makeuseof

OpenHAB Beginner's Guide

part 2

by James Bruce

OpenHAB Beginner's Guide Part 2: ZWave, MQTT, Rules and Charting

Written by James Bruce

Published October 2015.

Read the original article here: http://www.makeuseof.com/tag/openhab-beginners-guide-part-2-zwave-mqtt-rules-charting/

Table of contents

Free doesn't always mean "not as good as paid", and OpenHAB is no exception. The open source home automation software far exceeds the capabilities of any other home automation system on the market – but it's not easy to get set up. In fact, it can be downright frustrating.

In part 1 of the guide, I walked you through installing OpenHAB on a Raspberry Pi, introduced the core concepts of OpenHAB, and showed you how to add your first items into the system. Today we'll be going further:

- Adding ZWave devices
- Adding a Harmony Ultimate controller
- Introducing rules
- Introducing MQTT, and installing an MQTT broker on your Pi, with sensors on an Arduino
- Recording data and graphing it

Watch: What Works Best With OpenHAB? Hue, Harmony, MQTT, and Z Wave Experiences

Introduction to Z-Wave

Z-Wave has been the dominant home automation protocol for years: it's reliable, has been extensively developed, and works over a much longer range than any other smart home products. There's hundreds of Z-Wave sensors available to you that perform a wide range of tasks. OpenHAB *can* work with Z-Wave, but is a hassle to set up, and reliability is not guaranteed.

If you're considering the purchase of a house full of Z-Wave sensors specifically for use with OpenHAB, I'd urge you to reconsider. It may work out great for you, or it may be plagued with small but persistent problems. At least, don't buy a house full of sensors until you've had a chance to try out a few. The only reason to choose Z-Wave is if you're not 100% settled on OpenHAB, and would like to leave your options open in future: Z-Wave for instance works with Samsung SmartThings hub, as well as Z-Wave specific hubs such as Homeseer, and a range of other software options such as Domoticz.

Though OpenHAB includes a Z-Wave binding, you still need to **configure the Z-Wave network first**, before OpenHAB can start querying it for data. If you've got a Rasberry controller board, you have some software supplied for configuring the network, so we won't be covering that here. If you bought an Aeotec USB Z-Stick controller or similar, you likely don't have any software included, so read on.

If you already have a Z-Wave network setup, you can just plug your controller into the Pi and start configuring the binding and items. If this is your first foray into Z-Wave, it's a little more complex.

First, on the hardware side: each controller has its own way of pairing with devices (technically known as "inclusion mode" in which a node ID is assigned). In the case of the Aotec Z-Stick, this means unplugging it from the USB port, and pressing the button once to place it into inclusion mode. Then take it near to the device you're pairing, and press the inclusion button on that too *(this will also vary: my Everspring socket requires the button to press 3 times in quick succession, so the lesson here is to read the manual for your device)*.

The Z-Stick flashes briefly to indicate success. This presents problems when plugging it back into the Pi, as a new port is assigned. Restart your Pi to have it reset back to the standard port if you find it's been dynamically reassigned a different one. Better still: don't plug it into the Pi until you've done all the hardware pairings first.

Installing HABmin and Z-Wave Bindings

Since OpenHAB doesn't actually a configuration utility for Z-Wave, we're going to install another web management tool which does – something called HABmin. Head on over to the HABmin Github repository download the current release. Once you've unzipped it, you'll find 2 **.jar** files in the addons directory – these should placed in the corresponding addons directory in your OpenHAB Home share *(if you're also using the Aotec gen5 Z-Stick, make sure you've got at least version 1.8 of the Z-Wave binding)*.

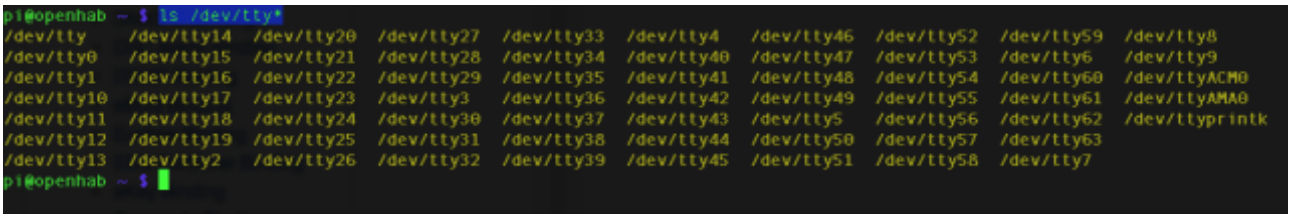
Next, create a new folder in the webapps directory, and called it "habmin" (lowercase is important). Copy the rest of the downloaded files into there.

Note: There's also a HABmin 2 under active development. Installation is much the same but with one additional .jar addon. It might be worth trying both just to see which you prefer.

If you haven't already, plug your controller into your Pi. Type the following to find the correct port.

```
ls /dev/tty*
```

You're looking for anything with USB in the name, or in my particular case, the Z-stick presented itself as **/dev/ttyACM0** (a modem). It might be easier to do the command once before you plug it in, and once after, so you can see what changes if you're unsure.



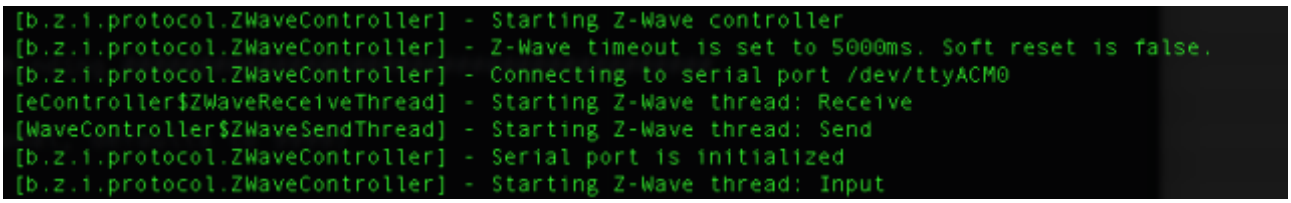
Open up the OpenHAB config file and modify the section on Z-Wave, uncommenting both lines and putting your actual device address. One final step for me was to allow the OpenHAB user to access the modem. This is only necessary if your controller is coming up as /dev/ttyACM0, not /dev/ttyUSB*.

```
sudo usermod –a –G dialout openhab
```

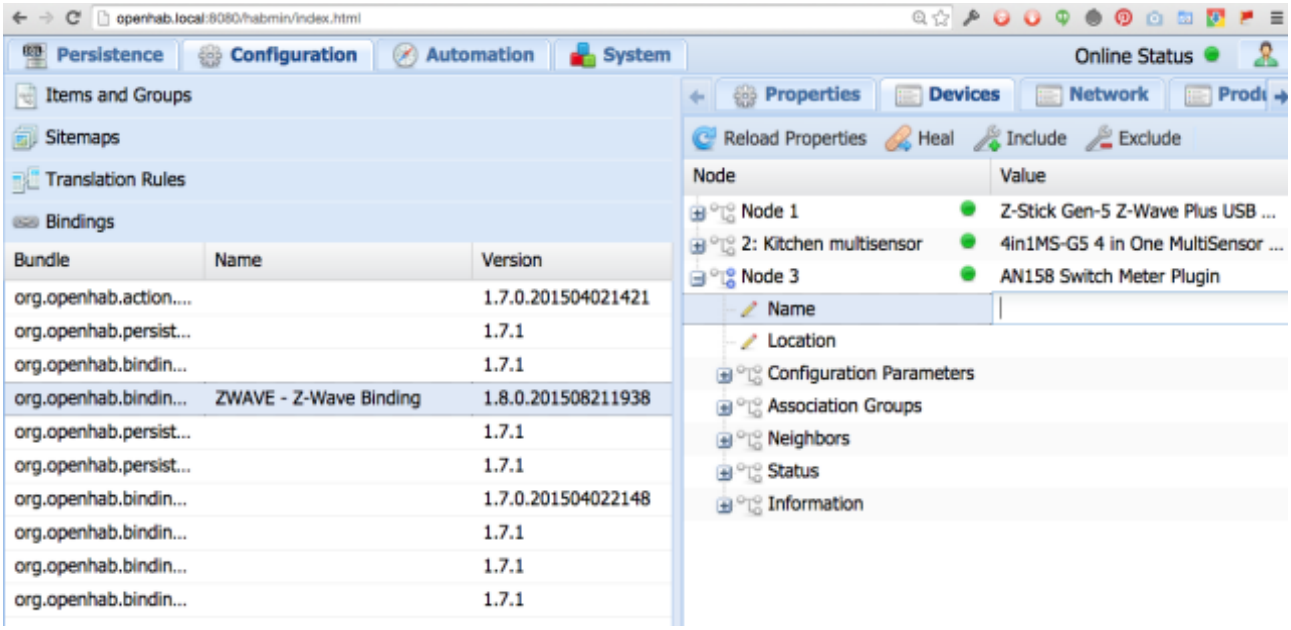
Now, to kick everything into action, restart OpenHAB

```
sudo service openhab restart
```

Hopefully, if you're checking the debug log, you'll see something like this. Congratulations, you're now talking Z-Wave. You may also find the debug log flooded with messages from various Z-Wave nodes. Let's start by checking HABMIN to see what it's found: http://openhab.local:8080/habmin/index.html (replacing openhab.local with your Raspberry Pi hostname or IP address).



There's a lot to see in HABMIN, but we're only really concerned with the **Configuration -> Bindings -> Z-Wave -> Devices** tab, as you can see below. Expand the node to edit the location and name label for your ease of reference.



Configuring Z-Wave Items

Each Z-Wave device will have a specific configuration for OpenHAB. Thankfully, most devices have already been explored and there will be examples out there for yours already. Configuring custom devices that aren't recognized is well beyond the scope of this guide, but let's assume it is supported for now.

First, I've got a basic Everspring AN158 power switch and meter on Node 3. A quick Googling led me to a blog post on Wetwa.re, with a sample item configuration. I adapted this as follows:

```
Switch Dehumidifier_Switch
"Dehumidifier" {zwave="3:command=switch_binary"}

Number Dehumidifier_Watts "Dehumidifier power consumption [%.1f
W]" { zwave="3:command=meter" }
```

Perfect.

Next up is an Aeotec Gen5 Multi-Sensor.

For this one, I found a sample config at iwasdot.com, and my multisensor is on Node 2.

```
Number Hallway_Temperature "Hallway Temperature [%.1f °C]"  (Hallway,
Temperature)
{zwave="2:0:command=sensor_multilevel,sensor_type=1,sensor_scale=0"}

Number Hallway_Humidity "Hallway Humidity    [%.0f %%]"  (Hallway,
Humidity) {zwave="2:0:command=sensor_multilevel,sensor_type=5"}
```

```
Number Hallway_Luminance "Hallway Luminance    [%.0f Lux]"

(Hallway) {zwave="2:0:command=sensor_multilevel,sensor_type=3"}

Contact Hallway_Motion "Hallway Motion [%s]" (Hallway, Motion)
{zwave="2:0:command=sensor_binary,respond_to_basic=true"}

Number sensor_1_battery "Battery [%s %%]" (Motion)
{zwave="2:0:command=battery"}
```

If the format of this looks strange to you, please head on back to the first beginner's guide, specifically the Hue binding section, where I explain how items are added. You'll probably only ever need to copy paste examples like this, but in case you have a new device, the binding documentation details all the commands.

Logitech Harmony Binding

Before we jump into rules, I wanted to add a quick note about working with the Harmony binding. I'm a big fan of the Harmony series of ultimate remotes to simplify the home media center experience, but they often stand as a separate system within the smart home. With OpenHAB, Logitech Harmony activities and full device control can now be a part of your centralised system, and even included in automation rules.

Begin by installing the three binding files that you find by using apt-cache to search for "harmony":

```
pi@openhab ~ $ apt-cache search harmony
abcmidi - converter from ABC to MIDI format and back
concordance - Harmony remote configuration tool
congruity - graphical utility to configure Logitech Harmony remotes
libconcord-dev - Harmony remote configuration tool - development files
libconcord2 - Harmony remote configuration tool - runtime libraries
python-libconcord - Harmony remote configuration tool - Python bindings
openhab-addon-action-harmonyhub - openHAB HarmonyHub Action
openhab-addon-binding-harmonyHub - openHAB HarmonyHub Binding
openhab-addon-io-harmonyhub - openHAB Harmony Client IO
pi@openhab ~ $
```

Don't forget to **chown** the bindings directory again when you're done:

```
sudo apt-get install openhab-addon-action-harmonyhub

sudo apt-get install openhab-addon-binding-harmonyhub

sudo apt-get install openhab-addon-io-harmonyhub

sudo chown -hR openhab:openhab /usr/share/openhab
```

To configure the binding, open up the openhab.cfg file and add a new section as follows:

```
########## HARMONY REMOTE CONTROLS ##########

harmonyhub:host=192.168.1.181 or your ip

harmonyhub:username=your-harmony-email-login

harmonyhub:password=your-password
```

The IP address is that of your Harmony hub. Use a network scanner to find that out. You'll also need to enter your login details, the ones you enter when you launch the standard Harmony config utility. That's it. Upon restarting your Hue, your debug log should have a sudden burst of output from the binding.

This is a JSON formatted list of all your activities, devices, and commands that can be sent. It's a good idea to copy this out for future reference. you can make it even easier to read with collapsible nodes by pasting into an online JSON formatter such as this one.

As well as the standard PowerOff activity which is a default, you'll find your own defined activities listed here by name. Now let's create a simple one button control to start activities. First, in your items file, add the following line. Change the group and icon if you like.

```
/* Harmony Hub */

String Harmony_Activity "Harmony [%s]" <television> (Living_Room)
{harmonyhub="*[currentActivity]" }
```

This is a **two-way String binding**, which is able to both fetch the current activity, and command the current activity to be something else. Now we can create a button for it, in the sitemap file.

```
Switch item=Harmony_Activity
mappings=[PowerOff='Off',Exercise='Exercise',
13858434='TV',Karaoke='Karaoke']
```

In the square bracket you'll see each activity along with the label. Generally you can refer directly to activities as you've named them on your remote, but the exception to this I found, was anything with a space in the activity name, such as "Watch TV". In this case, you'll need to use the activity ID. Again, you can find the ID in the JSON debug output. Save and refresh your interface, you should see something similar to this:

You can also refer to activities in your rules, as we'll see next. Read the wiki page for more info on the Harmony binding.

A General Introduction to Rules

Most smart home hubs include some kind of rules creation so you can automatically react to sensor data and events in the home. In fact, I'd argue that a truly smart home isn't one you need to spend time interacting with mobile apps – it's one that's invisible to the end user and completely automated. To this end, OpenHAB also includes a powerful rules scripting language that you can program, far exceeding the complexity of most smart home hubs or IFTTT recipes.

Programming rules sounds worse than it is. Let's start simple with a pair of rules that turn on or off the light depending on the presence sensor:

```
rule "Office light on when James present"

when

 Item JamesInOffice changed from OFF to ON

then

 sendCommand(Office_Hue,ON)

end



rule "Office light off when James leaves"

when
```

```
     Item JamesInOffice changed from ON to OFF

    then

     sendCommand(Office_Hue,OFF)

    end
```

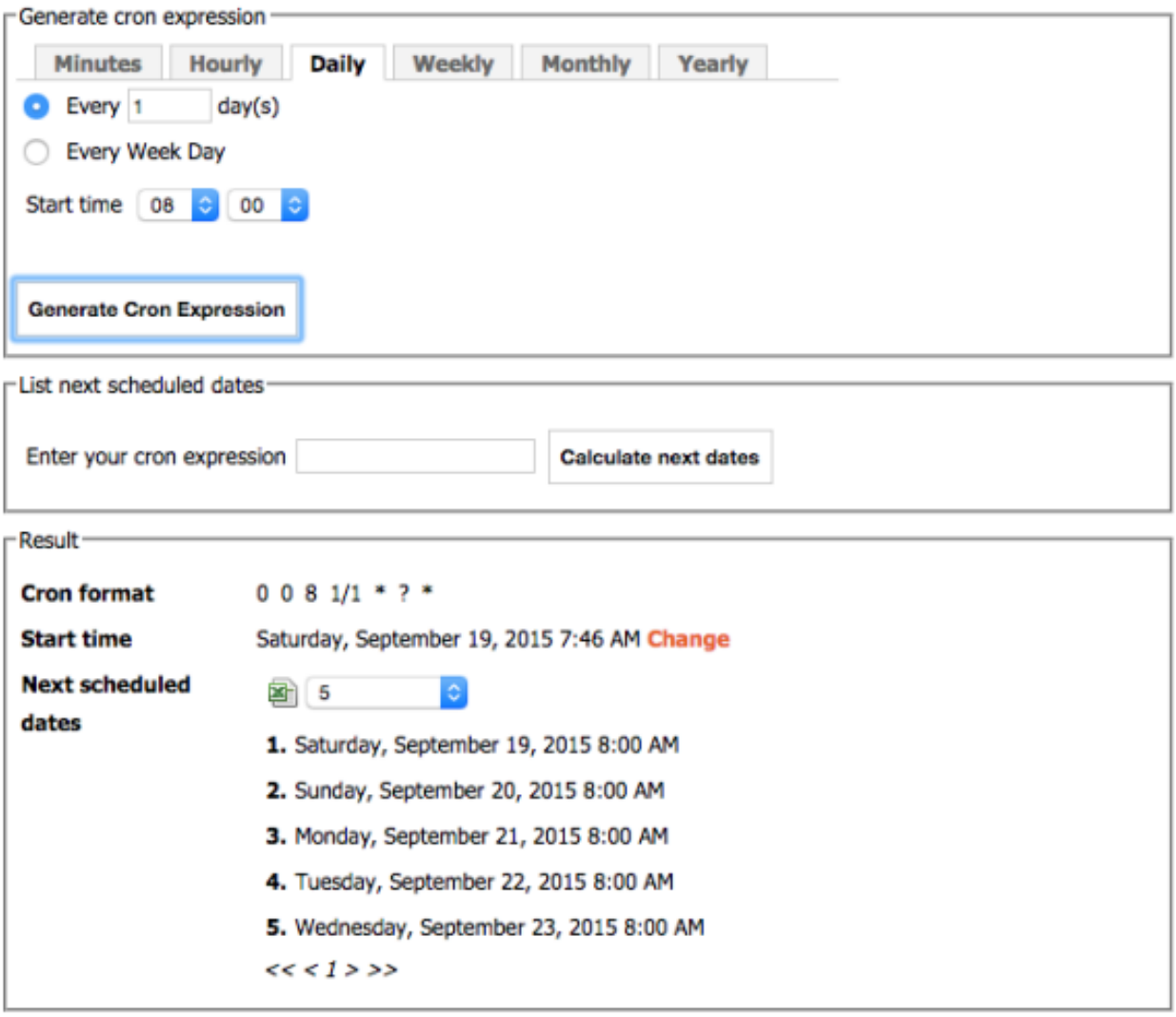
First, we name the rule – be descriptive, so you know what event is firing. Next, we define our simple rule by saying *when x is true, then do y*. End signifies the closure of that particular rule. There's a number of special words you can use in rules, but for now we're dealing with two simple bits of syntax – **Item**, which allows you to query the state of something; and **sendCommand**, which does exactly what you think it will. I told you this was easy.

It's probably unnecessary to use a pair of rules, but as my logic gets more complex it'll be beneficial to have them separate for whether I'm entering or leaving the area – and it might be a good idea to add a light sensor somewhere into the equation so we're not unnecessarily turning on lights.

Let's look at another example to create a scheduled rule.

```
    rule "Exercise every morning"

    when

         Time cron "0 0 8 1/1 * ? *"

    then

         harmonyStartActivity("Exercise")

    end
```

Again, we name the rule, state conditions when it should fire, and the actions to take. But in this case, we're defining a Time pattern. The funny code you see in the quotes is a CRON expression for Quartz Scheduler (the format is slightly different to a regular CRONtab). I used cronmaker.com to help create the expression, but you can also read the format guide for a detailed explanation and more examples.



My rules says simply “8am every morning, every day of the week, tell my Harmony Ultimate system to start the Exercise activity”, which in turn activates the TV, the Xbox, the amplifier, and presses the A button after a minute to launch the disk in the drive.

Sadly, OpenHAB isn’t yet able to do the exercise for me.

One more rule I want to show you is something I use to manage the humidity levels in my home. I have a single dehumidifier which I need to move around wherever needed, so I decided to look at all of my humidity sensors, find which one is the highest, and store that in a variable. It’s currently triggered every minute, but that can easily be lowered. Take a look first:

```
import org.openhab.core.library.types.*

import org.openhab.model.script.actions.*

import java.lang.String



rule "Humidity Monitor"
```

```
when Time cron "0 * * * * ?"

then

 var prevHigh = 0

 var highHum = ""



 Humidity?.members.forEach[hum|

       logDebug("humidity.rules", hum.name);

       if(hum.state as DecimalType > prevHigh){

            prevHigh = hum.state

            highHum = hum.name + ": " + hum.state + "%"

       }

 ]

    logDebug("humidity.rules", highHum);

    postUpdate(Dehumidifier_Needed,highHum);

end
```

The core of rule is in the **Humidity?.members.foreach** line. Humidity is a group name for my humidity sensors; **.members** grabs all of the items in that group; **foreach** iterates over them (with a curious square bracket format you're probably not familiar with). The syntax of rules is a derivative of Xtend, so you can read the Xtend documentation if you can't find an example to adapt.

You probably won't need to though – there are hundreds of example rules out there:

- Detailed explanation of rules on the official wiki
- The official rules samples wiki page
- Taking rules to new heights
- Advanced samples at IngeniousFool.net

MQTT for OpenHAB and Internet of Things

MQTT is a lightweight messaging system for machine-to-machine communication – a kind of Twitter for your Arduinos or Raspberry Pis to talk to each other (though of course it works with much more than just those). It's rapidly gaining in popularity and finding itself a home with Internet of Things devices, which are typically low resource micro-controllers that need a reliable way to transmit sensor data back to your hub or receive remote commands. That's exactly what'll we'll be doing with it.

But why reinvent the wheel?

MQ Telemetry Transport was invented way back in 1999 to connect oil pipelines via slow satellite connections, specifically designed to minimise battery usage and bandwidth, while still providing reliable data delivery. Over the years the design principles have remained the same, but the use case has shifted from specialised embedded systems to general Internet of Things devices. In 2010 the protocol was released royalty free, open for anyone to use and implement. We like free.

You might be wondering why we're even bothering with yet another protocol – we already have the HTTP after all – which can be used to send quick messages between all manner of web connected systems (like OpenHAB and IFTTT, particular with the new maker channel). And you'd be right. However, the processing overhead of an HTTP server is quite large – so much so that you can't easily run one on an embedded microcontroller like the Arduino (at least, you can, but you won't have much memory left for anything else). MQTT is the other hand is lightweight, so sending messages around your network won't clog the pipes up, and it can easily fit into our little Arduino memory space.

Watch: Getting started with MQTT

How does MQTT Work?

MQTT requires both a server (called a "broker") and one or more clients. The server acts as a middleman, receiving messages and rebroadcasting them to any interested clients.

Let's continue with the *Twitter-for-machines* analogy though. Just as Twitter users can tweet their own meaningless 140 characters , and users can "follow" other users to see a curated stream of posts, MQTT clients can subscribe to a particular channel to receive all messages from there, as well as publish their own messages to that channel. This publish and subscribe pattern is referred to as **pub/sub**, as opposed to the tradition **client/server** model of HTTP.

HTTP requires that you reach out to the machine you're communicating with, say Hello, then have a back and forth of constantly acknowledging each other while you get or put data. With pub/sub, the client doing the publishing doesn't need to know which clients are subscribed: it just pumps out the messages, and the broker redistributes them to any subscribed clients. Any client can both publish, and subscribe to topics, just like a Twitter user.

Unlike Twitter though, MQTT isn't limited to 140 characters. It's data agnostic, so you can send small numbers or large text blocks, JSON-formatted datagrams, or even images and binary files.

It isn't that MQTT is better than HTTP for everything – but it *is* more suitable if we're going to have lots of sensors all around the house, constantly reporting in.

It's also important to know that OpenHAB will not act as your MQTT broker – we'll address that bit later. However, OpenHAB will act as a client: it can both publish your OpenHAB activity log, as well as bind particular channels to devices, so you can for instance have a switch that's controlled by MQTT messages on a particular channel. This is ideal for creating a house full of sensors.

Install Mosquitto on Your Pi

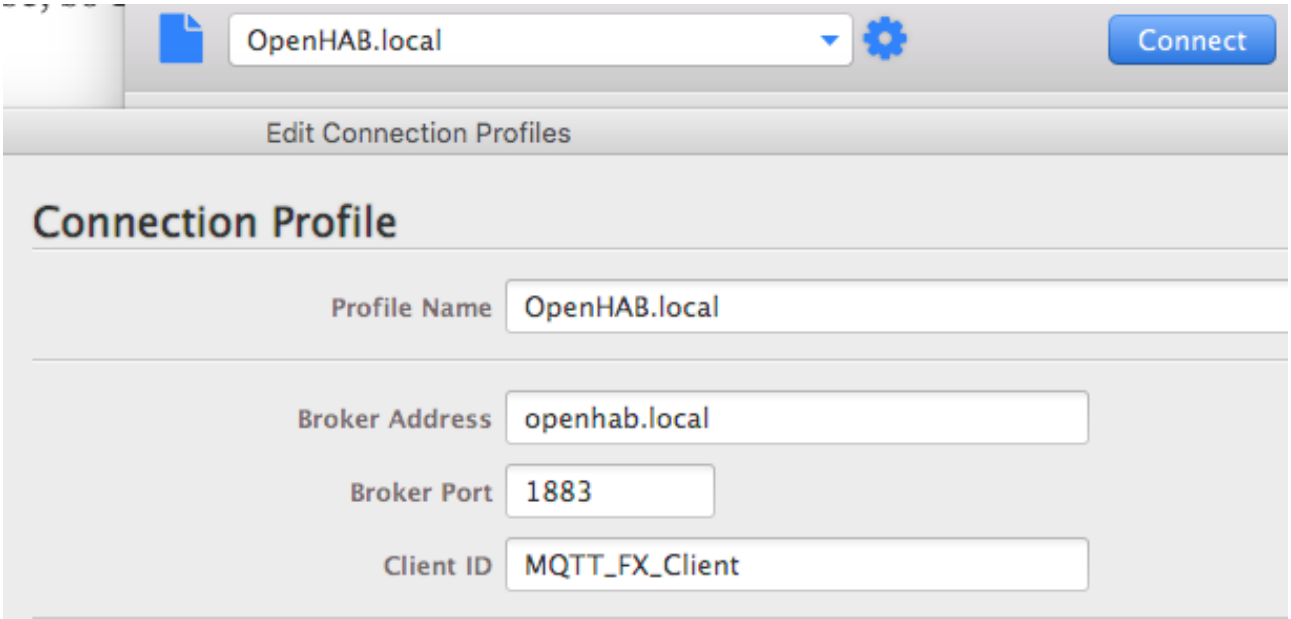
Although OpenHAB includes an MQTT client so you can subscribe to a topic and also publish messages, it won't act as the server. For that, you either need to use a web based MQTT broker (paid or free), or install the free software on your Pi. I'd like to keep it all in-house, so I've installed Mosquitto on the Pi.

Unfortunately, the version available via the usual apt-get is completely out of date. Instead, let's add the latest sources.

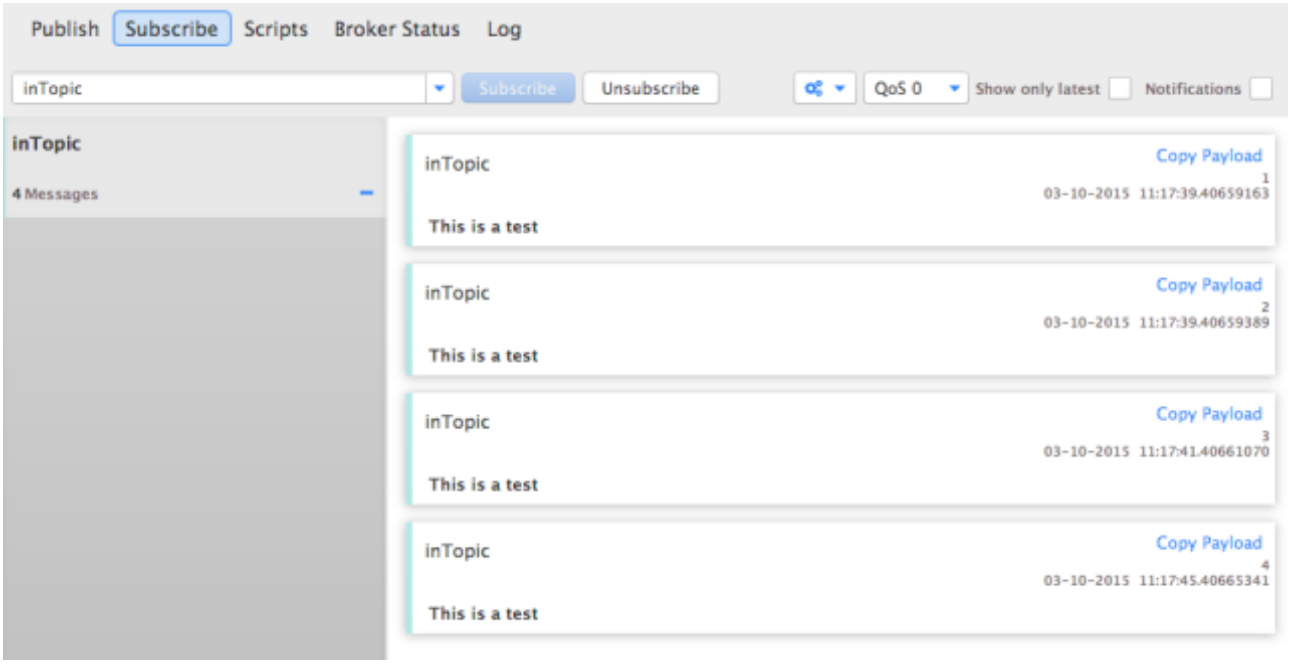
```
wget http://repo.mosquitto.org/debian/mosquitto-repo.gpg.key

sudo apt-key add mosquitto-repo.gpg.key

cd /etc/apt/sources.list.d/

sudo wget http://repo.mosquitto.org/debian/mosquitto-wheezy.list

sudo apt-get install mosquitto
```

That's all we need to do to have an MQTT server up and running on the local network. Your broker is running on port 1883 by default.

Check your MQTT server is working using the free MQTT.fx, which is cross-platform. Click the settings icon to create a new profile, and enter your Raspberry Pi's IP address or name. Save, and hit connect. If the little traffic light in the top right turns green, you're good to go.



For a quick test, click on the "subscribe" tab, and type **inTopic/** into the text box, then hit the **Subscribe** button. You're now subscribed to receive message on the topic named inTopic, though it'll be showing 0 messages. Go back to the publish tab, type inTopic into the small box, and a short message into the large text box below. Hit **Publish** a few times and look back on the subscribe tab. You should see a few messages having appeared in that topic.



Before we add some actual sensors to our network, we need to learn about topic levels, which enable us to structure and filter the MQTT network. Topic names are case-sensitive, shouldn't start with $, or include a space, or non-ASCII characters – standard programming practices for variable names, really.

The / separator indicates a topic level, which is hierarchical, for example the following are all valid topic levels.

```
inTopic/smallSubdivision/evenSmallerSubdivision

myHome/livingRoom/temperature

myHome/livingRoom/humidity

myHome/kitchen/temperature

myHome/kitchen/humidity
```

Already, you should be seeing how this tree structure is perfect for a smart home full of sensors and devices. The best practice for use with multiple sensors in a single room is to publish each sensor variable as it's own topic level – branching out to more specificity (as in the examples above) – rather than try to publish multiple types of sensor to the same channel.

Clients can then publish or subscribe to any number of individual topic levels, or use some special wildcard characters to filter from higher up in the tree.

The + wildcard substitutes for any one topic level. For instance:

```
myHome/+/temperature
```

would subscribe the client to both

```
myHome/livingRoom/temperature


myHome/kitchen/temperature
```

… but not the humidity levels.

The # is a multi-level wildcard, so you could fetch anything from the livingRoom sensor array with:

```
myHome/livingRoom/#
```

Technically, you can also subscribe to the root level # which you get you absolutely everything going passing through the broker, but that can be like sticking a fire hose in your face: a bit overwhelming. Try connecting to the public MQTT broker from HiveMQ and subscribing to #. I got about 300 messages in a few seconds before my client just crashed.

MQTT Beginner Tip: "**/myHome/"** is a different topic to "**myHome/"** – including a slash at the start creates a blank topic level, which while technically valid, isn't recommended because it can be confusing.

Now that we know the theory, let's have a go with an Arduino, Ethernet Shield, and a DHT11 temperature and humidity sensor – you've probably got one in your starter kit, but if not, just swap out the environmental sensor for a motion sensor(or even a button).

Publishing MQTT From an Arduino With Ethernet Connection

If you have a hybrid Arduino-compatible device with Wi-Fi or Ethernet built-in, that should also work. Eventually we'll want a better/cheaper way of communicating that having to use a network connection in every room, but this serves to learn the basics.

Start by downloading pubsubclient library from Github. If you've used the "Download as ZIP" button, the structure is a bit wrong. Unzip, rename the folder to just **pubsubclient**, then take the two files out of the **src** folder and move them up one level to the root of the downloaded folder. Then move the whole folder to your **Arduino/libraries** directory.

Here's my sample code you can adapt: the DHT11 signal output is on pin 7. Change the server IP for that of your Pi on the following line:

```
client.setServer("192.168.1.99", 1883);
```

Unfortunately, we can't use it's friendly name *(OpenHAB.local in my case)* as the TCP/IP stack on the Arduino is very simplistic and adding the code for Bonjour naming would be a lot of memory we don't want to waste. To change the topics that sensor data is being broadcast on, scroll down to these lines:

```
char buffer[10];


dtostrf(t,0, 0, buffer);


client.publish("openhab/himitsu/temperature",buffer);


dtostrf(h,0, 0, buffer);
```

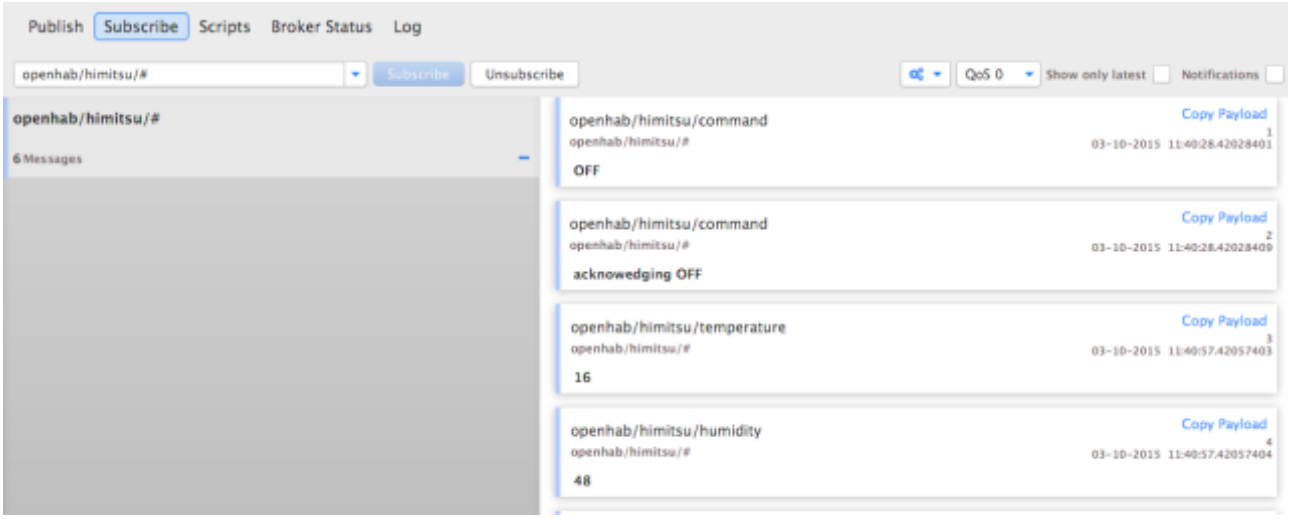
```
    client.publish("openhab/himitsu/humidity",buffer);
```

The code also includes subscription to a command channel. Find and adjust the following line:

```
    client.subscribe("openhab/himitsu/command");
```

Examine the code around there and you'll see that you could easily control an LED or relay for example by sending commands to specific channels. In the example code, it simply sends a message back acknowledging receipt of the command.

Upload your code, plug your Arduino into the network, and using MQTT.fx subscribe to either # or *openhab/himitsu/#* (or whatever you changed the room name to, but don't forget to include the # at the end). Pretty soon you should see messages coming in; and if you send ON or OFF to the command topic, you'll see acknowledgments coming back too.



MQTT Binding for OpenHAB

The final step in the equation is to hook this into OpenHAB. For that, of course we need a binding.

```
    sudo apt-get install openhab-addon-binding-mqtt

    sudo chown -hR openhab:openhab /usr/share/openhab
```

And edit the config file to enable the binding.

```
    mqtt:broker.url=tcp://localhost:1883

    mqtt:broker.clientId=openhab
```

Restart OpenHAB

```
    sudo service openhab restart
```

Then let's add an item or two:

```
/* MQTT Sensors */

Number Himitsu_Temp "Himitsu Temperature [%.1f °C]" <temperature>
(Himitsu,Temperature) {mqtt="<[broker:openhab/himitsu/
temperature:state:default]"}

Number Himitsu_Humidity "Himitsu Humidity [%.1f %%]" <water>
(Himitsu,Humidity) {mqtt="<[broker:openhab/himitsu/
humidity:state:default]"}
```

By now you should understand the format; it's getting a **Number item** from the MQTT binding, on a specified topic. This a simple example, you may wish to refer to the wiki page where it can get a lot more complex.

Congratulation, you now have the basis of a cheap Arduino-based sensor array. We'll be revisiting this in future and placing the Arduino's onto their own entirely separate RF network. I've also created an identical version for Wizwiki 7500 boards if you happen to have one of those.

Persistence and Graphing Data

By now you probably a bunch of sensors set up, whether from Z-Wave or custom Arduinos running MQTT – so you can view the current state of those sensors at any time, and you should also be to react to their value in rules. But the interesting thing about sensor values is generally that they change over time: that's where persistence and graphing comes in. **Persistence** in OpenHAB means saving the data over time. Let's go ahead and setup RRD4J (Round Robin Database for Java), so called because data is saved in a round robin fashion – older data is discarded to compress the size of the database.

Install rrd4j packages with the following commands.

```
sudo apt-get install openhab-addon-persistence-rrd4j

sudo chown -hR openhab:openhab /usr/share/openhab
```

Then create a new file called **rrd4j.persist** in the **configurations/persistence** folder. Paste in the following:

```
Strategies {

 everyMinute : "0 * * * * ?"

 everyHour   : "0 0 * * * ?"

 everyDay    : "0 0 0 * * ?"

 default = everyChange
```

```
}

Items {

 // persist everything when the value is updated, just a default,
and restore them from database on startup

 * : strategy = everyChange, restoreOnStartup



 // next we define specific strategies of everyHour for anything in
the Temperature group, and and every minute for Humidity

 Temperature* : strategy = everyHour

 Humidity* : strategy = everyMinute



 // alternatively you can add specific items here, such as

 //Bedroom_Humidity,JamesInOffice : strategy = everyMinute

}
```

In the first part of this file, we're defining strategies, which just means giving a name to a CRON expression. This is the same as we already did with My.OpenHAB, but this time we're create some new strategies that we can use of everyDay, everyHour and everyMinute. I haven't used them all yet, but I might be in future.

In the second half of the file, we tell rr4dj which data values to save. As a default, we're going to save everything each time it updates, but I've also specified some time based strategies for specific sensors. Temperatures I'm not too bothered about, so I've set that to save everyHour only, but humidity is a big concern for me, so I want to see how it's changing every minute. If there's other data you specifically want to save at set times, add those here now or adjust as needed.

Note: if you want to graph the data too, you MUST store it at least once a minute. It doesn't matter if your sensor data is even updated this quickly, you simply need to tell rr4dj to store it once a minute.

With that defined, you should begin to see some debug output telling you that values are being stored.

```
2015-09-23 11:01:59.692 [DEBUG] [.p.internal.PersistenceManager] - Scheduled strategy rrd4j.everyMinute with cron expression 0 * * * * ?
2015-09-23 11:01:59.696 [DEBUG] [.p.internal.PersistenceManager] - Scheduled strategy rrd4j.everyHour with cron expression 0 0 * * * ?
2015-09-23 11:01:59.700 [DEBUG] [.p.internal.PersistenceManager] - Scheduled strategy rrd4j.everyDay with cron expression 0 0 0 * * ?
2015-09-23 11:01:59.701 [DEBUG] [.o.m.c.i.folder.FolderObserver] - Refreshing folder 'rules'
2015-09-23 11:01:59.703 [DEBUG] [.o.m.c.i.folder.FolderObserver] - Refreshing folder 'scripts'
2015-09-23 11:01:59.704 [DEBUG] [.o.m.c.i.folder.FolderObserver] - Refreshing folder 'items'
2015-09-23 11:02:00.005 [DEBUG] [.o.m.r.i.engine.ExecuteRuleJob] - Executing scheduled rule 'Humidity Monitor'
2015-09-23 11:02:00.028 [DEBUG] [.p.rrd4j.internal.RRD4jService] - Stored 'Bedroom_Humidity' with state '58.0' in rrd4j database
2015-09-23 11:02:00.049 [DEBUG] [.p.rrd4j.internal.RRD4jService] - Stored 'Himitsu_Humidity' with state '51' in rrd4j database
2015-09-23 11:02:01.324 [DEBUG] [.io.net.http.SecureHttpContext] - security is disabled - processing aborted!
2015-09-23 11:02:01.332 [DEBUG] [o.i.r.i.resources.ItemResource] - Received HTTP PUT request at 'items/JamesInOffice/state' with value 'OFF'.
2015-09-23 11:02:02.218 [DEBUG] [inding.hue.internal.HueBinding] - Start Hue data refresh
2015-09-23 11:02:02.480 [DEBUG] [inding.hue.internal.HueBinding] - Done Hue data refresh.
2015-09-23 11:02:12.481 [DEBUG] [inding.hue.internal.HueBinding] - Start Hue data refresh
```

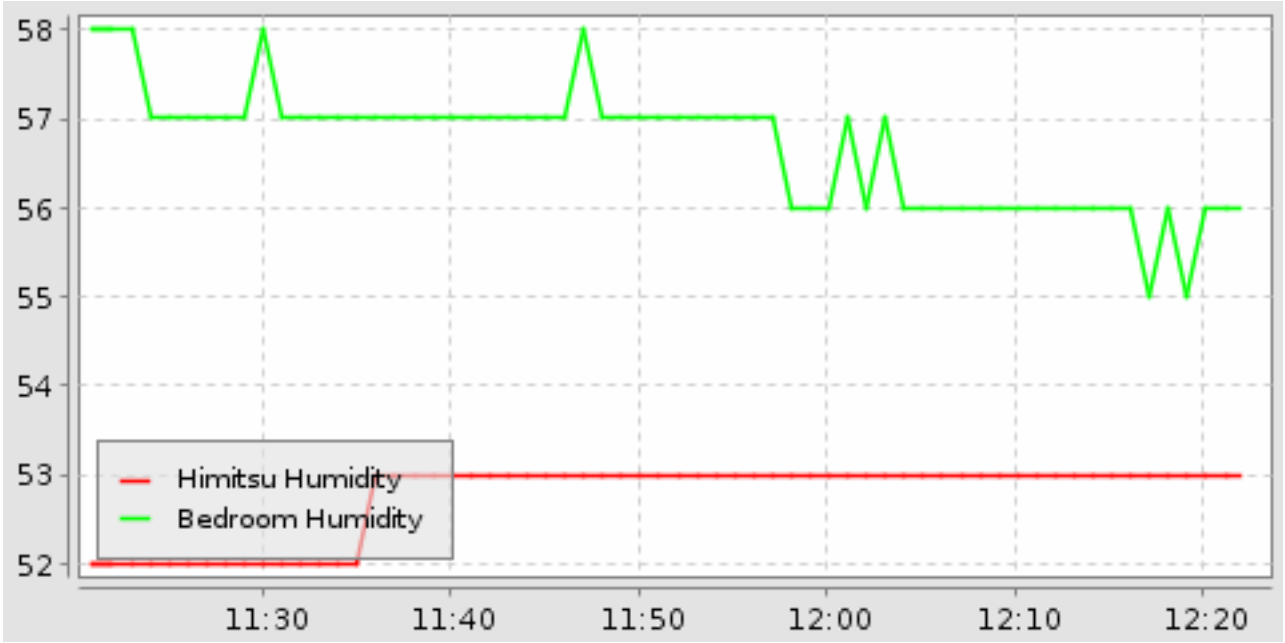
Next up, let's make some pretty graphs of all this data. It's really easy. To make a graph of an individual sensor, add the following to your site map:

```
Chart item=Bedroom_Humidity period=h
```

That's literally all you need. Valid values for period are *h, 4h, 8h, 12h, D, 3D, W, 2W, M, 2M, 4M, Y*; it should be obvious what these mean. It defaults to D for a full day of data if not specified.

To create a graph with multiple items, simply graph the group name instead:

```
Chart item=Humidity period=h
```



You might also be interested to know that you can use this graph elsewhere; it's generating an image using the following URL: *http://YOUROPENHABURL:8080/chart?groups=Humidity&period=h*

How's *Your* OpenHAB System Coming?

That's it for this installment of the guide, but don't expect this'll be last you hear from us about OpenHAB. Hopefully this and the beginner's guide have given you a solid grounding to develop your own complete OpenHAB system – but it's a process that's never really completely finished.

Thankfully, OpenHAB can scale well from a few devices to hundreds, from simple rule complexity to the ultimate in home automation – so how's your system coming along? Which devices did you choose? What's the next big project you're going to tackle?

Let's talk in the comments – and please, if you found this guide useful, click those share buttons to tell your friends how they too can setup their own OpenHAB system.