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

Table of Contents

A Word From The Author

Dear Reader,

Thank you for purchasing this book. It is my first book, I’m releasing it without a team of editors and publishers to back me up, therefore I must apologize for bad writing style and terrible grammar you will most certainly encounter multiple times. I hope that the value of contents will outweigh the lack of excellence in style, and that after reading this book you will fall in love with Hubot as much as I did.

Since I am a software developer, I treat this book as I would treat a software project. I’m releasing it early and will release often. Thanks to Leanpub for making it possible.

If you won’t find something you expected to see in this book, don’t get disappointed. I will be happy to expand it with your requests. Contact me and I will do my best to add new content to this book, as long as it’s related to Hubot and doesn’t require purchasing any commercial licenses. I believe that software should be free.

Your feedback and suggestions are highly anticipated. You can find me here:

```
Email:    tomas.varaneckas@gmail.com
Blog:     http://varaneckas.com
GitHub:   https://github.com/spajus
LinkedIn: https://www.linkedin.com/in/spajus
```

Happy Hubotting!

Tomas Varaneckas

Preface

> “The future is already here - it’s just not very evenly distributed.”
>
> `-- William Gibson`

If you are a software developer, you know the importance of feedback. Did the tests pass? Was the build successful? Has deployment started? Is deployment completed? Did it fail? Has the error rate increased in production? What are those errors?

In a fast paced, agile environment, you are responsible for keeping an eye on all these things, checking the build server, reading logs, having multiple browser tabs open, each with different things you want to check. You do all of that multiple times a day, and it consumes your precious time.

You want to fix issues as they appear, before your users start noticing.

You also do various chores - log to remote machines to run scripts, start deployments, do same things over and over again.

And you probably receive a lot of emails. Build server writes you more than any of your best buddies, and your girlfriend keeps asking who the hell is that Cron Daemon and why she keeps writing you day and night.

If only there was some robot that could help you carry your daily burdens, reduce the need to proactively look after all those important things, and would be ready to execute any mundane task you command it to.

Let me introduce you to Hubot - a friendly chat robot that can do all these things, plus show you animated pictures of cats and squirrels now and then.

Run, Hubot, Run!

An obvious first step of our journey is to make Hubot run. We will be focusing on Linux, so if you want to deploy it elsewhere, refer to the official documentation.

The Operating System

In this book we will be running everything on fresh installation of Debian 7 (Wheezy). Since Debian is the mothership of many popular Linux flavors, including Ubuntu, you should have no trouble doing the same with any distribution deriving from Debian.

Installing Dependencies

You will need node.js and npm. You may prefer to install it via your distribution's package manager, which is probably the easiest way, but to make sure we have the latest and greatest, we will build it from source.

Following installation is performed as `root` user. If you are using Ubuntu and having trouble becoming `root`, simply run `sudo su`. Or keep adding `sudo` to commands that require elevated permissions.

Compiling Node.js

First, make sure you have the build tools.

```
apt-get install python g++ make checkinstall
```

Now, get the latest source and extract it.

```
cd /usr/local/src
wget -N http://nodejs.org/dist/node-latest.tar.gz
tar xzvf node-latest.tar.gz
```

Enter the directory you just extracted. Remember the version number, you will need it for compiling. At the moment of writing it was `0.10.24`.

```
root@botserv:/usr/local/src# ls
node-latest.tar.gz  node-v0.10.24
root@botserv:/usr/local/src# cd node-v0.10.24/
root@botserv:/usr/local/src/node-v0.10.24#
```

Configure and install it.

```
./configure
checkinstall -y --install=no --pkgversion 0.10.24 # Set correct version number
dpkg -i node_*
```

At this point you should be able to run `node`.

```
root@botserv:~# node -v
v0.10.24
```

Installing NPM

To install `npm`, just `curl` the installation script and pipe it to `sh`. If you may want to make sure what the script does before executing it, first pipe it to `less`.

```
curl https://npmjs.org/install.sh | sh
```

Test if it worked.

```
root@botserv:~# npm -v
1.3.22
```

Installing CoffeeScript

Hubot is written in CoffeeScript, so let’s install that too.

```
npm install -g coffee-script
```

Installing Redis Server

There is one last thing you will need. Hubot needs a “brain” to store the data, and it needs Redis to work out of the box.

```
apt-get install redis-server
```

Make sure it’s running.

```
root@botserv:~# redis-cli
redis 127.0.0.1:6379> exit
```

Installing Hubot

Use npm to install Hubot

```
npm install -g hubot
```

You should now have `hubot` command available.

```
root@botserv:~# hubot -v
2.7.1
```

Creating System User For Hubot

Now, we are almost there, but from this point we don’t want to run things as `root` due to security concerns. If anyone manages to breach into your system using a security hole in Hubot or one of it’s scripts, the perpetrator will be isolated to Hubot’s system user.

Let’s create one. You may want to choose to use the default home directory to keep and run your Hubot, because there will be less hassle with permissions.

```
useradd --home-dir /home/hubot --create-home --shell /bin/bash hubot
```

Become the user and see if you can still invoke `hubot` command.

```
root@botserv:~# su - hubot
hubot@botserv:~$ hubot -v
2.7.1
```

Since hubot user has no password, you can only log in to it using `su` from a priviliged account. From non `root` user you can do `sudo su - hubot`.

```
spajus@botserv:~$ sudo su - hubot
[sudo] password for spajus:
hubot@botserv:~$
```

Henceforth we will be performing shell commands using `hubot` user rather than `root`, unless explicitly stated otherwise.

Creating Your First Hubot

Installing `hubot` package did not provide you an instance of Hubot yet, so you will have to create one. First, examine what `hubot` command gives you.

```
hubot@botserv:~$ hubot --help
Usage hubot [options]

Available options:
  -a, --adapter ADAPTER   The Adapter to use
  -c, --create PATH       Create a deployable hubot
  -d, --disable-httpd     Disable the HTTP server
  -h, --help              Display the help information
  -l, --alias ALIAS       Enable replacing the robot's name with alias
  -n, --name NAME         The name of the robot in chat
  -r, --require PATH      Alternative scripts path
  -v, --version           Displays the version of hubot installed
```

The most interesting part of the output is the `--adapter` option. We will explore several adapters in later chapters, but for now we will stick to Campfire, which is built-in and will need no extra effort to run.

Now, create your first Hubot instance.

```
hubot@botserv:~$ hubot --create first-hubot
Creating a hubot install at first-hubot
<some output omitted>
Renaming /home/hubot/first-hubot/gitignore -> /home/hubot/first-hubot/.gitign\
ore
```

To see if it works, start it in shell mode and run a `hubot ping` command.

```
hubot@botserv:~$ cd first-hubot/
hubot@botserv:~/first-hubot$ bin/hubot
<npm will install dependencies on first run, output omitted>
Hubot> [Sat Jan 04 2014 01:55:33 GMT-0500 (EST)] WARNING The HUBOT_AUTH_ADMIN\
 environment variable
not set
Hubot> [Sat Jan 04 2014 01:55:33 GMT-0500 (EST)] INFO Initializing new data f\
or brain
Hubot> hubot ping
Hubot> PONG
Hubot> exit
hubot@botserv:~/first-hubot$
```

You may see an error like this being printed in the output:

```
ERROR [Error: Redis connection to localhost:6379 failed - connect ECONNREFUSE\
D]
```

It means you don’t have `redis-server` installed or running. See the previous paragraph or choose a different brain implementation if you don’t want to use Redis for Hubot. You can edit `hubot-scripts.json` and replace `redis-brain.coffee` with `file-brain.coffee`.

Hubot will then use `brain-dump.json` file to save it's memory. If you don't provide any brain script at all, Hubot will still run, but it will lose all it's memory after every restart, and since many scripts rely on the brain to store their state, brainless Hubot is not suitable for production use.

Exploring Your Hubot Instance

Let's explore the newly created `first-hubot` directory and see what's in there.

We will use `tree` command to draw ascii tree of the directory contents. You can get it with `apt-get install tree`.

```
hubot@botserv:~$ tree first-hubot/
first-hubot/
├── bin
│   ├── hubot
│   └── hubot.cmd
├── external-scripts.json
├── hubot-scripts.json
├── node_modules
│   └── <output omitted>
├── package.json
├── Procfile
├── README.md
└── scripts
    ├── auth.coffee
    ├── events.coffee
    ├── google-images.coffee
    ├── help.coffee
    ├── httpd.coffee
    ├── maps.coffee
    ├── ping.coffee
    ├── pugme.coffee
    ├── roles.coffee
    ├── rules.coffee
    ├── storage.coffee
    ├── translate.coffee
    └── youtube.coffee
```

- `bin/` contains executables - `hubot` is for Linux, Mac and alike, and `hubot.cmd` is for Windows.
- `external-scripts.json` is the place where you define custom scripts that will be used with Hubot. It can be either a path to script, or npm package name.
- `hubot-scripts.json` holds a list of scripts you want to include from GitHub's hubot-scripts package, which holds hundreds of scripts contributed by the community.
- `node_modules/` holds the dependencies that are checked and updated by `npm` on every run, or by doing `npm install`.
- `package.json` is the NPM package definition. It holds the information about all the dependencies that your Hubot needs to run.
- `Procfile` contains launch instructions for Heroku. You can use foreman to run it in your own environment.
- `README.md` holds a short introduction about Hubot itself, along with instructions how to deploy your bot to Heroku, configure Campfire, etc.

- `scripts/` is a place to put your custom scripts. Any script placed there will be autoloaded and available after relaunching your Hubot. It contains some goodies already, though you may want to get rid of or modify some of them. For instance, `scripts/ping.coffee` contains `hubot die` command, which you don’t want to have unless your Hubot restarts after exiting. Somebody in the chat will definitely try to run it once in a while.

Configuring Adapters

In order to connect your Hubot to an actual chat server, you need to configure an adapter. You have already encountered the default `shell` adapter when you ran `bin/hubot` to get the text prompt. Now we will look through most of available adapters and learn to configure them.

Choosing The Chat Service

If you're in a stage of making a decision which chat service should you choose for your DevOps Chat, the following table may help you make the right choice.

Chat Service	Adapter	Integration	Stability	Embedded Images	Security	Price
Campfire	campfire	Very easy	Very stable	Yes	SSL	Free for 4 $12- $99/mont
HipChat	hipchat	Easy	Stable	Yes	SSL	Free for 5 $2/user/m
IRC	irc	Intermediate	Stable	No	SSL	Free
Skype	skype	Difficult	Problematic	No	Proprietary	Free
Jabber/XMPP	xmpp	Intermediate	Stable	Via plugins	SSL	Free
Slack	slack,xmpp	Intermediate	Stable (xmpp)	Yes	SSL	Free, $8- 99/user/m

First, let's find out how to connect to your favorite chat service, and then we will put everything together into a deployable, bulletproof solution.

You should know that you have to register a user for Hubot for the service you want to use. This book does not cover chat service account registrations. IRC is an exception, where only a nickname is enough, yet you still can register it so nobody else can take it.

Campfire

If you're wondering where is Campfire's free plan - as of the writing of this book it was hidden - you had to start a basic 30-day trial and then go to Account page and downgrade your plan to Free.

Campfire is the native habitat of Hubot - it has a built in adapter and perfect integration. When registering an account for your Hubot, don't worry that `hubot` username is taken, your bot will respond to whatever alias you give it.

To get Hubot on Campfire room, first create a regular user account for it, then run `bin/hubot -a campfire` and provide three environmental variables:

- HUBOT_CAMPFIRE_ACCOUNT - Your account name that appears in URL. I've registered Hubotorium, and my chat URL is https://hubotorium.campfirenow.com, therefore account is hubotorium.
- HUBOT_CAMPFIRE_TOKEN - API authentication token that you can find in your account info. You can find it by clicking My info, it looks like this: 3a5c1b47dd76db2e950fce12ac62d4f555a17961.
- HUBOT_CAMPFIRE_ROOMS - comma separated list of room IDs you want Hubot to join. You can find out your room id by entering it in your browser, it appears in the url: https://hubotorium.campfirenow.com/room/585163 - it's the number 585163 in my case. If there are two rooms, you would list them like this: 585163,585164

We can use your first-hubot instance to go to Campfire right away.

```
hubot@botserv:~/first-hubot$ HUBOT_CAMPFIRE_ACCOUNT=hubotorium \
 HUBOT_CAMPFIRE_TOKEN=3a5c1b47dd76db2e950fce12ac62d4f555a17961 \
 HUBOT_CAMPFIRE_ROOMS=585163,585164 \
 bin/hubot --adapter campfire --name hubot
```

Your bot should join the rooms right away. If it doesn't here are a couple of tips for troubleshooting problems.

Troubleshooting

If you provide bad HUBOT_CAMPFIRE_ACCOUNT, the error looks like this:

```
ERROR Campfire HTTPS status code: 404
ERROR Campfire HTTPS response data:
```

Bad HUBOT_CAMPFIRE_ROOMS will give you this:

```
ERROR Campfire HTTPS status code: 404
ERROR Campfire HTTPS response data:
ERROR Campfire error on room 58516: Access Denied User
ERROR Streaming connection closed for room 58516. :(
```

And incorrect HUBOT_CAMPFIRE_TOKEN will show no signs of error, but Hubot will simply not appear in your chatrooms.

HipChat

Before proceeding, log in to HipChat with your bot's user and set the @mention name to @hubot in account settings. HipChat adapter ignores the name setting you provide when starting Hubot, instead it reacts to whatever the @mention name is.

HipChat adapter requires a couple of extra system dependencies, so make sure you install them first. Do it with sudo or as root.

```
root@botserv:~# apt-get install libexpat1-dev libicu-dev
```

Then, back as hubot user, install hubot-hipchat node module.

```
hubot@botserv:~/first-hubot$ npm install --save hubot-hipchat
```

--save option tells npm to update package.json, you can look what was added in there.

```
hubot@botserv:~/first-hubot$ grep hubot-hipchat package.json
    "hubot-hipchat": "~2.6.4"
```

Dependencies are now settled.

There are two environmental variables you have to provide for HipChat adapter:

- `HUBOT_HIPCHAT_JID` - Jabber ID of Hubot’s HipChat account. You can find it in Account settings, under XMPP/Jabber info. It should look like `12345_678901@chat.hipchat.com`.
- `HUBOT_HIPCHAT_PASSWORD` - The password of Hubot’s HipChat account.

Aditional configuration options are available, you can find them documented pretty well at https://github.com/hipchat/hubot-hipchat.

Successful connection to HipChat should look like this:

```
hubot@botserv:~/first-hubot$ HUBOT_HIPCHAT_JID=12345_678901@chat.hipchat.com \
 HUBOT_HIPCHAT_PASSWORD=MySuperSecretPassword123 \
 bin/hubot -a hipchat
INFO Connecting HipChat adapter…
INFO Connected to hipchat.com as @hubot
INFO Joining 12345_hubotorium@conf.hipchat.com
```

Now your bot should join all rooms, say `@hubot help` to interact with it.

Troubleshooting

Providing incorrect `HUBOT_HIPCHAT_JID` or `HUBOT_HIPCHAT_PASSWORD` will result in Hubot hanging on the following message:

```
INFO Connecting HipChat adapter…
```

IRC

To install IRC Hubot adapter, run:

```
hubot@botserv:~/first-hubot$ npm install --save hubot-irc
```

It now appears in your `package.json`:

```
hubot@botserv:~/first-hubot$ grep hubot-irc package.json
    "hubot-irc": "~0.1.24"
```

Configuration is done using the following environmental variables:

- `HUBOT_IRC_SERVER` - IRC server address, i.e.: `irc.freenode.net`
- `HUBOT_IRC_PORT` - IRC server port number
- `HUBOT_IRC_ROOMS` - comma separated list of IRC channels to join, i.e.: `#hubotorium,#hubot`
- `HUBOT_IRC_NICK` - IRC nickname for your bot
- `HUBOT_IRC_USESSL` - Set to `true` to use SSL encryption. You really want that.
- `HUBOT_IRC_SERVER_FAKE_SSL` - Set to `true` if you use private server with self signed SSL certificate.
- `HUBOT_IRC_UNFLOOD` - When set to `true`, Hubot will throttle it’s messages, so IRC server doesn’t kick it out for flooding.

More configuration options are described in https://github.com/nandub/hubot-irc.

To connect to Freenode using SSL, you would have to do this:

```
hubot@botserv:~/first-hubot$ HUBOT_IRC_SERVER=irc.freenode.net \
 HUBOT_IRC_PORT=6697 \
 HUBOT_IRC_ROOMS="#hubotorium" \
 HUBOT_IRC_NICK=hubot5000 \
 HUBOT_IRC_UNFLOOD=true \
 HUBOT_IRC_USESSL=true \
 bin/hubot --adapter irc
hubot5000 has joined #hubotorium
hubot5000 has joined #hubot
```

You can use tab completion to call your Hubot:

```
<spajus> hubot5000, the rules
<hubot5000> 1. A robot may not injure a human being or, through inaction, all\
ow a human being to come to harm.
<hubot5000> 2. A robot must obey any orders given to it by human beings, exce\
pt where such orders would conflict with the First Law.
<hubot5000> 3. A robot must protect its own existence as long as such protect\
ion does not conflict with the First or Second Law.
```

Troubleshooting

IRC adapter is hard to troubleshoot, since it gives no output if something goes wrong and just hangs indefinitely. You have to make sure server host and port are defined correctly, and the nickname you are trying to use is not taken by anyone.

XMPP/Jabber

The XMPP/Jabber adapter has the same prerequirements as HipChat adapter - use `sudo` or `root` user to install the dependencies. You can also use it to connect to Slack, and it will work better than their own native adapter.

```
root@botserv:~# apt-get install libexpat1-dev libicu-dev
```

Then install `hubot-xmpp` node module.

```
hubot@botserv:~/first-hubot$ npm install --save hubot-xmpp
```

Check if it was added to your `package.json`:

```
hubot@botserv:~/first-hubot$ grep hubot-xmpp package.json
    "hubot-xmpp": "~0.1.8"
```

These are the environmental variables you may need to set:

- `HUBOT_XMPP_HOST` - hostname of XMPP server. Can be an IP address, or a domain name.
- `HUBOT_XMPP_PORT` - XMPP server port. Usually `5222`.
- `HUBOT_XMPP_USERNAME` - bot username in `user@server` format.
- `HUBOT_XMPP_ROOMS` - comma separated list of rooms, in `room@conference.server` format.

More options are documented at https://github.com/markstory/hubot-xmpp.

Let's run it.

```
hubot@botserv:~/first-hubot$ HUBOT_XMPP_USERNAME="hubot@botserv" \
 HUBOT_XMPP_PASSWORD="SuperSecret123" \
 HUBOT_XMPP_ROOMS="fun@conference.botserv,hubotorium@conference.botserv" \
 HUBOT_XMPP_HOST="127.0.0.1" \
 HUBOT_XMPP_PORT=5222 \
 bin/hubot --adapter xmpp --name hubot
INFO { username: 'hubot@botserv',
  password: '********',
  host: '127.0.0.1',
  port: '5222',
  rooms:
   [ { jid: 'fun@conference.botserv', password: false },
     { jid: 'hubotorium@conference.botserv', password: false } ],
  keepaliveInterval: 30000,
  legacySSL: undefined,
  preferredSaslMechanism: undefined }
INFO Hubot XMPP client online
INFO Hubot XMPP sent initial presence
```

Hubot should now join your chatrooms. You can use tab completion to give commands:

```
(11:35:42 AM) hubot entered the room.
(11:35:58 AM) spajus: hubot ping
(11:35:58 AM) hubot: PONG
```

XMPP adapter allows giving a different name to your hubot, use `--name <name>` to do so.

Troubleshooting

If you use Openfire XMPP server, you will have to disable TLS security, current version of `hubot-xmpp` does not play well with it. To do that, go to Openfire administration panel, “Server Settings” -> “Security Settings” and in “Client Connection Security” section click “Custom” and set “Old SSL method” and “TLS method” to `Not Available`.

If you set wrong `HUBOT_XMPP_HOST` or `HUBOT_XMPP_PORT`, you will see `Reconnect in 0` looping in Hubot’s output.

Incorrect `HUBOT_XMPP_USERNAME` or `HUBOT_XMPP_PASSWORD` will give you this:

```
ERROR XMPP authentication failure
```

If you don’t set a correct room name, i.e. skip the trailing `@conference.server`, you will get this:

```
ERROR [xmpp error][<presence to="hubot@botserv/e1b17544" from="fun/hubot" typ\
e="error" xmlns:stream="http://etherx.jabber.org/streams"><error code="404" t\
ype="cancel"><remote-server-not-found xmlns="urn:ietf:params:xml:ns:xmpp-stan\
zas"/></error></presence>]
```

Slack

Slack is becoming increasingly popular lately. It is similar to HipChat, but has more lively interface and a bunch of integrations. You can also use it for free with limited number of integrations and history limit of 10,000 lines.

Troubles With Official Integration

There is an official Hubot integration via hubot-slack adapter, but you may find it rather unusable for several reasons:

- It communicates with Hubot over HTTP endpoint, meaning you have to expose your server's open port to public, and there is no way to whitelist Slack's IP range, because they're running on Amazon's AWS. That's a red flag for those who care about their security.
- When Hubot is offline, Slack notices that it's HTTP endpoint is not responding and automatically disables the integration. You will have to reenable it manually. This becomes very annoying, since Slack flips the switch even when Hubot is restarting.
- With official Slack Hubot integration, Hubot cannot be invited to join private channels and cannot be talked to directly via private message.

We will not waste our time setting up the official integration, but will move on directly to the good stuff.

Slack Over XMPP

Fortunately, Slack allows you to enable XMPP and IRC access to your team chat, and Hubot's XMPP adapter works like a charm. Using Slack's Administrator account, go to "Account" -> "Administration" -> "Settings" -> "Gateways", check `Enable XMPP gateway (SSL only)` and click `Save Settings`.

Next, log in with your Hubot user, go to "Account" -> "Your Team" -> "Gateways" and look for "Getting Started: XMPP". It will contain the details about your XMPP client configuration. Refer to "XMPP/Jabber" section above to install and configure `hubot-xmpp` adapter, and then feed it with Slack's configuration. Here is an example for "yourteam":

```
HUBOT_XMPP_USERNAME="hubot@yourteam.xmpp.slack.com"
HUBOT_XMPP_PASSWORD="yourteam.avUFcpc3G7N0Ot1pOV10"
HUBOT_XMPP_ROOMS="hubotorium@conference.yourteam.xmpp.slack.com,\
devops@conference.yourteam.xmpp.slack.com"
HUBOT_XMPP_HOST="yourteam.xmpp.slack.com"
HUBOT_XMPP_PORT=5222
```

This works perfectly and has no limitations whatsoever.

Skype

I've been running headless Skype with Hubot for over a year, and I assure you - if you choose this path, it will be hell. Skype will randomly loose connection with Hubot, randomly crash, hang in a loop and eat up CPU for no reason, Hubot will sometimes refuse to answer, you will experience lag and delays, and guys in Redmond will probably send every command you type directly to the NSA. I have tried multiple times to switch our company chat to something better, but people were too attached to it, so I was carrying this burden, until we finally switched to Slack. Boy did my life get easier. But if you really, really want (or rather are forced) to run Hubot on Skype, here goes nothing.

Installing Headless Skype

Instructions for installing Skype usually vary among different Linux distributions and Skype versions, so you should try finding the latest information. Debian has a decent wiki page with instructions.

I had to do this to get Skype on Debian Wheezy:

```
root@botserv:~# dpkg --add-architecture i386
root@botserv:~# apt-get update
root@botserv:~# wget -O skype-install.deb http://www.skype.com/go/getskype-li\
nux-deb
root@botserv:~# dpkg -i skype-install.deb
root@botserv:~# apt-get -f install # Fix missing dependencies
```

Then we have to install some more tools to run Skype in headless environment:

```
root@botserv:~# apt-get install xvfb screen x11vnc
```

Start your headless Skype with `hubot` user in a detached screen:

```
hubot@botserv:~$ screen -d -m -S Skype xvfb-run -n 0 -s "-screen 0 800x600x16\
" /usr/bin/skype
```

Now, the tricky part. SSH into your server with port forwarding, become `hubot` user, find headless Skype's `Xauthority` file and start `x11vnc` on display :0, like this:

```
spajus@unbound:~$ ssh spajus@192.168.0.32 -L 5900:localhost:5900
spajus@botserv:~$ sudo su - hubot
hubot@botserv:~$ ps auxwwww | grep auth
hubot     7453  0.0  0.9  75852  9392 pts/1    S+   Jan05   0:00 Xvfb :0 -scr\
een 0 800x600x16 -nolisten tcp -auth /tmp/xvfb-run.gZi7Bq/Xauthority
hubot@botserv:~$ x11vnc -auth /tmp/xvfb-run.gZi7Bq/Xauthority -display :0
```

Now you can use any VNC client from your computer to connect to `localhost:5900` and control the headless Skype. You will have to do everything manually - log in, add contacts, get added to desired group chats. Keep the VNC window open, we're not done yet.

Installing Skype Adapter

You don't want to install this adapter from default packages, because it will most probably be outdated. Instead install it directly from GitHub.

```
hubot@botserv:~/first-hubot$ npm install --save https://github.com/netpro2k/h\
ubot-skype/tarball/master
```

Check if it was added to your `package.json`:

```
hubot@botserv:~/first-hubot$ grep hubot-skype package.json
    "hubot-skype": "https://github.com/netpro2k/hubot-skype/tarball/master"
```

Now you have to install `Skype4Py` Python bindings for this adapter to work.

```
root@botserv:~# apt-get install python-pip
root@botserv:~# pip install Skype4Py
```

First Run with Skype

You will probably need to find the path to Xauthority file that was used by Xvfb to start headless Skype. In the example it's `/tmp/xvfb-run.gZi7Bq/Xauthority`.

```
hubot@botserv:~$ ps auxwwww | grep auth
hubot     7453  0.0  0.9  75852  9392 pts/1    S+   Jan05   0:00 Xvfb :0 -scr\
een 0 800x600x16 -nolisten tcp -auth /tmp/xvfb-run.gZi7Bq/Xauthority
```

Run Hubot Skype for the first time, while keeping an eye on your VNC window.

```
hubot@botserv:~/first-hubot$ XAUTHORITY=/tmp/xvfb-run.gZi7Bq/Xauthority \
 DISPLAY=:0 \
 bin/hubot --adapter skype --name hubot
```

In VNC a popup telling that "Another program wants to use Skype" will appear. Make sure you tick "Remember this selection" and allow it.

You should now be able to chat with your bot.

```
Tomas Varaneckas: hubot ping
Hubot: PONG
```

Troubleshooting

There are many things that can go wrong with this. Most notable are `XAUTHORITY` and `DISPLAY` settings. Make sure you have provided correct values, otherwise you will get this message:

```
Skype4Py.errors.SkypeAPIError: Could not open XDisplay
```

Sometimes Hubot will stop responding, and it's output will show "Lost connection to Skype". Restarting Hubot alone will probably not help, so you will have to kill the existing instance of `skype` and start a fresh one, then start Hubot after 10 or 20 seconds. If you will try to start Hubot too early, it will fail to connect to Skype.

When it comes to scripting, you will run into trouble of determining the room name of your personal and group chats. Include room-info.coffee in `hubot-scripts.json` and query the room name using `hubot room info`. You will learn more about including scripts in next chapter.

```
Tomas Varaneckas: hubot room info
Hubot: This room is: #tomas.varaneckas/$hubot5000;4c1bd379c8db7f51
```

You would have to provide `#tomas.varaneckas/$hubot5000;4c1bd379c8db7f51` for a room name if you wanted Hubot to output something in there.

Deploying Hubot To Production

We explored ways of starting Hubot directly from command line. This method is great for testing things out, but in the end you will want to have something for the long run - a way to deploy your Hubot so that it survives system restart, random crashes introduced by broken scripts, etc. And then you need a way to manage all the environmental variables for Hubot.

If you don't feel very comfortable around Linux, I recommend going for Hubot Control - a self-hosted web application that takes care of your Hubot maintenance and scripting needs. We'll take a look into that later in the end of this chapter.

Creating Hubot Instance For Production

Enough playing around, we have to create a Hubot instance for production use. By now you should already know what adapter you will be using and how to make it work. Throughout the examples in this book I will use Campfire, since it has the best possible integration. It will be located in `/home/hubot/campfire` and generated like this:

```
hubot@botserv:~$ hubot --adapter campfire --name hubot --create ~/campfire
Creating a hubot install at /home/hubot/campfire
```

Hubot And Source Control Management

To avoid losing your work, it's highly recommended to check Hubot in to SCM. Hubot generates `.gitignore` file for you, therefore if you choose to use another SCM, add `node_modules` directory to your ignore list.

Checking Hubot Into Git

Using Git is a vast topic which is far beyond the scope of this book, so the only example will be how to put your newly generated Hubot into a fresh Git repository.

```
hubot@botserv:~/campfire$ git config --global user.name "Tomas Varaneckas"
hubot@botserv:~/campfire$ git config --global user.email "tomas.varaneckas@gm\
ail.com"
hubot@botserv:~/campfire$ git init .
Initialized empty Git repository in /home/hubot/campfire/.git/
hubot@botserv:~/campfire$ git add .
hubot@botserv:~/campfire$ git commit -m "Initial commit"
[master (root-commit) b29ebb9] Initial commit
 21 files changed, 880 insertions(+)
 <...>
hubot@botserv:~/campfire$ git remote add origin git@github.com:spajus/hubot-e\
xample.git
hubot@botserv:~/campfire$ git push -u origin master
Counting objects: 25, done.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (25/25), 12.00 KiB, done.
```

```
Total 25 (delta 0), reused 0 (delta 0)
To git@github.com:spajus/hubot-example.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

You will be able to find code examples used in this book at https://github.com/spajus/hubot-example.

Handling Environmental Variables

Hubot is configured using environmental variables, and the best way to manage them is to have a separate shell script that sets them all.

We will be using a file `/home/hubot/campfire/hubot.conf` for that. In example code repository it will appear as `hubot.conf.example`, because I don't want to publish my real API keys, so I will be keeping my `hubot.conf` in `.gitignore`.

You may also want to keep your variables somewhere safe for security reasons. You can use `/etc/hubot/hubot.conf` instead and keep everything there.

Hubot As A Daemon

Let's make ourselves `/etc/init.d/hubot` script that will allow us to start and stop Hubot as a daemon, and do it automatically on system boot. We will use `start-stop-daemon`, which is available in Debian based Linux distributions, for RedHat branch you can use `daemon` function from `/etc/init.d/functions`.

In our example we will keep the startup script in `hubot-example/misc/hubot.init.d.sh`, but in real world scenario this should be handled with Chef or Puppet. Symlink the script to `/etc/init.d/hubot` and add it to system startup.

```
root@botserv:~# ln -s /home/hubot/campfire/misc/hubot.init.d.sh /etc/init.d/h\
ubot
root@botserv:~# insserv hubot
```

If you decide to remove Hubot from system startup, run this:

```
root@botserv:~# insserv -r hubot
```

Contents of the startup script:

/etc/init.d/hubot

```
#! /bin/sh
### BEGIN INIT INFO
# Provides:          hubot
# Required-Start:    $remote_fs $syslog
# Required-Stop:     $remote_fs $syslog
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: Hubot stat / stop script
# Description:       Manages starting and stopping of Hubot node bot
### END INIT INFO

# Author: Tomas Varaneckas <tomas.varaneckas@gmail.com>

# Do NOT "set -e"
# PATH should only include /usr/* if it runs after the mountnfs.sh script
DESC="Hubot node bot"
NAME=hubot
USER=hubot
GROUP=hubot
BOT_PATH=/home/hubot/campfire
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/usr/local/bin:$BOT_PATH/node_modules:$BOT\
_PATH/node_modules/hubot/node_modules
DAEMON=$BOT_PATH/bin/$NAME
DAEMON_ARGS="--adapter campfire --name hubot"
PIDFILE=$BOT_PATH/$NAME.pid
LOGFILE=$BOT_PATH/$NAME.log
SCRIPTNAME=/etc/init.d/$NAME
INIT_VERBOSE=yes

# Read configuration variable file if it is present
[ -r $BOT_PATH/hubot.conf ] && . $BOT_PATH/hubot.conf

# Load the VERBOSE setting and other rcS variables
. /lib/init/vars.sh

# Define LSB log_* functions.
# Depend on lsb-base (>= 3.2-14) to ensure that this file is present
# and status_of_proc is working.
. /lib/lsb/init-functions

do_start()
{
  status="0"
  pidofproc -p $PIDFILE node >/dev/null || status="$?"
  [ "$status" = 0 ] && return 2;

  touch $PIDFILE && chown $USER:$GROUP $PIDFILE

        start-stop-daemon --no-close --user $USER --quiet --start --pidfile
$PIDFILE\
 -c $USER:$GROUP \
          --make-pidfile \
          --background --chdir $BOT_PATH --exec $DAEMON—\
                $DAEMON_ARGS >> $LOGFILE 2>&1 \
                ||  return 2
}

do_stop()
{
  status="0"
  pidofproc -p $PIDFILE node >/dev/null || status="$?"
  [ "$status" = 3 ] && return 1

        start-stop-daemon --stop --quiet --pidfile $PIDFILE
        RETVAL="$?"
        [ "$RETVAL" = 2 ] && return 2
        rm -f $PIDFILE
        return "$RETVAL"
}

case "$1" in
  start)
```

```
        [ "$VERBOSE" != no ] && log_daemon_msg "Starting $DESC" "$NAME"
        do_start
        case "$?" in
                0|1) [ "$VERBOSE" != no ] && log_end_msg 0 ;;
                2) [ "$VERBOSE" != no ] && log_end_msg 1 ;;
        esac
        ;;
  stop)
        [ "$VERBOSE" != no ] && log_daemon_msg "Stopping $DESC" "$NAME"
        do_stop
        case "$?" in
                0|1) [ "$VERBOSE" != no ] && log_end_msg 0 ;;
                2) [ "$VERBOSE" != no ] && log_end_msg 1 ;;
        esac
        ;;
  status)
        status_of_proc -p $PIDFILE node $NAME && exit 0 || exit $?
        ;;
  restart|force-reload)
        log_daemon_msg "Restarting $DESC" "$NAME"
        do_stop
        case "$?" in
          0|1)
                do_start
                case "$?" in
                        0) log_end_msg 0 ;;
                        1) log_end_msg 1 ;; # Old process is still running
                        *) log_end_msg 1 ;; # Failed to start
                esac
                ;;
          *)
                # Failed to stop
                log_end_msg 1
                ;;
        esac
        ;;
  *)
        echo "Usage: $SCRIPTNAME {start|stop|status|restart|force-reload}" >&2
        exit 3
        ;;
esac

:
```

Don’t type it in, find a copy at github.com/spajus/hubot-example

There is also RHEL/CentOS version at github.com/spajus/hubot-example

Example use of this script:

```
hubot@focus:~/campfire$ /etc/init.d/hubot status
[FAIL] hubot is not running… failed!
hubot@focus:~/campfire$ /etc/init.d/hubot start
[ ok ] Starting Hubot node bot: hubot.
hubot@focus:~/campfire$ /etc/init.d/hubot status
[ ok ] hubot is running.
hubot@focus:~/campfire$ /etc/init.d/hubot restart
[ ok ] Restarting Hubot node bot: hubot.
hubot@focus:~/campfire$ /etc/init.d/hubot status
```

```
[ ok ] hubot is running.
hubot@focus:~/campfire$ /etc/init.d/hubot stop
[ ok ] Stopping Hubot node bot: hubot.
hubot@focus:~/campfire$ tail -n 5 hubot.log
[Thu Jan 09 2014 21:07:05 GMT+0200 (EET)] INFO Data for brain retrieved from \
Redis
[Thu Jan 09 2014 21:24:56 GMT+0200 (EET)] WARNING The HUBOT_AUTH_ADMIN enviro\
nment variable not set
[Thu Jan 09 2014 21:24:57 GMT+0200 (EET)] INFO Data for brain retrieved from \
Redis
[Thu Jan 09 2014 21:25:11 GMT+0200 (EET)] WARNING The HUBOT_AUTH_ADMIN enviro\
nment variable not set
[Thu Jan 09 2014 21:25:12 GMT+0200 (EET)] INFO Data for brain retrieved from \
Redis
```

Now your Hubot is a first class citizen in Linux.

Managing Hubots With Hubot Control

If you like web interfaces, check out Hubot Control. You can use it for various things, like creating and controlling multiple Hubot instances, writing scripts, managing configuration. It has a built in script editor that has syntax highlighting and validation.

It's far from being perfect, and you will probably be happier by setting everything up like this book suggests, with `hubot.conf` and `init.d` script.

Using Hubot Scripts

There are hundreds of scripts available for Hubot, offering you lots of great functionality without writing any code of your own. While the majority of these scripts are designed for fun, some can be extremely useful, depending on your technology stack.

Getting Help

Most scripts have `Commands:` section in their documentation header, and these commands can be seen using `hubot help` or `hubot help <topic>` in the chatroom. You can also see the help by visiting `http://<hubot_host>:<hubot_port>/hubot/help` in your browser.

Where To Find Scripts

A majority of scripts come as part of `hubot-scripts` dependency of you Hubot instance. You can explore `node_modules/hubot-scripts/src/scripts` directory to see what's in there.

```
hubot@botserv:~/campfire$ ls node_modules/hubot-scripts/src/scripts/
```

There is also a catalog of these scripts available at http://hubot-script-catalog.herokuapp.com/, but it's more fun to dig around the source at GitHub: https://github.com/github/hubot-scripts

There is also a newly introduced GitHub organization for new Hubot scripts that do not come bundled together with every Hubot installation. You can find these scripts at https://github.com/hubot-scripts.

Enabling Bundled Scripts

If you want to enable a script that is in `hubot-scripts` package, simply add it's file name to `hubot-scripts.json`. It should have some scripts added in there right out of the box, so if you wan to add `gemwhois.coffee`, your `hubot-scripts.json` could look like this:

hubot-scripts.json

```
["redis-brain.coffee", "shipit.coffee", "gemwhois.coffee"]
```

You have to restart Hubot every time you modify `hubot-scripts.json`. Then try if it works.

```
Tomas V.  hubot gem whois rails
Hubot          gem name: rails
                 owners: David Heinemeier Hansson
                   info: Ruby on Rails is a full-stack web framework optimize\
d for programmer
                         happiness and sustainable productivity. It encourage\
s beautiful code
                         by favoring convention over configuration.
```

```
          version: 4.0.2
        downloads: 31088002
         homepage: http://www.rubyonrails.org
    documentation: http://api.rubyonrails.org
      source code: http://github.com/rails/rails
```

Installing Script Dependencies

Some scripts will have extra dependencies that you will have to install before use. For instance, http-info.coffee has two dependencies listed in it's header:

```
...
# Dependencies:
#   "jsdom": "0.2.15"
#   "underscore": "1.3.3"
...
```

Install them with `npm install --save <dependency>` and it will add the dependency to your `package.json` automatically:

```
hubot@botserv:~/campfire$ npm install --save jsdom underscore
hubot@botserv:~/campfire$ grep -E jsdom\|underscore package.json
    "underscore": "~1.5.2",
    "jsdom": "~0.8.10"
```

Note that installed dependencies are newer, and that sometimes can break the scripts, so if for some reason it does not work, try reinstalling the packages with exact versions that are provided, like this:

```
npm uninstall --save jsdom underscore
unbuild jsdom@0.8.10
unbuild underscore@1.5.2
npm install --save jsdom@0.2.15 underscore@1.3.3
hubot@botserv:~/campfire$ grep -E jsdom\|underscore package.json
    "underscore": "~1.3.3",
    "jsdom": "~0.2.15"
```

Now add `http-info.coffee` to `hubot-scripts.json`, restart the bot and casually mention some URL in the chatroom to see what happens.

```
Tomas V.  hubot help http
Hubot     http(s)://<site> - prints the title and meta description for sites \
linked.
Tomas V.  check out https://github.com/spajus/hubot-control
Hubot     spajus/hubot-control - GitHub
          hubot-control - Control Hubot via web interface
```

Installing Script Packages

Script packages from https://github.com/hubot-scripts organization need to be installed with `npm` first. Let's include hubot-plusplus, which allows you to keep track of karma for arbitrary things.

```
hubot@focus:~/campfire$ npm install --save hubot-plusplus
```

Then add `hubot-plusplus` to `external-scripts.json`, so it looks like this:

external-scripts.json

```
["hubot-plusplus"]
```

Finally, restart your Hubot and try it out.

```
Tomas V.  ruby++
Hubot     ruby has 1 points
Tomas V.  java--
Hubot     java has -1 points
```

Configuring Scripts

Many scripts require additional configuration via environmental variables. You can put them to your `hubot.conf`, next to adapter configuration. `http-info` script that we have previously enabled uses `HUBOT_HTTP_INFO_IGNORE_URLS` to exclude URLs by regex pattern. It's useful if you paste a lot of internal urls in your chat and you don't want Hubot to be all noisy about them. Let's make our Hubot ignore anything with `github.com/spajus` in it.

hubot.conf

```
# Campfire adapter configuration
...

# Ignore URLs that contain `github.com/spajus`
export HUBOT_HTTP_INFO_IGNORE_URLS="github\.com/spajus"
```

Restart Hubot and try it out.

```
Tomas V.  https://github.com/github/hubot
Hubot     github/hubot - GitHub
          hubot - A customizable, kegerator-powered life embetterment robot.
Tomas V.  https://github.com/spajus/hubot-pubsub
          http://hubot-script-catalog.herokuapp.com/
Hubot     Hubot Script Catalog
```

Uninstalling Scripts

After trying out a lot of scripts, you will definitely want to get rid of some. Just remove the script name from `hubot-scripts.json` or `external-scripts.json`, and optionally uninstall node modules.

```
hubot@focus:~/campfire$ npm uninstall --save hubot-plusplus
unbuild hubot-plusplus@1.0.0
```

Hubot Scripting

Hubot is written in Node.js, using CoffeeScript, 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
- 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 - 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.

```
      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 API and post the weather information.

script/weather.coffee

```coffeescript
# 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.

```coffeescript
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.

```coffeescript
  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 <location>”, and a callback function which will be triggered if a match is found.</location>

```
    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 and assign it to `fs` variable.

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

We check if `./logs/` directory exists, 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 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.

```
      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 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 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.

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 - `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.

```
    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, 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:

```
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. 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.

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 to write tests like this one:

```
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. 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. 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.

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, which also gives you a web based IDE for quick scripting.

scripts/template.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.

Roles And Authentication

In most cases this may be not necessary, but you may want to restrict more sensitive Hubot actions to a handful of people that can trigger them. The cleanest way to do this is using Hubot's Auth, which comes along with your Hubot instance. You can find the script at `scripts/auth.coffee`.

Setting Hubot Auth Admin

Every adapter implementation has different way of recognizing users. There is a handy command that shows you how Hubot sees users with your adapter - `hubot show users`. This is how it looks like in Campfire:

```
Tomas V.  hubot show users
Hubot     1502861 Tomas Varaneckas <tomas.varaneckas@gmail.com>
          1502862 Hubot <hubot@botserv.varaneckas.com>
          1522958 Jesse Pinkman <jesse@bluesky.com>
```

It may look a little different with other chat adapters, but all you need from this output is the ID of every user you want to be able to administer Hubot roles, and that ID is the first number. You should set `HUBOT_AUTH_ADMIN` environmental variable to comma separated list of admin user IDs.

To illustrate futher examples, I'll just set my own Campfire ID, since I don't really trust Jesse Pinkman:

hubot.conf

```
# Comma separated list of users who administer Hubot Auth
export HUBOT_AUTH_ADMIN=1502861
```

After restarting Hubot, I should be able to see myself having Admin role:

```
Tomas V.  hubot who has admin role?
Hubot     Tomas Varaneckas: The following people have the 'admin' role: Tomas\
 Varaneckas
```

Assigning Roles

Only Admin users can assign roles. You don't have to create a role before assigning. All you have to do is tell Hubot who is who using `hubot <user> has <role> role`. And you no longer have to use those cryptic IDs anymore:

```
Tomas V.  hubot Jesse Pinkman has developer role
Hubot     Tomas Varaneckas: Ok, Jesse Pinkman has the 'developer' role.
```

Check the assigned roles using `hubot what roles does <user> have?`:

```
Tomas V.  hubot what roles does Jesse Pinkman have?
Hubot     Tomas Varaneckas: Jesse Pinkman has the following roles: developer.
```

To remove the role from somebody, use `hubot <user> does not have <role> role`:

```
Tomas V.  hubot Jesse Pinkman does not have developer role
Hubot     Tomas Varaneckas: Ok, Jesse Pinkman doesn't have the 'developer' ro\
le.
```

You can assign multiple roles to multiple users.

Applying Roles

Now, time to break the bad news. While Hubot Auth is pretty flexible, you will have to edit your scripts to apply those roles. Luckily, there is not much to edit. There is a simple function that checks if user has a role - `robot.Auth.hasRole(msg.envelope.user, '<role>')`. This is how you use it in a script:

scripts/auth-example.coffee

```
module.exports = (robot) ->
  robot.respond /do dangerous stuff/i, (msg) ->
    if robot.auth.hasRole(msg.envelope.user, 'developer')
      doDangerousStuff(msg)
    else
      msg.reply "Sorry, you don't have 'developer' role"

  doDangerousStuff = (msg) ->
    msg.send "Doing dangerous stuff"
```

This is how it looks in action:

```
Tomas V.  hubot do dangerous stuff
Hubot     Tomas Varaneckas: Sorry, you don't have 'developer' role
Jesse P.  hubot do dangerous stuff
Hubot     Doing dangerous stuff
```

Restricting Command Execution

Hubot is a powerful tool, but with great power comes great responsibility. To control the danger, you can restrict where and when some commands can be executed.

Restricting Execution To Certain Rooms

There may be a situation when you would want some command to be executed only in one room. For example, if you deploy your application with Hubot, you want all deployments to happen in Deployments room, so that you don't have to skim through all the rooms to see if somebody is deploying something.

Hubot does not have any special means for this, but we can easily add restrictions to virtually any script with couple of lines of code. First, you need to know the ID of room you are in. This may be simple with some adapters, like Campfire, where room id is visible in the URL, or IRC, where room id is `#something`. There is room-info.coffee bundled with Hubot scripts to help you find the room id if you have trouble with that. Add `"room-info.coffee"` to `hubot-scripts.json`, restart Hubot and say `hubot room info` in the room you want to rescrict a command to. This is how room IDs look in Skype:

```
Tomas Varaneckas: hubot room info
Hubot:            This room is: #tomas.varaneckas/$hubot;5e184b59e0a11db6
```

Yes, `#tomas.varaneckas/$hubot;5e184b59e0a11db6` is how room ID looks in Skype, nobody could ever guess it.

Anyway, when you have your room ID handy, time to restrict the execution. Just check if `msg.message.room` matches your room ID in the beginning of every restricted command:

```
robot.respond /deploy (.*)/i, (msg) ->
  if msg.message.room != '<room-id>'
    msg.send 'You can only deploy in "Deployments" room.'
    return
  ...
```

Limiting Execution Hours

Imagine a fine Friday evening. You're sitting in a bar with your friends, having your third pint of beer, and suddenly Nagios starts knocking - your app just went down. You take out your laptop and spend 20 minutes trying to pinpoint the problem. The beer you just had doesn't help, but finally you realize that a very inefficient query was just deployed along with other changes, and database load went over the roof. You spend 30 more minutes reverting the change and putting the site back up. You cannot really blame the developer, who was trying to do his best and started the deployment at 8 PM on Friday night, doing overtime for the sake of the company.

That scenario was based on true story, and to prevent it from happening, you can restrict dangerous operations to working hours. Here is an example how to do it:

```
# Your TimeZone offset
time_offset = 3

# Function that tells if current time is within working hours
isWorkingHours = ->
  now = new Date()
  current_hour = (now.getHours() + time_offset) % 24
  current_day = now.getDay()
  current_hour in [9..18] && current_day in [1..5]

robot.respond /deploy (.*)/i, (msg) ->
  unless isWorkingHours()
    msg.send 'You can only deploy during working hours'
    return
  ...
```

This will definitely give you peace of mind next time you’re out on Friday night.

Monitoring With Hubot

Time to start turning Hubot into a whistleblower who is always first to tell you when something goes wrong. We will look through several possible scenarios that could prove to be useful in environment.

Hubot PubSub

After using simple HTTP notification script for many months, I realized that it's not convenient to route Hubot messages by providing room along with your request. You may want to split the stream into more rooms, and eventually routing gets harder to handle. That's where Hubot PubSub comes in handy.

Hubot PubSub uses publish-subscribe concept - you ask Hubot to subscribe rooms to various event types, and you can publish events which then get soft-routed to interested parties. It decouples message publishing and subscriptions - systems that publish events don't have to know what rooms will be subscribing them.

Take a look at https://github.com/spajus/hubot-pubsub - it has an animated GIF showing event routing in action.

Alternative To Hubot PubSub

Hubot PubSub will be used often in the scripts throughout this book. If you don't need the flexibility it offers and prefer to have fewer dependencies, simply use `robot.messageRoom room, message` instead of `robot.emit 'pubsub:publish', event, message`.

Installing Hubot PubSub

To install Hubot PubSub, go to your Hubot directory and install the package:

```
hubot@focus:~/campfire$ npm install --save hubot-pubsub
hubot-pubsub@0.0.2 node_modules/hubot-pubsub
```

Then add `"hubot-pubsub"` to `external-scripts.json`, so it looks like this:

```
["...", "hubot-pubsub"]
```

You need to restart Hubot afterwards

Subscribing To Event Notifications

Let's see if our room has any subscriptions, and subscribe it to get `error` events:

```
Tomas V.  hubot subscriptions
Hubot     Total subscriptions for 585164: 0
Tomas V.  hubot subscribe errors
Hubot     Subscribed 585164 to errors events
Tomas V.  hubot subscriptions
```

```
Hubot      errors -> 585164
           Total subscriptions for 585164: 1
```

You can subscribe as many rooms as you want to receive any set of events you like.

Publishing Events

There are two ways to publish an event. The simple way, mostly used for testing purposes or announcements, is asking Hubot to do it:

```
Tomas V.   hubot publish news network will be down for 5 minutes - upgrading h\
ardware
Hubot      news: network will be down for 5 minutes - upgrading hardware
           Notified 2 rooms about news
```

When you want to publish events from shell scripts or remote systems, use HTTP requests. This is how it's done with `curl`:

```
hubot@focus:~/campfire$ curl "http://localhost:8080/publish?event=errors&data\
=boom"
```

Then in chatrooms subscribed to `errors`, it will appear like this:

```
Hubot      errors: boom
```

You may want to use `POST` requests rather than `GET`:

```
hubot@focus:~/campfire$ curl -X POST \
  -d 'event=errors' \
  -d 'data=Stack trace in your face' \
  "http://localhost:8080/publish"
```

Using Event Namespaces For Advanced Message Routing

Hubot PubSub treats `.` as namespace separator, and it automatically notifies about sub-events, therefore if you have subscribed to `errors`, you will also receive `errors.db` and `errors.app`. And when the volume of `errors` stream starts getting too big to handle, you can divide and conquer it - subscribe one room to `errors.db`, and another one to `errors.app`. Plan ahead and you will never need to change anything in your applications and monitoring scripts.

A quick demo of how splitting up of event stream looks like:

Room: Errors

```
Tomas V.   hubot subscribe errors
Hubot      Subscribed 585163 to errors events
Hubot      errors.app: lost connection to redis
Hubot      errors.app.500: error 500 in /users/23/update: Transaction timeout
Hubot      errors.db: deadlock detected in users table
Hubot      errors.app.401: error 401 in /admin: bad login attempt from ip 10.1\
0.0.48
Hubot      errors.db: db2.infra.net: slave 10 seconds behind master
```

When you decide to create "Errors: App" and "Errors: DB" rooms:

Room: Errors

```
Tomas V.  hubot unsubscribe errors
Hubot     Unsubscribed 585163 from errors events
```

Room: Errors: App

```
Tomas V.  hubot subscribe errors.app
Hubot     Subscribed 585164 to errors.app events
Hubot     errors.app.404: error 404 in /users/foobar
Hubot     errors.app.401: error 401 in /users/login: bad login attempt from i\
p 45.47.184.12
Hubot     errors.app.response: average response time > 0.5 sec
```

Room: Errors: DB

```
Tomas V.  hubot subscribe errors.db
Hubot     Subscribed 585165 to errors.db events
Hubot     errors.db: db2.infra.net: slave 15 seconds behind master
Hubot     errors.db: query running over 60 seconds: "select * from users wher\
e status in (1,3,55)"
```

Later you can split `errors.app` stream into several parts, each covering different error code, and perhaps not subscribe to `errors.app.404` since it's very noisy.

Handling Unsubscribed Events

Sometimes you may get lost - some events are not received when you expect them, possibly due to mistype in event name. Also, you may want to know what events are not subscribed at all. There is one special event type that catches all unsubscribed events. Subscribe a room to `unsubscribed.event` and you will get this:

Room: Hubot Debugging

```
Tomas V.  hubot subscribe unsubscribed.event
Hubot     Subscribed 585168 to unsubscribed.event events
Hubot     unsubscribed.event: erors.app.fatal: lost connection to db1.infra.n\
et
```

Now you know that your app is sending `erors.app.fatal` instead of `errors.app.fatal`, so that's why you're not getting anything when you know it happens.

Securing Hubot PubSub

Events that contain sensitive data should not be sent over plain HTTP. Put Hubot under HTTPS, or use it only in local network or VPN.

To protect against unwanted message publishing, you may want to set a password that will be required when using HTTP endpoints for publishing messages. To do that, set `HUBOT_SUBSCRIPTIONS_PASSWORD` environmental variable and restart Hubot. Then provide `password` parameter along with your HTTP requests:

```
hubot@botserv:~$ curl "http://botserv:8080/publish?event=test&data=hello&pass\
word=secret"
```

Publishing Events From Ruby

You've seen how to publish events with `curl`, that means you can use it in bash scripts, but what about your application code?

All you have to do, is make an HTTP request. Every programming language has a library for that, and it's usually built-in. A basic Hubot client in Ruby can look like this:

```ruby
require 'net/http'

module Hubot
  def self.publish(message, event, host, password=nil)
    uri = URI("#{host}/publish")
    params = { event: event, data: message }
    params[:password] = password if password
    uri.query = URI.encode_www_form(params)
    Net::HTTP.get_response(uri)
  rescue => e
    puts "Failed to publish an event via Hubot: #{e}"
  end
end

Hubot.publish('I like rubies!', 'news', 'http://botserv:8080')
```

It would probably be a bad idea to use this script in production under heavy load - there are no timeout settings, and you don't want HTTP requests to Hubot to lock your app in case Hubot is restarting or down. Here is a better implementation that uses rest-client gem:

```ruby
require 'rest_client'

class Hubot
  TIMEOUT = 0.1
  ENDPOINT = 'http://botserv:8080/publish'
  PASS = 'secret'
  def self.publish(event, message)
    payload = { event: event, data: message, password: SECRET }
    RestClient::Request.execute(method: :post,
                                url: ENDPOINT,
                                open_timeout: TIMEOUT,
                                timeout: TIMEOUT,
                                payload: payload)
    rescue => e
      Rails.logger.warn("Hubot publish failed: #{event}: #{message}: #{e}")
    end
  end
end

Hubot.publish('news', 'rest-client implementation rocks!')
```

Now you can safely use it in production knowing it will have minimal impact to your users no matter what happens.

What And When To Monitor

Monitoring is a tricky game, those who are on call usually hate it. There is even a movement called Monitoring Sucks. Monitoring with Hubot can suck too, just start publishing everything to hubot and your chatrooms will soon be firehosed with messages that nobody will read.

A good way to monitor is to have a separate room for errors where you publish critical messages instantly, and warn about error rate when it reaches certain threshold. Critical messages must appear really often. If your application can run without major problems after an error - it's not critical.

Monitoring Error Rates With Graylog2 Alarms And Hubot PubSub

Graylog2 is an open source data analytics system for aggregating, searching and charting your logs. It can be configured to publish it's alarms to Hubot. In the older version of Graylog2 web interface (below 0.20.0), it is done using "Exec alarm callback" plugin. A short recipe to get you started:

1. Install Exec alarm callback graylog-server plugin
2. Configure some alarms in your graylog streams (`streams` -> `<stream>` -> `alarms`)
3. In Graylog2 web interface go to "System settings" (`settings` -> `system`)
4. Find "Exec alarm callback" configuration block
5. Check "Forced for all streams"
6. Click "Configure" and set the executable command to the path to alarm callback notification script, which should be executable and have following contents (make sure to change the configuration):

/opt/graylog2/hubot-alarm-callback

```ruby
#!/usr/local/bin/ruby

##### Configuration ########################################
hubot_pubsub_uri  = 'http://botserv:8080/publish'
hubot_pubsub_pass  = 'secret'
hubot_pubsub_event  = 'graylog.alert'

graylog_messages_uri  = 'https://graylog.infra.net/messages'
############################################################

require 'uri'
require 'net/http'

# Example:
# Stream message count alert: [errors]
topic  = ENV['GL2_TOPIC']

# Example:
# Stream [errors] received 549 messages in the last 5 minutes. Limit: 500
desc   = ENV['GL2_DESCRIPTION']

begin
  uri  = URI(hubot_pubsub_uri)
  params  = { password: hubot_pubsub_pass,
              event: hubot_pubsub_event,
              data: "#{graylog_messages_uri} -> #{desc}" }
  uri.query  = URI.encode_www_form(params)
  Net::HTTP.get_response(uri)
rescue => e
  puts "Hubot message failed: #{e}"
end
```

You can test the script like this:

```
graylog@graylog.infra.net:~$ GL2_TOPIC="test topic" \
  GL2_DESCRIPTION="test decription" \
  /opt/graylog2/hubot-alarm-callback
```

Now, subscribe some room to `graylog.alert` using hubot-pubsub, and you will start getting messages like this:

```
Hubot   graylog.alert: https://graylog.infra.net/messages ->
Stream [background-jobs] received 272 messages in the last 5 minutes. Limit: \
150
```

Receiving Nagios Alerts

It is possible to configure Hubot to receive Nagios service and alerts. We will do it by sending an HTTP POST from Nagios to Hubot. To start, we will create a simple Hubot script that consumes a POST on `/nagios/alert`:

Hubot script that would consume this HTTP POST would look like this:

scripts/nagios-alert.coffee

```
# Description:
#   Receives Nagios alerts and posts them to chatroom
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /nagios/alert (message=<message>)

module.exports = (robot) ->
  robot.router.post "/nagios/alert", (req, res) ->
    res.end()
    robot.emit 'pubsub:publish', 'nagios.alert', req.body.message
```

Now, let’s implement the rest in Nagios. We will assume that your Nagios configuration is at `/etc/nagios`.

To begin with, create a simple shell script that would take message as a variable and post it to Hubot’s HTTP endpoint. We will put it in `/etc/nagios/plugins/notify_by_hubot.sh`:

/etc/nagios/plugins/notify_by_hubot.sh

```
#!/bin/bash
HUBOT_URL="http://botserv:8080/nagios/alert"
MESSAGE=$1
curl $HUBOT_URL --data-urlencode message="$MESSAGE"
```

To use it from Nagios, we will have to add `notify-service-by-hubot` and `notify-host-by-hubot` command definitions to `/etc/nagios/objects/commands.cfg`:

/etc/nagios/objects/commands.cfg

```
define command{
  command_name notify-service-by-hubot
  command_line /etc/nagios/plugins/notify_by_hubot.sh \
    "Service $NOTIFICATIONTYPE$: $HOSTALIAS$/$SERVICEDESC$ is $SERVICESTATE$"
}

define command{
  command_name notify-host-by-hubot
  command_line /etc/nagios/plugins/notify_by_hubot.sh \
    "Host $NOTIFICATIONTYPE$: $HOSTALIAS$ is $HOSTSTATE$"
}
```

Then define Hubot contact in `/etc/nagios/objects/contacts.cfg`:

/etc/nagios/objects/contacts.cfg

```
# Hubot chat notifications
define contact{
  contact_name                    hubot
  use                             generic-contact
  alias                           Hubot
  service_notification_commands   notify-service-by-hubot
  host_notification_commands      notify-host-by-hubot
}
```

And add `hubot` contact to a group that is configured to receive notifications, i.e. `admins`:

/etc/nagios/objects/contacts.cfg

```
define contactgroup{
  contactgroup_name       admins
  alias                   Nagios Administrators
  members                 jimmy,spajus,hubot
}
```

Restart Nagios, subscribe your Hubot chatroom to `nagios.alert` and wait for it. Having a separate chatroom for Nagios alerts would be wise, because it may get noisy.

```
Tomas V.   hubot subscribe nagios.alert
Hubot      Subscribed 585164 to nagios.alert events
Hubot      nagios.alert: Service PROBLEM: jobs/resque is WARNING
Hubot      nagios.alert: Service PROBLEM: webapp/rake-stuck is WARNING
```

Graphing With Hubot

Hubot is notorious for littering your chat with fun animated GIFs and random pictures of pugs and squirrels. However, you can exploit this feature to actually do something useful - show graphs from various graphing systems, like Graphite or Cacti.

Graphite

Graphite can visualize a ton of metrics that you can collect via UDP using Statsd, and Hubot has a script for querying and drawing all those wonderful graphs. It ships with `hubot-scripts` package that is included in your Hubot.

This script was recently broken, so if you happen to have an older version, simply download the latest version from github/hubot-scripts and put it in your Hubot's `scripts/`. You can tell if you have the old version by examining the list of commands your `graphite.coffee` supports. Latest version should have `graphite list` command available.

In case you will be using the version that ships with your Hubot, don't forget to add `"graphite.coffee"` to `hubot-scripts.json`.

Configuring Hubot Graphite Script

You will have to set a couple of environmental variables to get this script working. Place them into `hubot.conf`.

hubot.conf

```
# Graphite URL
export GRAPHITE_URL="https://graphite.your.org"
# Graphite Port (optional unless non standard)
# export GRAPHITE_PORT=1234
# Graphite username:password for Basic Auth
export GRAPHITE_AUTH="hubot:supersecret"
```

Trying It Out

To use this script, you have to have some graphs saved at Graphite first. To do that, you have to be logged into Graphite and when you build your graph, save it using the Save icon on top-left. The graph should then be listed under `User Graphs` in the tree on the left.

To get the list of graphs you have, just say `graphite list`:

```
Tomas V.  graphite list
Hubot     metrics.deployments
          metrics.portal.logins
          metrics.portal.signups
          metrics.portal.errors
```

If the list is too big, you can search for graphs matching a query:

```
Tomas V.  graphite search portal
Hubot     metrics.portal.logins
          metrics.portal.signups
          metrics.portal.errors
```

Finally, request the graph you want to see using `graphite show <graph.name>`. You can provide only a part of graph name, just make sure it's unique because Hubot will always show the first graph that matches the query, so if you say `graphite show portal`, in our example Hubot will show `metrics.portal.logins` - you will have to use `graphite show signups` to see `metrics.portal.signups`.

```
Tomas V.  graphite show portal.errors
Hubot     https://graphite.your.org/render/...
```

You will see graph images if you use Campfire, Hipchat, or some other chat that supports displaying inline images. With other adapters, like IRC, you will only get a link that you will have to click to see the graph. Still, pretty useful.

Extending The Script

This script has some limitations. For instance, you graph will have fixed time window that you have defined when saving it. Wouldn't it be cool if you could say `hubot show deployments -1year` and get the graph with last year of deployments?

Let's modify the original script, so it will do just that. If you are using `graphite.coffee` that ships with Hubot, first remove it from `hubot-scripts.json`, then copy it to `scripts/` directory.

Now, all we want to do is to extend `graphite show <graph.name>` with optional extra parameter - `graphite show <graph.name> [offset]`. To do that, we will make a couple of small changes.

Find this code:

graphite.coffee

```
robot.hear /graphite show (\S+)/i, (msg) ->
  treeversal msg, (data) ->
    construct_url msg, data[0].graphUrl, (url) ->
      msg.send url
```

And change it to this:

graphite.coffee

```
robot.hear /graphite show (\S+)( (.+))?/i, (msg) ->
  offset = msg.match[3]
  treeversal msg, (data) ->
    construct_url msg, data[0].graphUrl, (url) ->
      if offset
        if url.match /&from=[-+0-9a-z]+/i
          url = url.replace /&from=[-+0-9a-z]+/i, ''
        url = url.replace '#', "&from=#{offset}#"
      msg.send url
```

What have we done?

graphite.coffee

```
robot.hear /graphite show (\S+)( (.+))?/i, (msg) ->
  offset = msg.match[3]
```

This new regexp matches extra word, which will appear in `msg.match[3]`. If you wondering why not `msg.match[2]` - it would contain extra space in the beginning.

graphite.coffee

```
        if offset
          if url.match /&from=[-+0-9a-z]+/i
            url = url.replace /&from=[-+0-9a-z]+/i, ''
          url = url.replace '#', "&from=#{offset}#"
```

Here we check if offset was provided, and if it was, we want to remove the original `&from=<something>` from graphite URL before enhancing it with new offset.

Since this URL always ends with `#<timestamp>&png`, it's a good place to hook in and add our custom `&from=<offset>`.

It should work:

```
Tomas V.  graphite show deployments -1year
Hubot     https://graphite.your.org/...&from=-1year#1393476771285&png
```

Never give up if a script doesn't do everything you want it to - it's called open source for a reason. And if your changes are significant enough, consider contributing them back.

GitHub Integration

GitHub is a wonderful platform for organizing and collaborating on software development, and if your company uses it to full extent, consider yourself lucky. There are so many ways to integrate Hubot with GitHub, it deserves a separate chapter.

Putting An End To GitHub Email Notifications

If you happen to work on an active GitHub repository, or belong to organization that does everything via GitHub, you should agree on a few things:

- GitHub is awesome
- GitHub is not perfect
- GitHub web notifications are a mess
- GitHub email notifications are out of hand

First thing you should do is to turn off email notifications. Forget all those filters, just use web notifications (look for a blue dot on top of the page), they are far better. And to get notified about things in real time without leaving junk in your inbox, we will use the power of Hubot. It's way better to receive notifications about all those issues and pull requests in your chat, not in your mail.

Why Not Just Use GitHub Service Hooks?

You may think - hey, GitHub has hundreds of service integrations, and you can even propose your own in github/github-services. And yes, when it comes to getting GitHub events to appear in your chat, sometimes it may fit your needs. For example, GitHub can post information about commits directly to Campfire. But what if you want to edit something? You can't change how the message looks, you can't make it tell you about new Pull Requests. Oh, but if you use IRC, then it will tell you about Pull Requests.

That's why it's nice to have Hubot in the middle, so you can leverage the power of GitHub to full extent, and have complete control over what you are getting.

Creating Auth Tokens

To integrate with GitHub API, you first have to create an Auth Token. To do that, you can either go to https://github.com/settings/applications and click "Create new token" in "Personal Authentication Tokens" panel, or you can do it in command line:

```
curl -u 'your-github-username' \
     -H "X-GitHub-OTP: 123456" \ # Two-Factor auth code, omit if non-relevant
     -d '{"scopes":["repo"],"note":"Hubot Auth Token"}' \
     https://api.github.com/authorizations
Enter host password for user 'your-github-username':
{
  "id": 5415883,
    "url": "https://api.github.com/authorizations/5415883",
    "app": {
      "name": "Hubot Auth Token (API)",
      "url": "http://developer.github.com/v3/oauth/#oauth-authorizations-api",
      "client_id": "00000000000000000000"
    },
    "token": "44283ef0f15c645629dd29222410ec9b58507f1b",
    "note": "Hubot Auth Token",
    "note_url": null,
    "created_at": "2014-01-25T17:59:38Z",
    "updated_at": "2014-01-25T17:59:38Z",
    "scopes": [
      "repo"
    ]
}
```

Store the `token` somewhere safe, as it is nearly as powerful as your password.

Creating GitHub API Hooks

Most of GitHub related monitoring is based on GitHub API hooks. You create a hook, give it an endpoint, and it will make HTTP requests when events occur.

You can create hooks over web interface - go to repository settings, select “Service Hooks”, then “WebHook URLs”. GitHub has recently renewed their hooks management, it is now much more flexible than it was before. Still, my prefered way of doing this is using the command line.

If you will use web based hook management, make sure you always select “Payload version” that ends with `json`, or otherwise scripts provided in this book may not work.

There are hooks available for over a dozen events, like `push`, `issues`, `pull_request`, `fork`, `team_add`, etc. You can find a full and up to date list in GitHub developer documentation.

Let’s create a very simple Hubot script that will be able to receive HTTP requests and dump their contents, so we can use it as a starting point for our integrations.

scripts/github-hook-test.coffee

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

  robot.router.get "/github/test", (req, res) ->
    dump 'Received GET:', req, res

  robot.router.post "/github/test", (req, res) ->
    dump 'Received POST:', req, res

  robot.router.put "/github/test", (req, res) ->
    dump 'Received PUT:', req, res

  dump = (message, req, res) ->
    console.log message, JSON.strigify(req.body, null, 2)
    res.end()
```

Keep in mind, that GitHub should be able to access Hubot’s HTTP endpoint to post the data. You should also take actions to secure your Hubot HTTP endpoint to prevent malicious attempts to perform unwanted HTTP requests. A common practice is to use firewall to restrict public access and whitelist GitHub IP range. You can find out GitHub IP range on their page, or even better, using our favorite tool - the command line:

```
hubot@botserv:~$ curl https://api.github.com/meta
{
  "verifiable_password_authentication": true,
  "hooks": [
    "192.30.252.0/22"
  ],
  "git": [
    "192.30.252.0/22"
  ]
}
```

Now, let’s create a hook that will tell us about pushes. You will need to know the public endpoint of your Hubot. To test it, just open `http://<hubot.server>/hubot/help` in your browser and see if it shows the help.

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  -d '{"name":"web","active":true,"events":["push"], \
      "config":{"url":"http://botserv.your.org:8080/github/test", \
      "content_type":"json"}}' \
  https://api.github.com/repos/spajus/hubot-example/hooks
{
  "url": "https://api.github.com/repos/spajus/hubot-example/hooks/1730429",
  "test_url": "https://api.github.com/repos/spajus/hubot-example/hooks/173042\
9/test",
  "id": 1730429,
  "name": "web",
  "active": true,
  "events": [
    "push"
  ],
  "config": {
    "url": "http://botserv.your.org:8080/github/test",
    "content_type": "json"
  },
  "last_response": {
    "code": null,
    "status": "unused",
    "message": null
  },
  "updated_at": "2014-01-26T06:02:04Z",
  "created_at": "2014-01-26T06:02:04Z"
}
```

After a push is made to the repo, here is what `hubot.log` shows:

```
Received POST: {
   "ref": "refs/heads/master",
   "after": "80398238f9f2952413278ae7a286a9e7b67af9a9",
   "before": "e52a9c120c7b6636966aa72cf6c37e5707023c14",
   "created": false,
   "deleted": false,
   "forced": true,
   "compare": "https://github.com/spajus/hubot-example/compare/e52a9c120c7b...\
80398238f9f2",
   "commits": [
    {
      "id": "80398238f9f2952413278ae7a286a9e7b67af9a9",
      "distinct": true,
      "message": "Add GitHub hooks test script",
      "timestamp": "2014-01-25T22:38:52-08:00",
      "url": "https://github.com/spajus/hubot-example/commit/80398238f9f29524\
13278ae7a286a9e7b67af9a9",
      "author": {
        "name": "Tomas Varaneckas",
        "email": "tomas.varaneckas@gmail.com",
        "username": "spajus"
      },
      "committer": {
        "name": "Tomas Varaneckas",
        "email": "tomas.varaneckas@gmail.com",
        "username": "spajus"
      },
      "added": [
        "scripts/github-hook-test.coffee"
      ],
      "removed": [],
      "modified": []
    }
  ],
  "head_commit": {
    "id": "80398238f9f2952413278ae7a286a9e7b67af9a9",
    "distinct": true,
    "message": "Add GitHub hooks test script",
    "timestamp": "2014-01-25T22:38:52-08:00",
    "url": "https://github.com/spajus/hubot-example/commit/80398238f9f2952413\
278ae7a286a9e7b67af9a9",
    "author": {
      "name": "Tomas Varaneckas",
      "email": "tomas.varaneckas@gmail.com",
      "username": "spajus"
    },
    "committer": {
      "name": "Tomas Varaneckas",
      "email": "tomas.varaneckas@gmail.com",
      "username": "spajus"
    },
    "added": [
      "scripts/github-hook-test.coffee"
    ],
    "removed": [],
    "modified": []
  },
  "repository": {
    "id": 15725904,
    "name": "hubot-example",
    "url": "https://github.com/spajus/hubot-example",
    "description": "Examples for \"Automation and Monitoring with Hubot\" boo\
k.",
    "homepage": "https://leanpub.com/automation-and-monitoring-with-hubot",
    "watchers": 0,
    "stargazers": 0,
    "forks": 0,
    "fork": false,
    "size": 204,
    "owner": {
      "name": "spajus",
      "email": "tomas.varaneckas@gmail.com"
    },
    "private": false,
    "open_issues": 0,
    "has_issues": true,
    "has_downloads": true,
    "has_wiki": true,
    "language": "CoffeeScript",
    "created_at": 1389156986,
    "pushed_at": 1390718321,
    "master_branch": "master"
  },
  "pusher": {
    "name": "spajus",
    "email": "tomas.varaneckas@gmail.com"
  }
}
```

There is a load of information here, and we will make ourselves a script that extracts the interesting bits.

Listing And Deleting GitHub API Hooks

Before we start using those hooks, it's good to know how to list and remove them. To list all hooks on one GitHub repository, run this:

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  https://api.github.com/repos/spajus/hubot-example/hooks
[
  {
    "url": "https://api.github.com/repos/spajus/hubot-example/hooks/1730429",
    "test_url": "https://api.github.com/repos/spajus/hubot-example/hooks/1730\
429/test",
    "id": 1730429,
    "name": "web",
    ...
  }
]
```

To delete the hook, do a `DELETE` request providing hook `id`:

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  -X DELETE \
  https://api.github.com/repos/spajus/hubot-example/hooks/1730429
```

Pushes

There is a ready-made Hubot script for displaying pushes - github-commits.coffee , but we will create ourselves a new one that is a little more dynamic and uses `hubot-pubsub` for routing.

scripts/github-pubsub-pushes.coffee

```
# Description:
#   hubot-pubsub based GitHub push notifier
#
```

```
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /github/pushes/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/github/pushes/pubsub/:event", (req, res) ->
    res.end('')
    event = req.params.event
    try
      payload = req.body
      prefix = ">>> "
      if payload.commits.length > 0
        merge_commit = false
        author = payload.commits[0].author.name
        for commit in payload.commits
          if commit.author.name != author
            merge_commit = true
            break
        if merge_commit
          message = "#{prefix}#{payload.pusher.name} merged #{payload.commits\
.length} " +
                         "commits on #{payload.repository.name}:" +
                         "#{payload.ref.replace('refs/heads/', '')} " +
                         "(compare: #{payload.compare})"
          robot.emit 'pubsub:publish', event, message
          if payload.commits.length < 10
            for commit in payload.commits
              robot.emit 'pubsub:publish', event,
                             "  * #{commit.author.name}: #{commit.message} (#{com\
mit.url})"
        else
          message = "#{prefix}#{payload.commits[0].author.name} " +
                         "(#{payload.commits[0].author.username}) " +
                         "pushed #{payload.commits.length} commits to " +
                         "#{payload.repository.name}:#{payload.ref.replace('refs/h\
eads/', '')}"
          if payload.commits.length > 1
            message += " (compare: #{payload.compare})"
            robot.emit 'pubsub:publish', event, message
            for commit in payload.commits
              robot.emit 'pubsub:publish', event, "  * #{commit.message} (#{c\
ommit.url})"
          else
            robot.emit 'pubsub:publish', event, message
            for commit in payload.commits
              do (commit) ->
                robot.emit 'pubsub:publish', event, "  * #{commit.message} (#\
{commit.url})"
      else
        if payload.created
          if payload.base_ref
            base_ref = ': ' + payload.base_ref.replace('refs/heads/', '')
          else
```

```
              base_ref = ''
            robot.emit 'pubsub:publish', event, "#{prefix}#{payload.pusher.name\
} " +
                          "created: #{payload.ref.replace('refs/heads/', '')}#{bas\
e_ref}"
          if payload.deleted
            robot.emit 'pubsub:publish', event, "#{prefix}#{payload.pusher.name\
} " +
                          "deleted: #{payload.ref.replace('refs/heads/', '')}"
      catch error
        console.log "github-pubsub-push error: #{error}. Payload: #{req.body}"
```

Script source available at https://github.com/spajus/hubot-example

You will get something like this:

```
Tomas V.  hubot subscribe github.pushes
Hubot     Subscribed 585164 to github.pushes events
Hubot     github.pushes: >>> Tomas Varaneckas (spajus) pushed 1 commits to hu\
bot-example:master
          github.pushes: * Update hubot-pubsub to 1.0.0 (https://github.com/s\
pajus/hubot-example/commit/...)
```

This script will also tell you about tags and branches.

Issues

It is possible to receive a Hubot notification whenever somebody opens or closes an issue on GitHub. It is configurable per repo.

First we create a hook for the script. It will be listening on `/github/issues/pubsub/:event`.

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  -d '{"name":"web","active":true,"events":["issues"],\
  "config":{"url":"http://botserv.your.org:8080/github/issues/pubsub/github.i\
ssues",\
  "content_type":"json"}}' \
  https://api.github.com/repos/spajus/hubot-example/hooks
```

Now, the script:

scripts/github-pubsub-issues.coffee

```
# Description:
#   An HTTP Listener that notifies about new Github issues
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /github/issues/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/github/issues/pubsub/:event", (req, res) ->
```

```
    res.end("")

    event = req.params.event

    announceIssue req.body, (data) ->
      robot.emit 'pubsub:publish', event, data


announceIssue = (data, cb) ->
  if data.action
    mentioned = data.issue.body.match(/(^|\s)(@[\w\-\/]+)/g)

    if mentioned
      unique = (array) ->
        output = {}
        output[array[key]] = array[key] for key in [0...array.length]
        value for key, value of output

      mentioned = mentioned.map (nick) -> nick.trim()
      mentioned = unique mentioned

      mentioned_line = "\nMentioned: #{mentioned.join(", ")}"
    else
      mentioned_line = ''

    cb "Issue #{data.action}: \"#{data.issue.title}\" " +
       "by #{data.issue.user.login}: #{data.issue.html_url}#{mentioned_line}"
```

Script source available at https://github.com/spajus/hubot-example

Subscribe your room to `github.issues` via hubot-pubsub and here's what you will get:

```
Tomas V.  hubot subscribe github.issues
Hubot     Subscribed 585164 to github.issues events
          github.issues: Issue opened: "Test the issue hook" by spajus: https\
://github.com/spajus/hubot-example/issues/1
          Mentioned: @spajus
Tomas V.  I'll go close this
Hubot     github.issues: Issue closed: "Test the issue hook" by spajus: https\
://github.com/spajus/hubot-example/issues/1
          Mentioned: @spajus
```

Pull Requests

Knowing about new pull requests as soon as they appear is essential for maintaining a healthy workflow. If you do pull requests, you must know how long can it take to wait for somebody to review and merge your pull request. It shouldn't be that way, and when pull request notifications start appearing in developer chatrooms, average response time drops tenfold.

Let's start by creating a new hook that will post data on `/github/pulls/pubsub/:event`.

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  -d '{"name":"web","active":true,"events":["pull_request"],\
  "config":{"url":"http://botserv.your.org:8080/github/issues/pubsub/github.p\
ulls",\
  "content_type":"json"}}' \
  https://api.github.com/repos/spajus/hubot-example/hooks
```

The script:

scripts/github-pubsub-pulls.coffee

```
# Description:
#   hubot-pubsub based GitHub pull request notifier
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /github/pulls/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/github/pulls/pubsub/:event", (req, res) ->

    event = req.params.event
    res.end("")

    announcePullRequest req.body, (data) ->
      robot.emit 'pubsub:publish', event, data

announcePullRequest = (data, cb) ->
  if data.action == 'opened'
    mentioned = data.pull_request.body.match(/(^|\s)(@[\w\-\/]+)/g)

    if mentioned
      unique = (array) ->
        output = {}
        output[array[key]] = array[key] for key in [0...array.length]
        value for key, value of output

      mentioned = mentioned.filter (nick) ->
        slashes = nick.match(/\//g)
        slashes is null or slashes.length < 2

      mentioned = mentioned.map (nick) -> nick.trim()
      mentioned = unique mentioned

      mentioned_line = "\nMentioned: #{mentioned.join(", ")}"
    else
      mentioned_line = ''

    cb "New pull request \"#{data.pull_request.title}\" " +
       "by #{data.pull_request.user.login}: " +
       "#{data.pull_request.html_url}#{mentioned_line}"
```

Script source available at https://github.com/spajus/hubot-example

Here’s how it looks in action:

```
Tomas V.  hubot subscribe github.pulls
Hubot     Subscribed 585163 to github.pulls events
Tomas V.  I'll go make a pull request now…
Hubot     github.pulls: New pull request "Add GitHub pull request notificatio\
n script"
          by spajus: https://github.com/spajus/hubot-example/pull/2
          Mentioned: @spajus, @other_responsible_guy
```

Team Add

When your company grows big, you may start worrying about repository permissions being given to wrong teams or people. Luckily, you Hubot can help. When new team gets added to a repository, `team_add` event can be fired. Let's write ourselves a script for that:

script/github-pubsub-team.coffee

```
# Description:
#   An HTTP Listener that notifies about repository team changes
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /github/team/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/github/team/pubsub/:event", (req, res) ->
    res.end("")
    event = req.params.event
    announceTeamChange req.body, (data) ->
      robot.emit 'pubsub:publish', event, data

  announceTeamChange = (data, cb) ->
    team = data.team.name
    team_perm = data.team.permission
    org = data.sender.login
    repo = data.repository.full_name
    cb "@#{org}/#{team} received #{team_perm} rights on #{repo}"
```

Script source available at https://github.com/spajus/hubot-example

Run this on every repository to enable `team_add` hooks:

```
hubot@botserv:~$ curl -H "Authorization: token <your_auth_token>" \
  -d '{"name":"web","active":true,"events":["team_add"],\
  "config":{"url":"http://botserv.your.org:8080/github/team/pubsub/github.tea\
m",\
  "content_type":"json"}}' \
  https://api.github.com/repos/<your_org>/<your_repo>/hooks
```

Subcribe a chatroom to `github.team` events, then add a team to repository and see what Hubot tells you:

```
Tomas V.   hubot subscribe github.team
Hubot      Subscribed 585163 to github.team events
Hubot      github.team: @example/devs received pull rights on example/app
```

Membership

Tracking repository permissions without organization is also possible. Here's the script:

script/github-pubsub-member.coffee

```
# Description:
#   An HTTP Listener that notifies about repo membership changes
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /github/member/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/github/member/pubsub/:event", (req, res) ->
    res.end("")
    event = req.params.event
    announceMemberChange req.body, (data) ->
      robot.emit 'pubsub:publish', event, data

  announceMemberChange = (data, cb) ->
    if data.action
      who = data.member.login
      by_who = data.sender.login
      repo = data.repository.full_name
      action = data.action
      cb "#{repo} membership change: @#{by_who} #{action} @#{who}"
```

Script source available at https://github.com/spajus/hubot-example

At the moment of writing, only `added` action was sent from GitHub. Still, it's pretty useful.

```
Tomas V.  hubot subscribe github.member
Hubot     Subscribed 585163 to github.member events
          github.member: spajus/hubot-example membership change: @spajus adde\
d @electrotek
```

Automatically Closing Old GitHub Issues

To prevent old open issues that nobody bothers to follow up with, we can make Hubot automatically run a daily check and close issues that had no activity for over a month.

The following script will be a little more advanced. It requires additional dependencies and configuration.

scripts/github-old-issues.coffee

```
# Description
#   Find and close old issues in GitHub
#
# Dependencies:
#   "githubot": "0.5.0"
#   "moment": "2.5.1"
#   "hubot-pubsub": "1.0.0"
#   "cron": "1.0.3"
#   "time": "0.10.0"
#
# Configuration:
#   HUBOT_GITHUB_TOKEN (optional, if you want to search in private repos)
#   HUBOT_GITHUB_ORG - your GitHub organization
#
# Commands:
#   hubot close old issues in <repo> - Close outdated issues in given repo
#
# Author:
#   spajus

# Override these with your target repos. Keep list empty if using org.
target_repos = [
  'spajus/hubot-example',
  'spajus/hubot-control'
]

# Override with your org. Keep blank if non relevant.
target_org = ''

# Set your time zone
timezone = 'America/Los_Angeles'

# Set desired time. 00 00 9 * * 1-5 is monday-friday at 9 AM.
cron_expression = '00 00 9 * * 1-5'

module.exports = (robot) ->

  github = require('githubot')(robot, apiVersion: 'preview')
  cronJob = require('cron').CronJob
  moment = require('moment')

  new cronJob(cron_expression, closeOldIssues, null, true, timezone)

  closeOldIssues = ->
    org = target_org || process.env.HUBOT_GITHUB_ORG
    if org
      robot.emit 'github:org:issues:close', org
    for repo in target_repos
      robot.emit 'github:repo:issues:close', repo

  robot.respond /close old issues (in )?(.+\/[^\s]+)/i, (msg) ->
    repo = msg.match[2]
    closeOldIssuesIn repo, (data) ->
      msg.send data

  robot.on 'github:org:issues:close', (org) ->
    github.get "/orgs/#{org}/repos", (data) ->
      for repo in data
        closeOldIssuesIn repo.full_name, (data) ->
          robot.emit 'pubsub:publish', 'github.issue.close', data

  robot.on 'github:repo:issues:close', (repo) ->
    closeOldIssuesIn repo, (data) ->
      robot.emit 'pubsub:publish', 'github.issue.close', data

  closeOldIssuesIn = (repo, cb) ->
    github.handleErrors (response) ->
```

```
        cb "Error: #{response.statusCode} #{response.error}. Repo: #{repo}"
      github.get "repos/#{repo}/issues?state=open", (data) ->
        reply = ''
        found = false
        old_time = moment().subtract 'months', 1
        for issue in data
          issue_time = moment issue.updated_at, 'YYYY-MM-DDTHH:mm:ssZ'
          if issue_time < old_time
            found = true
            post_data = { body: "Closing old issue: updated #{issue_time.fromNo\
w()}" }
            github.post "repos/#{repo}/issues/#{issue.number}/comments", post_d\
ata, (post_resp) ->
              console.log "Posted comment: #{post_resp.html_url}"
            close_data = { state: 'closed' }
            github.request 'PATCH', "repos/#{repo}/issues/#{issue.number}", clo\
se_data, (close_resp) ->
              console.log "Closed issue: #{close_resp.html_url}"
            reply = "#{reply}#{issue.title} (#{issue.html_url}) updated #{issue\
_time.fromNow()}\n"
        if found
          cb "Found #{data.length} open issues in #{repo}. Closed old ones:\n#{\
reply}"
        else
          cb "No old issues found in #{repo}"
```

Script source available at https://github.com/spajus/hubot-example

Before restarting Hubot with this script, make sure to change configuration in first couple of lines, and install the dependencies. Make sure to use `npm install --save` so the definitions will get automatically added to `package.json`. You will also have set use a valid auth token in using `HUBOT_GITHUB_TOKEN`. You can do that in `hubot.conf` if you've set up Hubot as described in this book.

```
npm install --save moment time cron githubot
```

To test if it works, you can manually trigger the closing of old issues by saying `hubot close old issues in some/repo`.

```
Tomas V.  hubot close old issues in spajus/hubot-example
Hubot     No old issues found in spajus/hubot-example
```

GitLab Integration

GitLab is an open source collaboration platform that you can host on your own servers. It is similar to GitHub and has a nice API, which we will explore and integrate with Hubot.

Creating Web Hooks

To create a web hook in GitLab, simply go to projet's settings and select "Web Hooks" in the menu. Then add web hook URL and tick the checkbox of the payload you want to be delivered.

Warning: GitLab has a bug that causes some events to get fired twice, therefore you may see some webhook event duplications.

Pushes

The following script will allow you to subscribe to repository push events using GitLab's Web Hook. To make it work, add a web hook with `http://<hubot.server>:<hubot.port>/gitlab/pushes/pubsub/gitlab.pushes` url to any GitLab repo.

scripts/gitlab-pubsub-pushes.coffee

```
# Description:
#   hubot-pubsub based GitLab push notifier
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# Commands:
#   None
#
# URLS:
#   POST /gitlab/pushes/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/gitlab/pushes/pubsub/:event", (req, res) ->
    res.end('')
    event = req.params.event
    try
      payload = req.body
      prefix = ">>> "
      if payload.commits.length > 0
        merge_commit = false
        author = payload.commits[0].author.name
        for commit in payload.commits
          if commit.author.name != author
            merge_commit = true
```

```
              break
          if merge_commit
            message = "#{prefix} merged #{payload.commits.length} " +
                      "commits on #{payload.repository.name}:" +
                      payload.ref.replace('refs/heads/', '')
            robot.emit 'pubsub:publish', event, message
            if payload.commits.length < 10
              for commit in payload.commits
                robot.emit 'pubsub:publish', event,
                           "  * #{commit.author.name}: #{commit.message} (#{com\
mit.url})"
          else
            message = "#{prefix}#{payload.commits[0].author.name} " +
                      "pushed #{payload.commits.length} commits to " +
                      "#{payload.repository.name}:#{payload.ref.replace('refs/h\
eads/', '')}"
            robot.emit 'pubsub:publish', event, message
            for commit in payload.commits
              robot.emit 'pubsub:publish', event, "  * #{commit.message} (#{com\
mit.url})"
      catch error
        console.log "gitlab-pubsub-pushes error: #{error}. Payload: #{req.body}"
```

Script source available at https://github.com/spajus/hubot-example

Example output:

```
Tomas V.  hubot subscribe gitlab.pushes
Hubot     Subscribed 585164 to gitlab.pushes events
Hubot     gitlab.pushes: >>> Tomas Varaneckas pushed 1 commits to Andromeda:m\
aster
          gitlab.pushes: * Improve planetary defense system (http://gitlab.bo\
tserv.org/spajus/andromeda/commit/1b8f2e53…)
```

Issues

To receive Issue notifications, begin by creating a web hook on your target repository. Set webhook url to `http://<hubot.server>:<hubot.port>/gitlab/pushes/issues/gitlab.issues` and check “Issues events”.

The script:

scripts/gitlab-pubsub-issues.coffee

```
# Description:
#   hubot-pubsub based GitLab issue notifier
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# Commands:
#   None
#
# Configuration:
#   GITLAB_ROOT_URL
#   GITLAB_API_TOKEN
#
# URLS:
#   POST /gitlab/issues/pubsub/<pubsub-event>
```

```
#
# Authors:
#   spajus

module.exports = (robot) ->
  api_root = "#{process.env.GITLAB_ROOT_URL}/api/v3"
  robot.router.post "/gitlab/issues/pubsub/:event", (req, res) ->
    res.end('')
    event = req.params.event
    try
      payload = req.body
      attribs = payload.object_attributes
      project_url = "#{api_root}/projects/#{attribs.project_id}"
      robot.http(project_url)
        .header('PRIVATE-TOKEN', process.env.GITLAB_API_TOKEN)
        .get() (err, res, body) ->
          body = JSON.parse(body)
          issue_url = "#{body.web_url}/issues/#{attribs.id}"
          issue_message = "#{payload.object_kind} #{attribs.state}: #{attribs\
.title}"
          message = "#{issue_message} (#{issue_url})"
          robot.emit 'pubsub:publish', event, message
    catch error
      console.log "gitlab-pubsub-issues error: #{error}. Payload: #{req.body}"
```

Script source available at https://github.com/spajus/hubot-example

You will have to configure it first. Find your “Private token” in “Account settings” page, then add `export GITLAB_ROOT_URL="http://gitlab.your.org"` and `export GITLAB_API_TOKEN=<your-private-token>` to `hubot.conf`, restart Hubot, subscribe to an appropriate event and give it a try:

Example output:

```
Tomas V.  hubot subscribe gitlab.issues
Hubot     Subscribed 585164 to gitlab.issues events
Hubot     gitlab.issues: issue opened: Squadron one out of fuel (http://gitla\
b.botserv.org/spajus/andromeda/issues/3
          gitlab.issues: issue closed: Squadron one out of fuel (http://gitla\
b.botserv.org/spajus/andromeda/issues/3)
```

Merge Requests

Merge Requests in GitLab is like Pull Requests in GitHub. And you can get notifications about them in your chat as well. The script is nearly identical to the one that handles Issues, but with a few slight differences. If GitLab API was a little more consistent, one script would do.

Merge Requests script’s webhook is set up just like in two scripts above, just check “Merge Request events”, and probably choose a different pubsub event name, like `gitlab.merges`.

You should also add `GITLAB_ROOT_URL` and `GITLAB_API_TOKEN` to your `hubot.conf`, as described in “Issues” section above. Now, the nearly identical script:

scripts/gitlab-pubsub-merges.coffee

```
# Description:
#   hubot-pubsub based GitLab merge notifier
#
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# Commands:
#   None
#
# Configuration:
#   GITLAB_ROOT_URL
#   GITLAB_API_TOKEN
#
# URLS:
#   POST /gitlab/merges/pubsub/<pubsub-event>
#
# Authors:
#   spajus

module.exports = (robot) ->
  api_root = "#{process.env.GITLAB_ROOT_URL}/api/v3"
  robot.router.post "/gitlab/merges/pubsub/:event", (req, res) ->
    res.end('')
    event = req.params.event
    try
      payload = req.body
      attribs = payload.object_attributes
      project_url = "#{api_root}/projects/#{attribs.target_project_id}"
      robot.http(project_url)
        .header('PRIVATE-TOKEN', process.env.GITLAB_API_TOKEN)
        .get() (err, res, body) ->
          body = JSON.parse(body)
          issue_url = "#{body.web_url}/merge_requests/#{attribs.id}"
          issue_message = "#{payload.object_kind} #{attribs.state}: #{attribs\
.title}"
          message = "#{issue_message} (#{issue_url})"
          robot.emit 'pubsub:publish', event, message
    catch error
      console.log "gitlab-pubsub-merges error: #{error}. Payload: #{req.body}"
```

Script source available at https://github.com/spajus/hubot-example

How it looks in action:

```
Tomas V.  hubot subscribe gitlab.merges
Hubot     Subscribed 585164 to gitlab.merges events
Hubot     gitlab.merges: merge_request opened: Add even more phasers (http://\
gitlab.botserv.org/spajus/andromeda/merge_requests/3)
```

These scripts should give you a decent starting point for deeper integration with GitLab's API. Try adding author's name to Merge Request notification. Hint, you will have to query `/users/#{author_id}` to get that data.

Building And Deploying With Hubot And Jenkins

If you have a team of developers, having instant feedback about broken and restored builds is invaluable. Most build servers send emails by default, but if you can make it do an HTTP request with build status after every build, you can forget those emails and do things more agile by making Hubot tell about broken builds in developer chatroom.

You will learn how to do this with Jenkins, which is one of the most popular continuous integration servers available.

If you're not using any build server in your company and do not have automated builds of the software you make, you should really start doing it. There is just no excuse for that.

Notifying About Broken Jenkins Builds

Jenkins doesn't have a capability to make HTTP requests with build status updates right out of the box, but there is Notification Plugin that is easy to install and configure. There is also jenkins-notifier script available for Hubot, that is designed to handle these notifications. You can simply install and use it, but for the sake of learning we will roll our own version and use Hubot PubSub for more flexible notification routing.

After installing Jenkins Notification Plugin, open your build configuration page and find Job Notifications section. There you should configure it like this:

```
Format: JSON
Protocol: HTTP
URL: http://botserv:8080/jenkins/status
```

If you would analyze the payload that comes from jenkins, it would look something like this:

```
{"name":"test",
 "url":"job/test/",
 "build":{
    "full_url":"http://jenkins/job/test/13/",
    "number":13,
    "phase":"FINISHED",
    "status":"SUCCESS",
    "url":"job/test/13/"
  }
}
```

Phase can be `STARTED`, `COMPLETED` or `FINISHED`, and status can be `SUCCESS`, `UNSTABLE` or `FAILURE`. All we have to do is handle these different statuses.

The script itself is pretty simple:

scripts/jenkins-pubsub-notifier.coffee

```
# Description:
#   An HTTP Listener that notifies about new Jenkins build failures
#
```

```
# Dependencies:
#   "hubot-pubsub": "1.0.0"
#
# URLS:
#   POST /jenkins/status
#
# Authors:
#   spajus

module.exports = (robot) ->

  robot.router.post "/jenkins/status", (req, res) ->
    @failing ||= []
    res.end('')
    data = req.body
    return unless data.build.phase == 'FINISHED'

    if data.build.status == 'FAILURE' || data.build.status == 'UNSTABLE'
      if data.name in @failing
        broke = 'still broken'
      else
        broke = 'just broke'
        @failing.push data.name
      message = "#{broke} #{data.name} " +
        "#{data.build.display_name} (#{data.build.full_url})"

    if data.build.status == 'SUCCESS'
      if data.name in @failing
        index = @failing.indexOf data.name
        @failing.splice index, 1 if index isnt -1
        message = "restored #{data.name} " +
          "#{data.build.display_name} (#{data.build.full_url})"

    if message
      event = "build.#{data.build.status}"
      robot.emit 'pubsub:publish', event, message
```

Script source available at https://github.com/spajus/hubot-example

Now subscribe a room to `build` events and try to break something:

```
Tomas V.  hubot subscribe build
Hubot     Subscribed 585164 to build events
Hubot     build.FAILURE: just broke test #11 (http://jenkins/job/test/11/)
Hubot     build.FAILURE: still broken test #12 (http://jenkins/job/test/12/)
Hubot     build.SUCCESS: restored test #13 (http://jenkins/job/test/13/)
```

Executing Jenkins Builds

It’s easy to make Hubot trigger Jenkins builds. First, you have to enable remote build triggering in your Jenkins job. To do that, go to job configuration and under “Build Triggers” select “Trigger builds remotely (e.g., from scripts)” option, then enter the “Authentication Token” - it can be any string. For our script to work universally across all Jenkins builds, use the same Authentication Token for all your builds. Note that if you omit the “Authentication Token”, Jenkins will not save “Trigger Builds Remotely” as checked.

Under the “Authentication Token” there will be instructions of how to trigger the build:

> Use the following URL to trigger build remotely: JENKINS_URL/job/test/build?token=TOKEN_NAME or /buildWithParameters?token=TOKEN_NAME Optionally append &cause=Cause+Text to provide text that will be included in the recorded build cause.

Then test if it works:

```
hubot@botserv:~$ curl -v "http://jenkins/job/test/build?token=test"
> GET /job/test/build?token=test HTTP/1.1
> User-Agent: curl/7.30.0
> Host: jenkins
> Accept: */*
>
< HTTP/1.1 201 Created
< Date: Fri, 21 Feb 2014 04:11:24 GMT
< Content-Length: 0
< Connection: keep-alive
< Location: http://jenkins/queue/item/792/
```

As you can see, Jenkins added build to queue and returned it in `Location` header along with response. You can use Jenkins API to query the information:

```
hubot@botserv:~$ curl "http://jenkins/queue/item/792/api/json"
```

There are two possible response types, one when job is enqueued:

```
{ "actions":[{"causes":[{"shortDescription":"Started by remote host 127.0.0.1\
"}]}],
"blocked":false,
"buildable":false,
"id":795,
"inQueueSince":1392957015732,
"params":"",
"stuck":false,
"task":{"name":"test",
          "url":"http://jenkins/job/test/",
          "color":"blue"},
"url":"queue/item/795/",
"why":"In the quiet period. Expires in 4.2 sec",
"timestamp":1392957035732}
```

And one when job execution starts right away:

```
{"actions":[{"causes":[{"shortDescription":"Started by remote host
127.0.0.1"}]}],
"blocked":false,
"buildable":false,
"id":806,
"inQueueSince":1392958094707,
"params":"",
"stuck":false,
"task":{"name":"test",
          "url":"http://jenkins/job/test/",
          "color":"blue_anime"},
"url":"queue/item/795/",
"why":null,
"cancelled":false,
"executable":{"number":30,
              "url":"http://jenkins/job/test/30/"}}
```

There are some subtle differences, so we will have to be sure to cover both scenarios in our scripts.

It’s also possible to query all Jenkins queue items:

```
hubot@botserv:~$ curl "https://jenkins.vinted.net/queue/api/json"
{"items":[...]}
```

We can remove build from the queue using `POST` request to `/queue/cancelItem?id=<item_id>`:

```
hubot@botserv:~$ curl -X POST "http://jenkins/queue/cancelItem?id=792"
```

It’s a pretty powerful API, so let’s befriend it with Hubot.

To begin with, let’s write a script that allows triggering Jenkins builds from chatroom by providing build name:

scripts/jenkins-builder.coffee

```
# Description
#   Triggers Jenkins jobs from chatroom
#
# Configuration:
#   HUBOT_JENKINS_URI - Base Jenkins URI
#   HUBOT_JENKINS_BUILD_TOKEN - Token for triggering Jenkins builds
#
# Commands:
#   hubot build <job> - build Jenkins job by name
#
# Author:
#   spajus

module.exports = (robot) ->

  jenkins_uri = process.env.HUBOT_JENKINS_URI
  build_token = process.env.HUBOT_JENKINS_BUILD_TOKEN

  robot.respond /build (.+)/i, (msg) ->
    job = msg.match[1]
    url = "#{jenkins_uri}/job/#{encodeURI(job)}/build"
    msg.robot.http(url).query(token: build_token).get() (err, res, body) ->
      item_url = res.headers.location
      msg.robot.http("#{item_url}api/json").get() (err, res, body) ->
        data = JSON.parse body
        if data.executable
          msg.send "Building #{data.task.name} (#{data.executable.url})"
        else if data.task
          msg.send "Added #{data.task.name} (#{data.task.url}) to build queue\
: #{data.why}"
        else
          msg.send "Building #{data.name} (#{data.url})"
```

You will have to set `HUBOT_JENKINS_URI` and `HUBOT_JENKINS_BUILT_TOKEN` in `hubot.conf` first, or just provide it right in the script:

```
  jenkins_uri = process.env.HUBOT_JENKINS_URI || 'http://your-jenkins.net'
  build_token = process.env.HUBOT_JENKINS_BUILD_TOKEN || 'secret12345'
```

The execution works like this:

```
Tomas V.   hubot build test
Hubot      Building test (http://jenkins/job/test/33/)
Tomas V.   hubot build test
Hubot      Added test (http://jenkins/job/test/) to build queue:
           In the quiet period. Expires in 19 sec
```

Passing Parameters To Jenkins Builds

To get more flexibility, you can pass parameters to Jenkins builds. That becomes handy if you, for example, want to specify the branch you want to build. Let’s make a parameterized Jenkins build first. To do that, in Jenkins job configuration check “This build is parameterized” and click “Add Parameter” button. Create a “String parameter” with following settings:

```
Name:          branch
Default Value: master
Description:   Git branch to build
```

You can now refer to this parameter using `$branch` anywhere in your job configuration. Go ahead to “Source Code Management” and set “Branches to build” value to `$branch`.

Save the job configuration, and you should notice that “Build” link is now named “Build with Parameters”. If you click it, build doesn’t start instantly - you have to specify the branch, witch is prefilled with “master”. Try if it builds the right branch first, and if you got everything right, move on to Hubot script.

We will add a new command to the script we just created. This is how it will look like: `hubot build job-name branch=test`.

You may notice, that Hubot already responds to `/build (.+)/i`, that will match `hubot build job-name branch=test`, so we have to change the regex of our first command. Jobs names are usually alphanumeric with dashes or underscores, so this should work:

```
  robot.respond /build ([\w_-]+)$/i, (msg) ->
```

Tip: to quickly test if a piece of Hubot script works as you expect it to, run `coffee` from command line and you will get an interactive shell where you can do this:

```
hubot@botserv:~/campfire$ coffee
coffee> command = "hubot build Some_Job-1"
'hubot build Some_Job-1'
coffee> command.match /build ([\w_-]+)$/i
[ 'build Some_Job-1',
  'Some_Job-1',
  index: 6,
  input: 'hubot build Some_Job-1' ]
coffee> command = "hubot build job-name foo=bar"
'hubot build job-name foo=bar'
coffee> command.match /build ([\w_-]+)$/i
null
```

Now we need a regular expression that would match parameterized jobs. Let’s continue to use the coffee shell to create one and try it out:

```
coffee> command = "hubot build job-name foo=bar branch=master"
'hubot build job-name foo=bar branch=master'
coffee> matches = command.match /build ([\w_-]+) (.+)$/i
[ 'build job-name foo=bar branch=master',
  'job-name',
  'foo=bar branch=master',
  index: 6,
  input: 'hubot build job-name foo=bar branch=master' ]
coffee> matches[1]
'job-name'
coffee> matches[2]
'foo=bar branch=master'
```

We can see that `/build ([\w_-]+) (.+)$/i` gives us job name in `matches[1]` and all the parameters in `matches[2]`. We can split the parameters on whitespace and build a dictionary by splitting each piece by `=`. Let's put our new Hubot command together:

```
  robot.respond /build ([\w_-]+) (.+)$/i, (msg) ->
    job = msg.match[1]
    params = msg.match[2].split /\s+/
    query = { token: build_token }
    for param in params
      [k, v] = param.split '='
      query[k] = v
    url = "#{jenkins_uri}/job/#{encodeURI(job)}/buildWithParameters"
    msg.robot.http(url).query(query).get() (err, res, body) ->
      item_url = res.headers.location
      msg.robot.http("#{item_url}api/json").get() (err, res, body) ->
        data = JSON.parse body
        if data.executable
          msg.send "Building #{data.task.name} (#{data.executable.url})"
        else if data.task
          msg.send "Added #{data.task.name} (#{data.task.url}) to build queue\
: #{data.why}"
        else
          msg.send "Building #{data.name} (#{data.url})"
```

You will notice a considerable amount of code duplication between this new command and the one that triggers a job without parameters. Try combining these two actions into one to eliminate the duplication.

Deploying With Jenkins

If you are eager to start deploying your app with `hubot deploy`, wait until you read this. While it's faily simple to make deployments directly from Hubot script, you should make Jenkins run your deployments instead, and then trigger Jenkins job with Hubot.

Why the extra step? Jenkins gives you several advantages:

1. Live console output. If you tried to output your build logs directly to chatroom though Hubot, it could get ugly. Build logs can easily grow large, and chatroom is no place for such things. With Jenkins build you get to have a link that displays live console output and keeps it archived next to every build.
2. A history of all your deployments, including console output of each and every one of them. Combine it with good performance monitoring tool (like NewRelic, and you

will be able to pinpoint and eliminate performance bottlenecks right away, minutes after they are rolled out.

3. Build trends view. It will show you how deployment duration changes over time, and how often your builds tend to fail. This is good, because you will notice when you start going down the hill with poor speed and quality.

How deployment can be implemented with Jenkins depends heavily on your technology stack and infrastructure, this is why I cannot give you a ready to use recipe for it. You should be able to build it yourself using the knowledge from this book.

Good news is that everything you can do manually over SSH connection, Jenkins can do it for you - in “Build” section add a step called “Execute shell” and you can do things like this:

```
Command:
ssh deployer.your.org "STARTED_BY=$user BRANCH=$branch /opt/deploy.sh"
```

Now you can use Hubot to invoke this parameterized Jenkins build. Use `msg.message.user.name` to get the name of person who started the deployment.

To make it fully integrated, implement feedback from your deployment script back to Hubot. You can find some ideas how to do that in “Monitoring With Hubot” chapter of this book. Just trigger pubsub messages over HTTP to notify Hubot about deployment start, end, and possible failures.

Invoking Chef’s Knife With Hubot

Chef is one of most popular weapons of choice for managing IT infrastructure as code. And what could be better than controlling chef directly from your chat using Hubot? In this chapter we will write a custom Hubot script that will be running knife commands for you.

Configuring Knife

To begin the integration, you have to get `knife` command to work on your bot server. For that you need to install `chef` gem and get .pem files and knife.rb file.

You have a couple of options here, one is to make `knife` work from any directory by placing the configuration at `/etc/chef`, other is to have your `*.pem` and `knife.rb` in some directory from where knife command will work without being available globally. Our example script will be running knife from `/home/hubot/knife` directory.

Hubot Chef Script

Here is a quick example of Hubot script with one, yet very powerful chef’s `knife` command:

scripts/chef.coffee

```coffee
# Description
#   Hubot script that runs Chef's knife
#
# Commands:
#   hubot knife <command> - execute knife command
#
# Author:
#   spajus

module.exports = (robot) ->
  knife_opts = { cwd: '/home/hubot/knife' }
  cp = require 'child_process'
  robot.respond /knife (.*)/i, (msg) ->
    cp.exec "knife #{msg.match[1]}", knife_opts, (error, stdout, stderr) ->
      msg.send stdout if stdout
      msg.send "Error: #{stderr}" if stderr
```

If your `knife` command works globally, you may set `knife_opts` to `{}`.

This script is written using the technique explained in “Hubot Scripting” chapter - see the calendar script in “Reacting To Messages In Chatroom” section for explanation how invoking shell commands works with Hubot scripts.

Example of this script in action:

```
Tomas V.  hubot knife role list | grep jobs$
Hubot     analytics-jobs
          core-jobs
```

Yes, even pipe works. This script is so powerfull, that you MUST take precautions to secure it properly, or somebody can can do devastating things not only to hubot’s machine, but to all your servers. A good way is to execute `cp.exec` only when it’s invoked in `devops` or `admins` room or by small set of trusted users. See “Roles And Authentication” chapter to see how it can be done, and use it at your own risk.

Advanced Hubot Chef Script

While the script we’ve written is very powerful already, we may want to make our lives easier and let Hubot do the thinking. This is especially helpful if you want to expose a set of secure commands for your developers to perform actions without getting to know how `chef` and `knife` works. Behold, the advanced chef integration script, that includes `hubot knife` command.

scripts/chef.coffee

```coffee
# Description
#   Hubot script that runs Chef's knife
#
# Commands:
#   hubot knife <command> - execute knife command (only in devops chat)
#   hubot server list - list all our servers registered with chef
#   hubot server list <pattern> - list our servers registered with chef match\
ing a pattern
#   hubot server search <pattern> - search for servers matching chef role (* \
works)
#   hubot servers <pattern> - search for servers matching '*-<pattern>' chef \
role
#   hubot server roles - list all chef roles
#   hubot server roles <pattern> - list chef roles matching a pattern
#
# Author:
#   spajus

module.exports = (robot) ->

  cp = require 'child_process'
  knife_opts = { cwd: '/home/hubot/knife' }

  handle_response = (msg) ->
    (error, stdout, stderr) ->
      msg.send stdout if stdout
      msg.send "Error: #{stderr}" if stderr

  robot.respond /knife (.*)/i, (msg) ->
    if msg.message.room != '<insert devops room id>'
      msg.send "Do it in devops room please"
      return
    cp.exec "knife #{msg.match[1]}",
      knife_opts, handle_response(msg)

  robot.respond /server list$/i, (msg) ->
    cp.exec "knife node list",
      knife_opts, handle_response(msg)

  robot.respond /server list (.*)/i, (msg) ->
    cp.exec "knife node list | grep #{msg.match[1]}",
      knife_opts, handle_response(msg)

  robot.respond /server roles?$/i, (msg) ->
    cp.exec "knife role list",
      knife_opts, handle_response(msg)

  robot.respond /server roles? (.*)/i, (msg) ->
    cp.exec "knife role list | grep #{msg.match[1]}$",
      knife_opts, handle_response(msg)

  robot.respond /server search (.*)/i, (msg) ->
    cp.exec "knife search node 'roles:#{msg.match[1]}' -a run_list",
      knife_opts, handle_response(msg)

  robot.respond /servers (.*)/i, (msg) ->
    cp.exec "knife search node 'roles:*-#{msg.match[1]}' -i",
      knife_opts, handle_response(msg)
```

You can find this script at https://github.com/spajus/hubot-example.

Script in action:

```
Tomas V.  hubot help server
Hubot     hubot server list - list all our servers registered with chef
          hubot server list <pattern> - list our servers registered with chef\
 matching a pattern
          hubot server roles - list all chef roles
          hubot server roles <pattern> - list chef roles matching a pattern
          hubot server search <pattern> - search for servers matching chef ro\
le (* works)
          hubot servers <pattern> - search for servers matching '*-<pattern>'\
 chef role
Tomas V.  hubot server list indexer
Hubot     indexer.botserv.org
Tomas V.  hubot server list redis
Hubot     redis1.botserv.org
Tomas V.  hubot server list static
Hubot     static1.botserv.org
          static2.botserv.org
          static-x1.botserv.org
          static-x2.botserv.org
          static11.botserv.org
          static12.botserv.org
          static13.botserv.org
          static14.botserv.org
Tomas V.  hubot server roles redis
Hubot     core-redis
          redis
Tomas V.  hubot server search core-redis
Hubot     2 items found
          db3.botserv.org:
            run_list: role[machine], role[core-redis]
          redis1.botserv.org:
            run_list: role[machine], role[redis], role[core-redis]
Tomas V.  hubot servers uk
```

```
Hubot     14 items found
          indexer.botserv.org
          static2.botserv.org
          front2.botserv.org
          app4.botserv.org
          jobs9.botserv.org
          db4.botserv.org
          app1.botserv.org
          search2.botserv.org
          front1.botserv.org
          app2.botserv.org
          db1.botserv.org
          static1.botserv.org
          db2.botserv.org
          search1.botserv.org
```

Tune this script and add shortcuts to your favorite `knife` commands.

Third Party Hubot Chef Scripts

There are Hubot scripts out there that work nearly the same. For example, this package at GitHub: jjasghar/hubot-chef. It may be a good alternative if you want something working right out of the box, however, I believe Hubot Chef integration is too important, you can’t just throw something in and expect it will suit your needs. You should write every bit of it yourself, add safety precautions, enrich it with shorthands for most often used commands. By getting very intimate with this integration script you will be sure it does all you want, and you will be able to sleep at night knowing that you built it in a way that nobody can do any harm to your infrastructure.

Hubot With Puppet, Ansible, Or Something Else

This book will not cover these, however you can use same techniques that were used in Chef’s integration to write yourself a custom integration with any infrastructure management tool. Just make sure you follow through the chapter and understand how `chef.coffee` works.

There Are No Limits

You’ve reached the end of the book. I hope you enjoyed it, learned new things and got inspired to find new ways of using and extending Hubot.

By now you should be able to integrate it with anything that has an API or can make HTTP requests. The only limit is your imagination.

If you build something great, and I’m sure you will, consider releasing it as open source and sharing with the rest of us. Let’s make Hubot better together!