3]

var request = https.request({
  method:   "GET",
  host:     "api.github.com",
  path:     "/repos/" + owner + "/" + repo + "/issues",
  headers:  {"User-Agent": "Node.js"}
})

request.on("response", function(response) {
  if (response.statusCode !== 200) {
    console.error("Repository not found")
    process.exit(1)
  }

  response.pipe(concat(function(body) {
    var issues = JSON.parse(body.toString("utf8"))
    console.log(issues.length + " open issues:\n")

    issues.forEach(function(issue) {
      var date = new Date(issue.created_at).toGMTString()
      console.log("#" + issue.number + ": " + issue.title)
      console.log(issue.html_url)
      console.log("[" + issue.user.login + "] " + date + "\n")
    })
  }))
})

request.on("error", function(error) {
  console.error("Error connecting to GitHub: " + error.message)
  process.exit(1)
})

request.end()
```

Since this program is almost impossible to test as-is, we're going to need to break it up and unit-test the internal bits. We can start with isolating the dependency on external resources, i.e. create an API wrapper for talking to GitHub.

6.2.2. Wrapping external dependencies

The first step in our refactoring will be to extract the logic for talking to GitHub into a simple API that we call from elsewhere in the program. This means that instead of having HTTP request code with `console.log()` and `process.exit()` calls baked into its response handlers, we move the HTTP logic into a module that just returns whatever the result of the request was.

It's important to keep calls to `console.log()` and `process.exit()` out of our application code since those will interfere with the tests, either by interspersing the test output with program output or by prematurely ending the test run. This has the side effect of making the code more reusable, since it doesn't make assumptions about what the user wants to do with the result of the request.

Here's a module containing just the HTTP logic and error handling; it makes a request, and if it's successful it hands back the parsed JSON body.

Figure 6.18. node/github_issues/lib/github.js

```
var https  = require("https"),
    concat = require("concat-stream")

var GitHub = {
  HOSTNAME: "api.github.com",

  getIssues: function(owner, repo, callback) {
    var path    = "/repos/" + owner + "/" + repo + "/issues",
        headers = {"User-Agent": "Node.js"},
        params  = {method: "GET", host: this.HOSTNAME, path: path, headers: headers},
        request = https.request(params)

    request.on("response", function(response) {
      if (response.statusCode !== 200) {
        return callback(new Error("Repository not found: " + path))
      }
      response.pipe(concat(function(body) {
        var issues = JSON.parse(body.toString("utf8"))
        callback(null, issues)
      }))
    })

    request.on("error", function(error) {
      callback(new Error("Error connecting to GitHub: " + error.message))
    })

    request.end()
  }
}

module.exports = GitHub
```

(I could have used the `request`[16] module instead of the `https` module since it's easier to use and easier to stub out. However, this example demonstrates some useful techniques that are more broadly applicable.)

Now, we need to make sure this works. That is, we need to write some tests that say what this module returns given different HTTP response scenarios. To do this, we need to stub out the `https.request()` API and make it return responses that we control. The way we're going to do this illustrates a very general approach for how to stub out dependencies in your programs, it's not just tied to HTTP.

The first thing to realise is that the code in `GitHub.getIssues()` doesn't *know* that it's making an HTTP request. Yes, it's calling an API called `https.request()`, but those are just names, they don't have any deep technical meaning to the code itself. All the code actually depends upon is the fact that `https.request()` returns some object that responds to two methods, `request.on()` and `request.end()`. Therefore, we can use a vanilla `EventEmitter` as our fake response object, adding a stubbed `end()` method to it.

The second thing is that the response object we get from `request.on("response")` is just a stream — we call `pipe()` on it to run the response body through a JSON parser. So, we can use any other stream that returns JSON as a stand-in, and add a `statusCode` property to it since that's the only other aspect of the response the code cares about. We can get some canned response data by grabbing a snapshot from the internet:

Figure 6.19. Saving stub response data

```
$ curl https://api.github.com/repos/faye/faye-websocket-node/issues \
    > node/github_issues/spec/fixtures/issues.json
```

[16] https://npmjs.org/package/request

Once we have that data, we can use `fs.createReadStream()` to create a stream that emits the contents of our snapshot, which is enough to trick our code into thinking it's processing a response body.

The tests proceed as follows. In the `before()` block we set up the fake request and response objects as we've described, and stub `https.request()` to return our fake request. In each test, we manipulate the response — either setting a status code, or emitting an error instead — and check what `github.getIssues()` returns. The order of instructions in the tests is a little unusual; since this API is asynchronous we make a call to it *before* deciding how the request should be resolved. If we emit a `"response"` event before making the `getIssues()` call, nothing would happen.

Figure 6.20. `node/github_issues/spec/github_spec.js`

```
var JS     = require("jstest"),
    github = require("../lib/github"),
    events = require("events"),
    fs     = require("fs"),
    https  = require("https"),
    path   = require("path")

JS.Test.describe("GitHub", function() { with(this) {
  before(function() { with(this) {
    this.request  = new events.EventEmitter()
    this.response = fs.createReadStream(path.join(__dirname, "fixtures", "issues.json"))

    var params = {method: "GET", host: "api.github.com", path: "/repos/foo/bar/issues"}
    stub(https, "request").given(objectIncluding(params)).returns(request)
    stub(request, "end")
  }})

  it("yields the issues on a 200 response", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertNull( error )
        assertEqual( 3, issues.length )
        assertMatch( /^Forward networking error events/, issues[0].title )
      })
    })
    response.statusCode = 200
    request.emit("response", response)
  }})

  it("yields an error on a 404 response", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertEqual( "Repository not found: /repos/foo/bar/issues", error.message )
        assertEqual( undefined, issues )
      })
    })
    response.statusCode = 404
    request.emit("response", response)
  }})

  it("yields an error on request error", function(resume) { with(this) {
    github.getIssues("foo", "bar", function(error, issues) {
      resume(function() {
        assertEqual( "Error connecting to GitHub: No network connection", error.message )
        assertEqual( undefined, issues )
      })
    })
    request.emit("error", new Error("No network connection"))
  }})
}})
```

This, then, describes a general approach to stubbing out system resources your programs depend on: the code that's using the `https` or `fs` modules doesn't really *know* it's making HTTP calls or talking to the filesystem; it just knows it's calling some functions, listening to events and so on. When you stub out these components, you don't need to fully recreate them, you only need to supply enough of their interface to make the code work. If you can use something that looks almost the same, like an event emitter or a stream, by all means do so, and stick whatever extra methods and properties you need onto the fake object. It's JavaScript: there's no static type system and you can change objects pretty much however you like, and this gives you a lot of flexibility when stubbing things.

Once we've shown that the `GitHub` module works as required, we are free to stub its interface rather than that of `https` when we're testing other components. Wrapping HTTP calls in a simpler API that's tailored to our use case makes it easier to stub that functionality out in an expressive way, without dealing with the complexity of HTTP requests, query strings, serialization, status codes, and so on. We've boiled it down to a single function that yields either an error or a list of issue objects, which is simpler to work with.

6.2.3. Presenting information

The other major component of Figure 6.17, "`node/github_issues/bin/blackbox`" besides fetching the issue data is how to display the issues to the user. In the black-box program, this is done by putting `console.log()` calls in the response handler of the API request, but when we modularise the program for testing this won't fly. In general, code that interacts with, and possibly mutates, globals like `process.stdout`, `process.argv` and `process.env` is hard to test, because it can't be isolated from the rest of the system or from the tests themselves. Code that uses `console.log()` will inject things into the middle of the test output, and code that requires changes to `process.env` means setting things the entire process can see and will be affected by. In short: global side-effects considered harmful.

In this situation there's a straightforward way out: rather than think of the presentation logic as a side effect, think of it as a pure function[17], one that takes an issue object and returns a textual representation of it for displaying in the terminal.

We can write a quick spec for what this function should do:

Figure 6.21. `node/github_issues/spec/presenters_spec.js`

```
var JS         = require("jstest"),
    presenters = require("../lib/presenters")

JS.Test.describe("Presenters", function() { with(this) {
  before(function() { with(this) {
    this.issue = {
      number:     99,
      title:      "The 99th problem",
      html_url:   "http://example.com/issues/99",
      user:       {login: "zee-j"},
      created_at: "2013-10-24T19:00:29Z"
    }
  }})

  it("presents an issue as text", function() { with(this) {
    assertEqual( "#99: The 99th problem\n" +
                 "http://example.com/issues/99\n" +
                 "[zee-j] Thu, 24 Oct 2013 19:00:29 GMT\n",
                 presenters.text(issue) )
  }})
}})
```

[17] A *pure function* is a function that has no side effects and does not rely on global state; its only output is its return value and it computes this using only the values of its parameters.

The issue doesn't have to contain real GitHub data, and it doesn't have to be complete. Remember: when faking things out, you only need the fake to have the same interface as the object it stands in for, from the point of view of the code that's using it. So, we don't have a whole issue object here, we only have the fields the presenter needs. We don't use real data, but the field values are of the same type and structure as the real thing: we replace objects with objects, URLs with URLs, strings with strings and numbers with numbers.

Writing a function that satisfies this is fairly easy; we could use a template library but I'm just going to use string concatenation[18]:

Figure 6.22. node/github_issues/lib/presenters.js

```
var Presenters = {
  text: function(issue) {
    var date = new Date(issue.created_at).toGMTString()

    return "#" + issue.number + ": " + issue.title + "\n" +
           issue.html_url + "\n" +
           "[" + issue.user.login + "] " + date + "\n"
  }
}

module.exports = Presenters
```

So now we have presentation logic that's decoupled from `console.log()`, and we can test it and reuse it with ease. The final step is to make a class that represents the CLI itself.

6.2.4. Modelling the CLI

We've extracted out most of the ingredients of the `blackbox` program, but we still need to test the program itself, not just its components. Since we need to be able to stub out some of the internal APIs, we're not going to do this by running the program as a child process. Therefore, we need to wrap the contents of the executable in a function or class, so we can run it multiple times with different inputs in our test suite.

The job of this class will be to glue the ingredients together and wire them up to the user interface available to the command-line program: the `argv`, `env`, `exit`, `stdin`, `stdout` and `stderr` APIs. But, the class should not refer to any of those directly; since they are shared globals visible to everything in the test process, manipulating them can cause other program components or the tests themselves to behave strangely. You don't want the program mixing its output into your tests' output, or calling `process.exit()` and quitting the test suite early.

So, our approach will be to inject these `process` APIs that the program needs access to; the class's constructor will take a parameter called `ui` (for 'user interface') that contains all the things the program needs to interact with. In the tests, we can use this parameter to pass in different arguments, environment variables, and fake standard I/O streams, and when we run the program for real it's just a matter of writing a tiny glue script that passes in the real things.

Here's the `CLI` class, which takes `argv` and `stdout` from the `ui` parameter. Its `run()` method executes the logic of the program and invokes a callback to tell us whether there was an error. Rather than calling `process.exit()` in this class, we pass an error back to the caller and let it deal with it properly.

[18] If you are doing this for real, make sure you escape any characters/bytes in the data you're displaying that have special meaning to the terminal; otherwise it's possible for a cleverly named issue to make itself display in green, or hide your cursor, or clear the screen completely.

Figure 6.23. node/github_issues/lib/cli.js

```
var github     = require("./github"),
    presenters = require("./presenters")

var CLI = function(ui) {
  this._argv   = ui.argv
  this._stdout = ui.stdout
}

CLI.prototype.run = function(callback) {
  var owner = this._argv[2],
      repo  = this._argv[3],
      self  = this

  github.getIssues(owner, repo, function(error, issues) {
    if (error) return callback(error)
    self._stdout.write(issues.length + " open issues:\n\n")
    issues.forEach(function(issue) {
      self._stdout.write(presenters.text(issue) + "\n")
    })
    callback(null)
  })
}

module.exports = CLI
```

There's not much logic in this class: the code for gathering the data we need, and most of the code for presenting it, is in other classes that we've already tested. Our main concerns when testing this class are to check that it passes the command-line arguments through to the GitHub client correctly, that it writes the issue information to stdout when the repo exists, and that when there is an error it yields the error to the callback function.

As with all integration testing, the point is not to test all the edge cases of the components that make up the program; that should be done using unit tests of the components themselves. The aim should be to test the main different types of behaviour whose logic is in the `CLI` class. In this case, there is only one real *decision* the class makes: what to do on success and what do to on failure, and we only need an example of each test to check the code is wired up correctly.

The tests for the `CLI` class are listed below. We instantiate the class with an array containing our 'fake' `process.argv` arguments, and a stream to stand in for `process.stdout`. We can use Node's `stream.PassThrough` class for this; it's a read/write stream that emits whatever is written to it, so the `CLI` class can write to it as it would to the real stdout, and we can read that output in our tests by piping the stream into `concat()`.

Although this program doesn't make use of the environment or config files, the same approach can be used for those: just as we pass an array in for `argv`, we can pass an object in for `env` and pass in paths for config files to use during tests. When running the program for real, we would pass in `process.env` and the default config file location instead. Or, we could make the executable responsible for reading the config file instead, and it would pass the resulting settings into the `CLI` class; the tests would then just pass in config data without creating fake config files for the class to read.

As in the `GitHub` tests, we use the `issues.json` fixture file to provide some fake data for the success case, and we use `expect()` rather than `stub()` since we want to assert that the input from `argv` is actually passed through to the GitHub client correctly. For the failure case, we make the client yield an error instead of some issue data, and check that the error is passed back to the caller.

Figure 6.24. node/github_issues/spec/cli_spec.js

```
var JS     = require("jstest"),
    fs     = require("fs"),
    path   = require("path"),
    stream = require("stream"),
    concat = require("concat-stream"),
    CLI    = require("../lib/cli"),
    github = require("../lib/github")

JS.Test.describe("CLI", function() { with(this) {
  before(function() { with(this) {
    this.stdout = new stream.PassThrough()
    this.cli = new CLI({
      argv:   ["none", "bin/github", "foo", "bar"],
      stdout: stdout
    })
  }})

  describe("when GitHub returns issues", function() { with(this) {
    before(function() { with(this) {
      var fixture = fs.readFileSync(path.join(__dirname, "fixtures", "issues.json"), "utf8")
      expect(github, "getIssues").given("foo", "bar").yields([null, JSON.parse(fixture)])
    }})

    it("displays the issues", function(resume) { with(this) {
      cli.run(function() { stdout.end() })

      stdout.pipe(concat(function(output) {
        resume(function() {
          assertEqual(
            '3 open issues:\n' +
            '\n' +
            '#27: Forward networking error events to the websocket event listeners.\n' +
            'https://github.com/faye/faye-websocket-node/pull/27\n' +
            '[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT\n' +
            '\n' +
            '#26: Allow server to limit max frame length (avoid DOS and crash)\n' +
            'https://github.com/faye/faye-websocket-node/issues/26\n' +
            '[glasser] Sat, 12 Oct 2013 02:12:05 GMT\n' +
            '\n' +
            '#16: Compression support with "deflate-frame" protocol extension\n' +
            'https://github.com/faye/faye-websocket-node/issues/16\n' +
            '[nilya] Fri, 14 Sep 2012 00:37:40 GMT\n' +
            '\n',
            output.toString("utf8") )
        })
      }))
    }})
  }})

  describe("when GitHub returns an error", function() { with(this) {
    before(function() { with(this) {
      stub(github, "getIssues").given("foo", "bar").yields([new Error("ECONNREFUSED")])
    }})

    it("yields the error", function(resume) { with(this) {
      cli.run(function(error) {
        resume(function() { assertEqual( "ECONNREFUSED", error.message ) })
      })
    }})
  }})
}})
```

Now that we've put all the program's logic into classes that we can test, all the executable has to do is create an instance of `CLI`, passing in the real `process` APIs for it to talk to. It then runs the program, and if there's an error it prints the error and exits with a non-zero status. This file is now so simple it doesn't really need testing; all the logic is somewhere else.

Figure 6.25. `node/github_issues/bin/modular`

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

var CLI = require("../lib/cli")

var program = new CLI({
  argv:   process.argv,
  stdout: process.stdout
})

program.run(function(error) {
  if (!error) return
  console.error(error.message)
  process.exit(1)
})
```

If the program makes use of environment variables or configuration files, or needs to know if stdout is a TTY[19], the executable can pass those in too, and the tests can pass in crafted inputs, alternate config paths, and so on.

Figure 6.26. Passing environment variables and configuration paths to a CLI

```
var program = new CLI({
  config: "/etc/github_issues/config.json",
  argv:   process.argv,
  env:    process.env,
  stdout: process.stdout,
  tty:    require("tty").isatty(1)
})
```

6.2.5. Command-line arguments

If you're using an option parser like nopt, it makes sense to put the parsing logic in the `CLI` class rather than the executable glue file. Argument parsing can become quite complex and the parser libraries sometimes have bugs in them, so it's worth testing how the `argv` array gets processed rather than leaving the parsing in the executable and bypassing it in the tests.

These parsers can operate on any array, not just `process.argv`, so you can easily use them in any code you like. Here's another version of the `CLI` class that has been modified to take a `--limit` option to limit the number of issues it displays.

[19] http://nodejs.org/api/tty.html

Figure 6.27. node/github_issues/lib/cli_limit.js

```
var nopt       = require("nopt"),
    github     = require("./github"),
    presenters = require("./presenters")

var CLI = function(ui) {
  this._argv   = ui.argv
  this._stdout = ui.stdout
}

CLI.prototype.run = function(callback) {
  var params = nopt({limit: Number}, {}, this._argv),
      owner  = params.argv.remain[0],
      repo   = params.argv.remain[1],
      self   = this

  github.getIssues(owner, repo, function(error, issues) {
    if (error) return callback(error)

    if (params.limit) {
      issues = issues.slice(0, params.limit)
    }

    self._stdout.write(issues.length + " open issues:\n\n")
    issues.forEach(function(issue) {
      self._stdout.write(presenters.text(issue) + "\n")
    })
    callback(null)
  })
}

module.exports = CLI
```

Putting the parsing logic in this class means we can test it easily. We can write a test similar to the success case in `cli_spec.js`, and change the `argv` value we pass in to include the `"--limit"` option. Remember when doing this that the elements of `process.argv` are always strings; the shell does not know you mean some things to be numbers. So, even when passing in numeric options in a fake `argv` array, pass them as strings rather than as numbers so that your tests are a realistic recreation of what happens when the program is run for real.

The modified test is shown below.

Figure 6.28. node/github_issues/spec/cli_limit_spec.js

```
var JS     = require("jstest"),
    fs     = require("fs"),
    path   = require("path"),
    stream = require("stream"),
    concat = require("concat-stream"),
    CLI    = require("../lib/cli_limit"),
    github = require("../lib/github")

JS.Test.describe("CLI with limit", function() { with(this) {
  before(function() { with(this) {
    this.stdout = new stream.PassThrough()
    this.cli = new CLI({
      argv:   ["node", "bin/github", "foo", "bar", "--limit", "1"],
      stdout: stdout
    })
    var fixture = fs.readFileSync(path.join(__dirname, "fixtures", "issues.json"), "utf8")
    expect(github, "getIssues").given("foo", "bar").yields([null, JSON.parse(fixture)])
  }})

  it("displays a limited number of issues", function(resume) { with(this) {
    cli.run(function() { stdout.end() })

    stdout.pipe(concat(function(output) {
      resume(function() {
        assertEqual(
          '1 open issues:\n' +
          '\n' +
          '#27: Forward networking error events to the websocket event listeners.\n' +
          'https://github.com/faye/faye-websocket-node/pull/27\n' +
          '[ramsperger] Thu, 24 Oct 2013 19:00:29 GMT\n' +
          '\n',
          output.toString("utf8") )
      })
    }))
  }})
}})
```

6.2.6. Standard input

Since we've used a `PassThrough` stream as a stand-in for standard output in our previous tests, it might be tempting to do the same thing for standard input, so we can simulate the user typing in answers to questions by writing to this stream. But this actually tends to be more hassle than it's worth; having access to the program's inner workings lets us manipulate it in more powerful and structured ways than those permitted by simple text streams. And in some cases, this approach is not even possible: some interactive input modules, such as the `pw`[20] module, do not work if the input stream is not a TTY.

You might be wondering why I've been referring to the object we pass to the `CLI` constructor as the 'user interface', since so far it's mostly been a clone of the `process` object. Well, it's also a place to put higher-level functions that describe user interactions — questions the program can ask, and ways the user can input information. Asking for a password is a good example of this.

The following executable uses the `pw` module to ask the user for a password. This is untestable since `pw` requires a TTY, so replacing it with a 'fake' stream is not an option. Running code that uses `pw` in tests will result in your tests hanging while waiting for a password that is not forthcoming. So, we leave some small amount of code in the executable to integrate with `pw` — just enough logic to handle asking the user for a password — while leaving the rest of the program logic in a `CLI` class where we can test it.

[20] https://npmjs.org/package/pw

Figure 6.29. node/password_cli/bin/authenticate

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

var pw  = require("pw"),
    CLI = require("../lib/cli")

var program = new CLI({
  stdout: process.stdout,

  askForPassword: function(callback) {
    process.stderr.write("Password: ")
    pw("*", process.stdin, process.stderr, callback)
  }
})

program.run(function(error) {
  if (!error) return
  console.error(error.message)
  process.exit(1)
})
```

The `CLI` class invokes the `askForPassword()` method on the `ui` object. In production, this will actually ask the user for a password, but in tests we can stub out `askForPassword()` to inject different user input into the program.

If the password is correct, this program prints a welcome message to stdout, but if the password is wrong then it yields an error. Again, we yield an error here to avoid having the `CLI` class call `process.exit()` and aborting our tests early.

Figure 6.30. node/password_cli/lib/cli.js

```
var CLI = function(ui) {
  this._ui = ui
}

CLI.prototype.run = function(callback) {
  var self = this

  this._ui.askForPassword(function(password) {
    if (password === "open sesame!") {
      self._ui.stdout.write("Come on in!\n")
      callback(null)
    } else {
      callback(new Error("ACCESS DENIED"))
    }
  })
}

module.exports = CLI
```

To test this class, we inject a `ui` object as before, and use stubs of `askForPassword()` to test each scenario: one scenario has the function yield the correct password, and one does not. This accurately mimics the response of the `pw` module when used for real. We can use this approach to fake out many other kinds of terminal user interactions, including ncurses-style interfaces or interactions with system services such as using the ssh-agent[21] module to sign something using the user's SSH keys. The function name should express the high-level purpose of the interaction, rather than the technical details of its implementation, to make it easy to write tests against.

[21] https://npmjs.org/package/ssh-agent

Figure 6.31. node/password_cli/spec/cli_spec.js

```
var JS     = require("jstest"),
    stream = require("stream"),
    CLI    = require("../lib/cli")

JS.Test.describe("Password CLI", function() { with(this) {
  before(function() { with(this) {
    this.ui  = {stdout: {}}
    this.cli = new CLI(ui)
  }})

  describe("when the user enters the right password", function() { with(this) {
    before(function() { with(this) {
      stub(ui, "askForPassword").yields(["open sesame!"])
    }})

    it("prints a welcome message", function(resume) { with(this) {
      expect(ui.stdout, "write").given("Come on in!\n")
      cli.run(resume)
    }})
  }})

  describe("when the user enters an incorrect password", function() { with(this) {
    before(function() { with(this) {
      stub(ui, "askForPassword").yields(["let me in!"])
    }})

    it("yields an error", function(resume) { with(this) {
      cli.run(function(error) {
        resume(function() { assertEqual( "ACCESS DENIED", error.message ) })
      })
    }})
  }})
}})
```

There's one interesting twist to this test, which is that we've used a bare object `{}` instead of a `PassThrough` stream to stand in for `process.stdout`. This is because the program's interaction with stdout is so simple that it can be tested with a single `expect(stdout, "write")` assertion. But also, the fact that we're using a bare object means we'll get an error if the program calls `stdout.write()` when it's not supposed to, since the object does not have a `write()` method. This would not be true if we used a stream object; the program could write to it when it's not supposed to and we would be none the wiser.