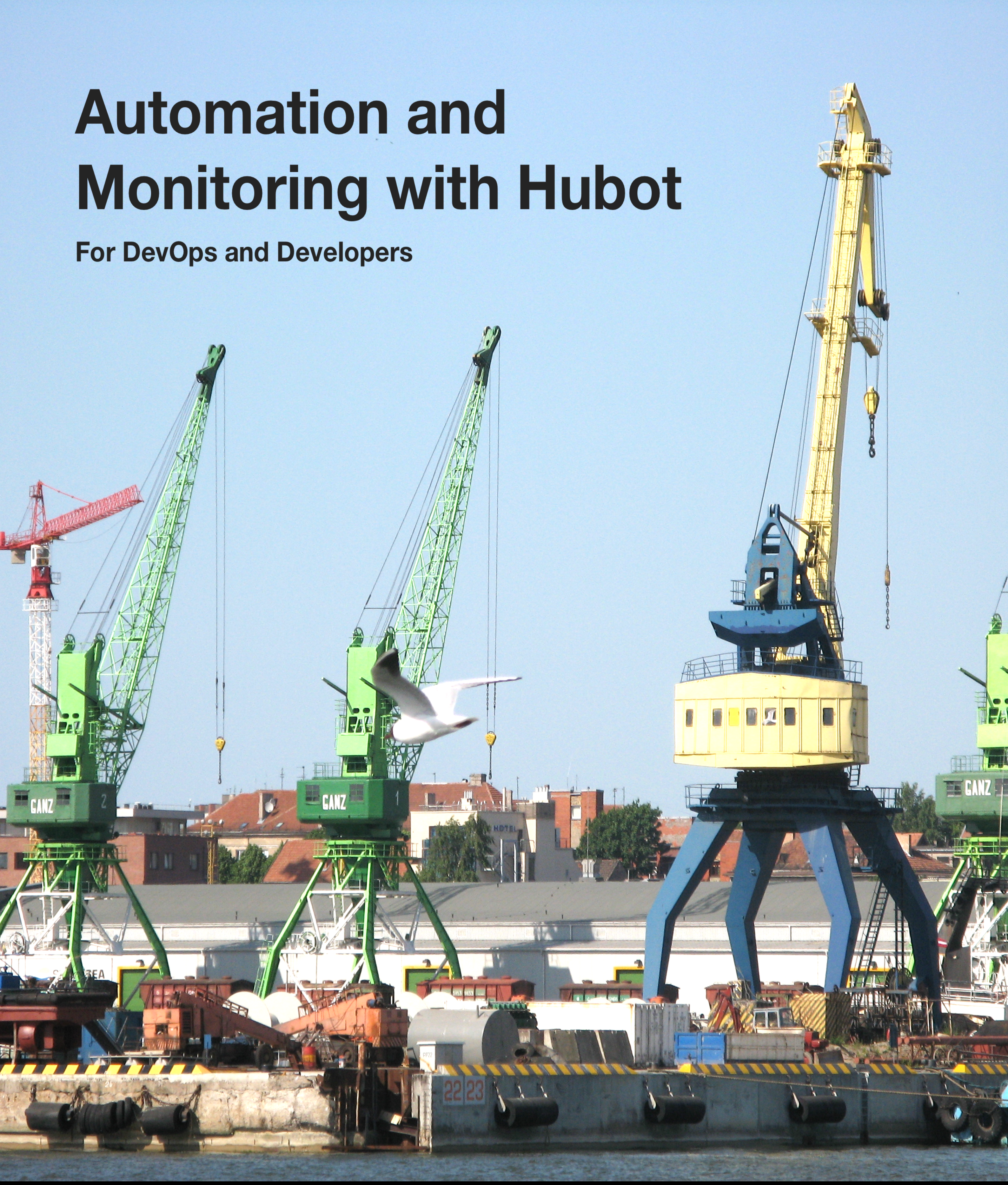
Automation and Monitoring with Hubot

For DevOps and Developers



Tomas Varaneckas

Automation and Monitoring with Hubot

For DevOps and Developers

Tomas Varaneckas

This book is for sale at http://leanpub.com/automation-and-monitoring-with-hubot

This version was published on 2014-09-29

Contents

Hubot Scripting

Hubot is written in Node.js[1], using CoffeeScript[2], which is a JavaScript wrapper that resembles Python and aims to remove the shortcomings of JavaScript and make use of it's wonderful object model. Following the tradition of Hubot, all scripts in this book will be written in CoffeeScript, but you may use JavaScript, simply name your scripts with `.js` rather than `.coffee` file extension.

Hello, World!

To start with, create `scripts/hello.coffee` in your hubot directory with following contents:

```
# Description:
#   Greet the world
#
# Commands:
#   hubot greet - Say hello to the world

module.exports = (robot) ->
  robot.respond /greet/i, (msg) ->
    msg.send "Hello, World!"
```

Now restart Hubot and try it out in your chatroom.

```
Tomas V.  hubot help greet
Hubot     hubot greet - Say hello to the world
Tomas V.  hubot greet
Hubot     Hello, World!
```

Wonderful, isn't it?

Basic Operations

Hubot is event driven, and when you write scripts for it, you define callbacks that should happen when some event occurs. Event can be:

- Message in the chatroom

[1] http://nodejs.org
[2] http://coffeescript.org/

- Private message to Hubot
- A text pattern detected in any message
- HTTP request

Callback can result in:

- Message in the chatroom
- Reply to a message
- Emotion in the chatroom
- HTTP response (if trigger was HTTP request)
- New HTTP request
- Executing a shell command
- Executing something on a remote server

Hubot can do anything that can be done with Node.js.

We'll learn how to exploit everything Hubot can offer by writing a fully functional script that covers a different piece of functionality. We will be analyzing it line by line, so you will get a perfectly clear understanding of what's happening.

Reacting To Messages In Chatroom

Let's try to create something more useful than hello world. We want Hubot to print out this month's calendar when we say "hubot calendar" or "hubot calendar me". We will use `cal` - a shell command that prints out a calendar like this:

```
hubot@botserv:~$ cal
    January 2014
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

To do that, we will create `calendar.coffee` in `scripts/` directory and use `robot.respond` to handle the event.

scripts/calendar.coffee

```
child_process = require('child_process')
module.exports = (robot) ->
  robot.respond /calendar( me)?/i, (msg) ->
    child_process.exec 'cal -h', (error, stdout, stderr) ->
      msg.send(stdout)
```

Let's analyze what happens line by line.

```
child_process = require('child_process')
```

Here we require child_process[3] - a node module for making system calls. We assign the module to `child_process` variable.

```
module.exports = (robot) ->
```

When Hubot requires `calendar.coffee`, `module.exports` is the object that gets returned. The `(robot) ->` part is a function that takes `robot` argument. This is how this line would look like in JavaScript:

```
module.exports = function(robot) {
```

Every Hubot script must export a function that takes `robot` argument and uses it to set up event listeners.

```
  robot.respond /calendar( me)?/i, (msg) ->
```

`robot.respond` is a function that takes two arguments - a regular expression to match the message, and a callback function that takes `msg` argument, which has a variety of functions for doing various actions. The regex `/calendar( me)?/i` would match `calendar` and `calendar me` in case insensitive fashion. Since we are using `respond`, it also expects the message to begin with `hubot`, or whatever your bot name is.

```
    child_process.exec 'cal -h', (error, stdout, stderr) ->
```

Here we call `exec` function on `child_process` variable, and provide two parameters - a system call that should be executed, and a callback function that takes 3 arguments - `error`, `stdout`, and `stderr`. `cal -h` displays ASCII calendar without highlighting current day.

[3]http://nodejs.org/api/child_process.html#child_process_child_process

```
      msg.send(stdout)
```

Finally, we use `msg`, which was passed into `robot.respond` callback function, to send standard output from `cal -h` command that we just executed.

To understand Hubot scripting better, you can try to understand concepts of Node.js. It's all about callbacks. In our calendar script there are two nested callbacks, one for `robot.respond`, another for `child_process.exec`.

Now restart Hubot and test the new script.

```
Tomas V.  hubot calendar
Hubot         January 2014
          Su Mo Tu We Th Fr Sa
                      1  2  3  4
           5  6  7  8  9 10 11
          12 13 14 15 16 17 18
          19 20 21 22 23 24 25
          26 27 28 29 30 31
```

It works as expected, but we also want this command to appear in `hubot help`, since it's not useful to have commands that nobody knows about. We have to add a documentation block on top of our script to get the effect. The final version of our script looks like this:

script/calendar.coffee

```
# Description:
#   Prints out this month's ASCII calendar.
#
# Commands:
#   hubot calendar [me] - Print out this month's calendar

child_process = require('child_process')
module.exports = (robot) ->
  robot.respond /calendar( me)?/i, (msg) ->
    child_process.exec 'cal -h', (error, stdout, stderr) ->
      msg.send(stdout)
```

Now `hubot help` and `hubot help calendar` will tell everyone about your script.

Reacting To Message Parts

Hubot can eavesdrop on chatrooms and react to certain words or phrases that were said without talking to the bot directly. Use `robot.hear` to do it.

Our new script will listen for "weather in <…>", query Open Weather Map[4] API and post the weather information.

[4]http://openweathermap.org/

script/weather.coffee

```coffee
# Description:
#   Tells the weather
#
# Configuration:
#   HUBOT_WEATHER_API_URL - Optional openweathermap.org API endpoint to use
#   HUBOT_WEATHER_UNITS - Temperature units to use. 'metric' or 'imperial'
#
# Commands:
#   weather in <location> - Tells about the weather in given location
#
# Author:
#   spajus

process.env.HUBOT_WEATHER_API_URL ||=
  'http://api.openweathermap.org/data/2.5/weather'
process.env.HUBOT_WEATHER_UNITS ||= 'imperial'

module.exports = (robot) ->
  robot.hear /weather in (\w+)/i, (msg) ->
    city = msg.match[1]
    query = { units: process.env.HUBOT_WEATHER_UNITS, q: city }
    url = process.env.HUBOT_WEATHER_API_URL
    msg.robot.http(url).query(query).get() (err, res, body) ->
      data = JSON.parse(body)
      weather = [ "#{Math.round(data.main.temp)} degrees" ]
      for w in data.weather
        weather.push w.description
      msg.reply "It's #{weather.join(', ')} in #{data.name}, #{data.sys.count\
ry}"
```

Run it for a test drive.

```
Tomas V.  I wonder what is the weather in Vilnius right now
Hubot     Tomas Varaneckas: It's 28 degrees, shower snow, mist in Vilnius, LT
Tomas V.  and weather in California?
Hubot     Tomas Varaneckas: It's 37 degrees, Sky is Clear in California, US
```

I wish I were in California right now. Anyway, let's take this script apart. We'll skip documentation, since it's pretty straightforward.

```
process.env.HUBOT_WEATHER_API_URL ||=
  'http://api.openweathermap.org/data/2.5/weather'
process.env.HUBOT_WEATHER_UNITS ||= 'imperial'
```

`process.env` allows you to read and set environmental variables, and our script uses a couple of them. One for defining the API endpoint, another one for measurment unit type. In CoffeeScript `x ||= y` is a shorthand for `x = (x != null) ? x : y`, meaning it will only set the variable if it has not been set before. This way you can override the values and set `HUBOT_WEATHER_UNITS=metric` to get Hubot tell degrees in Celsius rather than Farenheit.

```
  robot.hear /weather in (\w+)/i, (msg) ->
```

`robot.hear` works almost like `robot.respond`, with one exception. `robot.respond` requires message to begin with Hubot's name, while `robot.hear` reacts on any part of message, which is exactly what we want. It takes two arguments, a regex that matches "weather in

```
    city = msg.match[1]
```

`msg.match` is an array of regex matches, with `0` being the full message, and in our case `1` being the content of the parentheses, which is simply any word. Yes, this script will fail to work with "San Francisco". So, we set city to be the first word that comes after "weather in".

```
    query = { units: process.env.HUBOT_WEATHER_UNITS, q: city }
    url = process.env.HUBOT_WEATHER_API_URL
```

Here we construct a query string parameters that will be passed to the weather API, and set the URL we are going to call. We will read `units` from `HUBOT_WEATHER_UNITS` environmental variable, and set query to `city`. If we would construct the query string ourselves, we would need to worry about URL-encoding special characters, but since we're passing an object, it will be taken care of for us. Final request will be made to following url: `http://api.openweathermap.org/data/2.5/weather?units=imperial&q=chicago`.

```
    msg.robot.http(url).query(query).get() (err, res, body) ->
      data = JSON.parse(body)
```

Now we call the url using `HTTP GET`, set the query string parametrs using `.query()`, and provide a callback function to handle the response. Callback parameters are error (if any), HTTP response object and plain text response body. Our API returns JSON, so we parse the response body into `data` variable.

```
      weather = [ "#{Math.round(data.main.temp)} degrees" ]
```

Here we create a `weather` array with single element - `data.main.temp` is `{ main: { temp: ... } }` from the response JSON, and since it is returned in high precision, we round it to an integer with `Math.round`. And finally we make it a string with "degrees" at the end.

```
      for w in data.weather
        weather.push w.description
```

We loop `{ weather: [ ... ] }` from response JSON, getting the `description` out of every element and pushing it to the end of `weather` array.

```
      msg.reply "It's #{weather.join(', ')} in #{data.name}, #{data.sys.count\
ry}"
```

When we have our `weather` array all packed up with data, we join it into comma separated string and form a nice string containing the weather data, city name and country code.

Capturing All Messages

Sometimes you may want Hubot to process all messages in all chatrooms. For example, if you are writing a logging system. Here is a simple one:

scripts/logger.coffee

```
# Description
#   Logs all conversations
#
# Notes:
#   Logs can be found at bot's logs/ directory
#
# Author:
#   spajus

module.exports = (robot) ->
  fs = require 'fs'
  fs.exists './logs/', (exists) ->
    if exists
      startLogging()
    else
      fs.mkdir './logs/', (error) ->
        unless error
          startLogging()
        else
          console.log "Could not create logs directory: #{error}"
  startLogging = ->
    console.log "Started logging"
    robot.hear //, (msg) ->
      fs.appendFile logFileName(msg), formatMessage(msg), (error) ->
        console.log "Could not log message: #{error}" if error
  logFileName = (msg) ->
```

```
    safe_room_name = "#{msg.message.room}".replace /[^a-z0-9]/ig, ''
    "./logs/#{safe_room_name}.log"
  formatMessage = (msg) ->
    "[#{new Date()}] #{msg.message.user.name}: #{msg.message.text}\n"
```

The breakdown:

```
  fs = require 'fs'
```

We require Node's built in file system module[5] and assign it to `fs` variable.

```
  fs.exists './logs/', (exists) ->
```

We check if `./logs/` directory exists[6], and since NodeJS is asynchronous, we have to provide a callback function `(exists) ->`, that will get called with `true` or `false` after file system check actually happens.

```
    if exists
      startLogging()
    else
      fs.mkdir './logs/', (error) ->
        unless error
          startLogging()
        else
          console.log "Could not create logs directory: #{error}"
```

All this is happening in the `(exists) ->` callback function. If directory `./logs/` exists, we start logging by calling `startLogging()` function immediately, otherwise we call mkdir[7] to create this directory. It has another callback function, `(error) ->`. It gets called after directory creation is over. If there was no error, we call `startLogging()` function, otherwise we use `console.log` to inform that we failed to start logging because directory could not be created.

```
  startLogging = ->
    console.log "Started logging"
    robot.hear //, (msg) ->
```

This is the definition of `startLogging()` function we've called above. It uses `console.log` to announce that logging was initiated, then uses `robot.hear //, (msg) ->` to register a listener that reacts to all chat messages. That is because `robot.hear` does not require a message to be prefixed with `hubot`, and `//` is a regular expression that would match just anything.

[5]http://nodejs.org/api/fs.html
[6]http://nodejs.org/api/fs.html#fs_fs_exists_path_callback
[7]http://nodejs.org/api/fs.html#fs_fs_mkdir_path_mode_callback

```
    fs.appendFile logFileName(msg), formatMessage(msg), (error) ->
      console.log "Could not log message: #{error}" if error
```

When `robot.hear` gets triggered, `(msg) ->` is called, and this is what happens inside. We use appendFile[8] to create or append a file that `logFileName(msg)` function will return, and write the output of `formatMessage(msg)` function there. `appendFile` has a callback function to handle errors. We define it as `(error) ->` and use `console.log` to inform about the failure if `error` is present.

Time to try this out. After restarting Hubot, say something:

```
Tomas V.  Hello, anybody here?
          hubot ping
Hubot     PONG
Tomas V.  oh good, I hope you're not logging anything
```

It should appear in your Hubot's `logs/` directory:

```
hubot@botserv: ~/campfire$ cat logs/585164.log
[2014-03-22 21:54:26] Tomas Varaneckas: Hello, anybody here?
[2014-03-22 21:54:32] Tomas Varaneckas: hubot ping
[2014-03-22 21:54:47] Tomas Varaneckas: oh good, I hope you're not logging an\
ything
```

Unfortunately Hubot will not be able to see it's own messages. It can be done after tweaking the internals, but that's a whole different story. Other than that, all messages will get logged.

Capturing Unhandled Messages

If you want to capture only those messages that were not handled by any Hubot script, it's very simple to do:

```
module.exports = (robot) ->
  robot.catchAll (msg) ->
    msg.send "I don't know how to react to: #{msg.message.text}"
```

Serving HTTP Requests

Hubot has a built-in express[9] web framework that can serve HTTP requests. By default it runs on port `8080`, but you can change the value using `PORT` environmental variable. This time we will create a script that responds to HTTP requests and posts request body in one or more rooms. We'll name this script `notifier.coffee`.

It will accept HTTP POST requests, so there will be no limits for what the body can be.

[8]http://nodejs.org/api/fs.html#fs_fs_appendfile_filename_data_options_callback
[9]http://expressjs.com

scripts/notifier.coffee

```
# Description:
#   Send message to chatroom using HTTP POST
#
# URLS:
#   POST /hubot/notify/<room> (message=<message>)

module.exports = (robot) ->
  robot.router.post '/hubot/notify/:room', (req, res) ->
    room = req.params.room
    message = req.body.message
    robot.messageRoom room, message
    res.end()
```

To try it out, we will make a `POST` request using `curl`.

```
hubot@botserv:~$ curl -X POST \
                      -d message="Hello from $(hostname) shell" \
                      http://localhost:8080/hubot/notify/585164
```

And we get this in our chatroom.

```
Hubot   Hello from botserv shell
```

Let's dig in to the source.

```
  robot.router.post '/hubot/notify/:room', (req, res) ->
```

`robot.router.post` creates a listener for HTTP POST requests to `/hubot/notify/:room` URL, where `:room` is a variable defining your room. It also takes a callback function that has two parameters, request and response. You can find out everything about `robot.router` by examiming express api documentation[10] - `robot.router` is the express `app`.

```
    room = req.params.room
```

`req.params` contains params from the URL, so in our case, if URL is `/hubot/notify/123`, variable `room` is set to `123`.

```
    message = req.body.message
```

We read the value of `message` POST parameter and assign it to `message` variable.

[10]http://expressjs.com/api.html

```
    robot.messageRoom room, message
    res.end()
```

Now, we send the `message` to given `room` and end the HTTP response. It would work without `res.end()`, but it's always nice to respond to the request, otherwise the HTTP client may hang while expecting a response.

While this script looks nothing important, this concept is incredibly useful in building your own chat based monitoring. You can trigger any sort of events from anywhere and make Hubot tell everything about it by doing an HTTP request.

Cross Script Communication With Events

To reduce script complexity, or to introduce communication between two or more scripts, one can use Hubot event system, which consists of two simple functions: `robot.emit event, args` and `robot.on event, (args) ->`. We will now write two scrips - `event-master.coffee` and `event-slave.coffee`. Master will listen to us and trigger events that Slave will listen to and process.

scripts/event-master.coffee

```
# Description:
#   Controls slave at event-slave.coffee
#
# Commands:
#   hubot tell slave to <action> - Emits event to slave to do the action

module.exports = (robot) ->
  robot.respond /tell slave to (.*)/i, (msg) ->
    action = msg.match[1]
    room = msg.message.room
    msg.send "Master: telling slave to #{action}"
    robot.emit 'slave:command', action, room
```

scripts/event-slave.coffee

```
# Description:
#   Executes commands from `event-master.coffee`

module.exports = (robot) ->
  robot.on 'slave:command', (action, room) ->
    robot.messageRoom room, "Slave: doing as told: #{action}"
    console.log 'Screw you, master...'
```

It runs like this:

```
Tomas V.  hubot tell slave to bring beer
Hubot     Slave: doing as told: bring beer
Hubot     Master: telling slave to bring beer
```

Meanwhile in `hubot.log`:

hubot.log

```
Screw you, master...
```

Notice that “Slave” responded before “Master”, even though `msg.send` was called in “Master” script first. It’s a perfect example to help you understand how Node.js works. Nearly everything is being done asynchronously using callback functions, the only way to ensure the order of execution is to use callbacks. To make “Master” send his message first, we have to put `robot.emit` in `msg.send` callback in `event-master.coffee`, like this:

```
msg.send "Master: telling slave to #{action}", ->
  robot.emit 'slave:command', action, room
```

This way `msg.send` is execute first, and only when it’s done, the callback function is called and `robot.emit` gets executed.

In `robot.emit` call, `slave:command` is just a string that describes the event, `action` and `room` are the parameters that are passed along with the event trigger. There can be as many listeners as needed for every event type. We have placed ours in `event-slave.coffee`:

```
  robot.on 'slave:command', (action, room) ->
    robot.messageRoom room, "Slave: doing as told: #{action}"
    console.log 'Screw you, master...'
```

Our callback function is pretty simple, it just posts a message to given room, and logs “Screw you, master...” behind everyone’s back using `console.log`. It’s a good technique for debugging your scripts.

Periodic Task Execution

You can make Hubot execute something using node-cron[11], which works perfectly with combination of firing events - let one of your scripts listen to an event, and another one fire them periodically.

First install the dependencies in your Hubot directory:

[11]https://github.com/ncb000gt/node-cron

```
hubot@botserv:~campfire$ npm install --save cron time
```

Then create a script called `scripts/cron.coffee` and define all periodic executions there:

scripts/cron.coffee

```
# Description:
#   Defines periodic executions

module.exports = (robot) ->
  cronJob = require('cron').CronJob
  tz = 'America/Los_Angeles'
  new cronJob('0 0 9 * * 1-5', workdaysNineAm, null, true, tz)
  new cronJob('0 */5 * * * *', everyFiveMinutes, null, true, tz)

  room = 12345678

  workdaysNineAm = ->
    robot.emit 'slave:command', 'wake everyone up', room

  everyFiveMinutes = ->
    robot.messageRoom room, 'I will nag you every 5 minutes'
```

Now let's break it down:

```
  cronJob = require('cron').CronJob
  tz = 'America/Los_Angeles'
  new cronJob('0 0 9 * * 1-5', workdaysNineAm, null, true, tz)
  new cronJob('0 */5 * * * *', everyFiveMinutes, null, true, tz)
```

Here we require the `cron` dependency and assign it's `CronJob` prototype to `cronJob` variable and assign our desired time zone to `tz`. Then we create two jobs, first will run every workday at 9 AM in Los Angeles time and will execute `workdaysNineAm` function. The other one will execute every five minutes and call `everyFiveMinutes` function.

```
  room = 12345678

  workdaysNineAm = ->
    robot.emit 'slave:command', 'wake everyone up', room

  everyFiveMinutes = ->
    robot.messageRoom room, 'I will nag you every 5 minutes'
```

We assign room id to `room` variable, which we will use in following functions. `workdaysNineAm` emits an event for the slave script we created earlier, and `everyFiveMinutes` just posts a message to a room.

You can also do the same automation using your OS cron that would run `curl` on Hubot's HTTP endpoints, but this is more elegant.

Debugging Your Scripts

It’s frustrating when things don’t work the way they should, but there are several techniques to help you narrow down the problem.

Log strings to `hubot.log`:

```
console.log "Something happened: #{this} and #{that}"
```

Inspect an object and print it in the chatroom:

```
util = require('util')
msg.send util.inspect(strange_object)
```

Recover from an error and log it:

```
try
  dangerous.actions()
catch e
  console.log "My script failed", e
```

Advanced Debugging With Node Inspector

Sometimes it’s not enough just to print out the errors. For those occasions you may need heavy artillery - a full fledged debugger. Luckily, there is node-inspector[12]. You will be especially happy with it if you are familiar with Chrome’s web inspector. To use `node-inspector`, first install the `npm` package. You should do it once, using `-g` switch to install it globally. Install as root.

```
root@botserv:~# npm install -g node-inspector
```

To start the debugger, run `node-inspector` either in the background (followed by `&`) or in a new shell. In following example it’s started without preloading all scripts (otherwise it’s a long wait), and inspector console running on port `8123`, because both `hubot` and `node-inspector` use port `8080` by default. We could set `PORT=8123` for `hubot` instead, but setting it for `node-inspector` is more convenient.

```
hubot@focus:~/campfire$ node-inspector --no-preload --web-port 8123
Node Inspector v0.7.0-1
   info  - socket.io started
   Visit http://127.0.0.1:8123/debug?port=5858 to start debugging.
```

Now, we will put `debugger` to add a breakpoint to our `weather.coffee` script and debugger will stop on that line when it gets executed.

[12]https://github.com/node-inspector/node-inspector

script/weather.coffee

```
      for w in data.weather
        weather.push w.description
      debugger
      msg.reply "It's #{weather.join(', ')} in #{data.name}, #{data.sys.count\
ry}"
```

Now we have to start Hubot in a little different way:

```
hubot@focus:~/campfire$ coffee --nodejs --debug node_modules/.bin/hubot
debugger listening on port 5858
```

Then open `http://127.0.0.1:8123/debug?port=5858` - the link that `node-inspector` gave you in it's output in *Chrome*, or any other Blink based browser. Expect a little delay, because it will load all the scripts that Hubot normally requires just in time when needed. When you are able to see Sources tree in the top-left corner of your browser (you may need to click on the icon to expand it), get back to Hubot console and ask for the weather:

```
Hubot> what is the weather in Hawaii?
Hubot>
```

Don't expect a response, because Chrome should now switch to `weather.coffee` and stop the execution at `debugger` line. Now you can step over the script line by line, add additional breakpoints by clicking on line nubers in any souce file from the Source tree in the left, or use the interactive console - there is Console tab at the top of the debugger, and a small › icon in bottom-left corner, which I prefer because it doesn't close the source view.

You can type any JavaScript in the console, and it will execute. Let's examine our `weather` array:

```
> weather
  - Array[2]
    0: "74 degrees"
    1: "broken clouds"
    length: 2
```

And the response from the weather API:

```
> data
  - Object
    base: "cmc stations"
    + clouds: Object
    cod: 200
    + coord: Object
    dt: 1389847230
    id: 5856195
    + main: Object
    name: ""
    - sys: Object
      country: "United States of America"
      message: 0.308
      sunrise: 1389892287
      sunset: 1389931892
    - weather: Array[1]
      - 0: Object
        description: "broken clouds"
        icon: "04n"
        id: 803
        main: "Clouds"
      length: 1
    + wind: Object
```

You can expand any part of the object tree to see what's in it. You can also call functions:

```
> msg.send("Hello from node-inspector")
```

And in Hubot shell you should see:

```
Hubot> Hello from node-inspector
```

You can debug your web applications or any other JavaScript or CoffeeScript code using this technique. It's even easier for web applications - just open Chrome Inspector and you're set.

Writing Unit Tests For Hubot Scripts

Unit tests for Hubot scripts are a tricky subject that is either misunderstood or avoided. There is a strange trend among packages in github.com/hubot-scripts[13] to write tests like this one:

[13]https://github.com/hubot-scripts

```
chai = require 'chai'
sinon = require 'sinon'
chai.use require 'sinon-chai'

expect = chai.expect

describe 'hangouts', ->
  beforeEach ->
    @robot =
      respond: sinon.spy()
      hear: sinon.spy()

    require('../src/hangouts')(@robot)

  it 'registers a respond listener', ->
    expect(@robot.respond).to.have.been.calledWith(/hangout/)
```

You can find this test at hubot-scripts/hubot-google-hangouts[14]. This test checks that Hubot script compiles and that it has the following lines:

```
module.exports = (robot) ->
  robot.respond /hangout( me)?\s*(.+)?/, (msg) ->
```

That's better than nothing, but still a bit pointless, don't you think? Luckily, there are better ways to do this. Take a look at hubot-mock-adapter[15]. Tests will certainly be more difficult to write, but they would actually test the script itself, not just the fact that it gets loaded.

To see an example of `hubot-mock-adapter` in action, take a look at tests of hubot-pubsub[16].

Since unit testing is a vast subject and it can take another book to fully cover, we're not going to dig any deeper.

Hubot Script Template

You can use this template as a starting point for your new Hubot scripts. It is taken from Hubot Control[17], which also gives you a web based IDE for quick scripting.

[14]https://github.com/hubot-scripts/hubot-google-hangouts/
[15]https://github.com/blalor/hubot-mock-adapter
[16]https://github.com/hubot-scripts/hubot-pubsub/blob/master/spec/pubsub_spec.coffee
[17]https://github.com/spajus/hubot-control

scripts/template.coffee

```coffee
# Description
#   <description of the scripts functionality>
#
# Dependencies:
#   "<module name>": "<module version>"
#
# Configuration:
#   LIST_OF_ENV_VARS_TO_SET
#
# Commands:
#   hubot <trigger> - <what the respond trigger does>
#   <trigger> - <what the hear trigger does>
#
# URLS:
#   GET /path?param=<val> - <what the request does>
#
# Notes:
#   <optional notes required for the script>
#
# Author:
#   <github username of the original script author>

module.exports = (robot) ->

  robot.respond /jump/i, (msg) ->
    msg.emote "jumping!"

  robot.hear /your'e/i, (msg) ->
    msg.send "you're"

  robot.hear /what year is it\?/i, (msg) ->
    msg.reply new Date().getFullYear()

  robot.router.get "/foo", (req, res) ->
    res.end "bar"
```

This is how these examples look in action:

```
Tomas V.  hubot jump
Hubot     *jumping!*
Tomas V.  wow, your'e amazing
Hubot     you're
Tomas V.  anybody knows what year is it?
Hubot     Tomas Varaneckas: 2014
```

To check HTTP response, we'll use `curl`:

```
hubot@botserv:~$ curl http://localhost:8080/foo
bar
```

Using Hubot Shell Adapter For Script Development

You may find it inconvenient to restart Hubot every time you change your script. In many cases you can test your work using built-in shell adapter, like this:

```
hubot@botserv:~/campfire$ PORT=8888 bin/hubot
[Fri Jan 10 2014 01:35:37 GMT-0500 (EST)] INFO Data for brain retrieved from \
Redis
Hubot> hubot help greet
Hubot> Hubot greet - Say hello to the world
Hubot> hubot greet
Hubot> Hello, World!
Hubot> exit
```

In this example we set `PORT=8888` to avoid "Address already in use" error if Hubot is alread running as a service.

Developing Scripts With Hubot Control

If you use Hubot Control, you can develop scripts with it's web based editor, which offers syntax checking and highlighting, integration with `git`, and a way to restart Hubot without logging in to the server.

Learning More

We've scratched the surface of what you can do with Hubot. One of the best ways to learn more about writing Hubot scripts by studying the source code of existing ones. Best places to start:

- The old script catalog: https://github.com/github/hubot-scripts
- The new script packages: https://github.com/hubot-scripts

Throughout the rest of the book we will cover a number of use cases of integrating Hubot with a variety of applications and web services. You will learn how to make Hubot an invaluable addition to your DevOps stack.