Contents

[**First Steps** 6](#_Toc523038163)

[**Installing Sensu Itself** 8](#_Toc523038164)

[Repos 8](#_Toc523038165)

[Actually Installing It 8](#_Toc523038166)

[About the package 8](#_Toc523038167)

[Configuring Sensu Server 9](#_Toc523038168)

[Installing the First Check 9](#_Toc523038169)

[Configuring a handler 9](#_Toc523038170)

[Big Chown 10](#_Toc523038171)

[Start The Server and API 10](#_Toc523038172)

[**Tinkering With Uchiwa** 10](#_Toc523038173)

[Make Some Events 10](#_Toc523038174)

[Clients 12](#_Toc523038175)

[Checks 12](#_Toc523038176)

[Stashes 13](#_Toc523038177)

[Aggregates 13](#_Toc523038178)

[Datacenter 13](#_Toc523038179)

[**Conclusion** 14](#_Toc523038180)

[**Installing a Real Handler** 14](#_Toc523038181)

[**Intro** 14](#_Toc523038182)

[**Exploring Community Handlers** 14](#_Toc523038183)

[**Installation** 15](#_Toc523038184)

[**Adding a Real Check** 18](#_Toc523038185)

[Intro 18](#_Toc523038186)

[Going for a Nagios Plugin 19](#_Toc523038187)

[Configuration 19](#_Toc523038188)

[Installing Community Plugin 20](#_Toc523038189)

[Further Reading 21](#_Toc523038190)

[**Install sensu with chef** 21](#_Toc523038191)

[Installing the required cookbooks 21](#_Toc523038192)

[Making our sensu\_server wrapper cookbook 22](#_Toc523038193)

[**Installing a Client** 23](#_Toc523038194)

[**Installing Uchiwa** 23](#_Toc523038195)

[Installing a Disk Check 23](#_Toc523038196)

[Installing a Email Handler 24](#_Toc523038197)

[**Conclusion** 25](#_Toc523038198)

[**How to Tune Sensu Check Parameters** 25](#_Toc523038199)

[**Anatomy of a Sensu Check Definition** 25](#_Toc523038200)

[How Checks Are Defined 26](#_Toc523038201)

[Sensu-plugin Attributes 26](#_Toc523038202)

[Custom check attributes 27](#_Toc523038203)

[**Conclusion** 28](#_Toc523038204)

[**Writing Your Own Sensu Checks** 29](#_Toc523038205)

[**Your First Check Script** 29](#_Toc523038206)

[**Porting to Ruby** 30](#_Toc523038207)

[Expanding the check with negation 32](#_Toc523038208)

[Conclusion 34](#_Toc523038209)

[Further Reading: 34](#_Toc523038210)

[Final version of the check\_file\_exists 34](#_Toc523038211)

[**Writing Your Own Sensu Handlers** 35](#_Toc523038212)

[What Language Handlers are Written In 35](#_Toc523038213)

[Getting Started 35](#_Toc523038214)

[Writing a Real Handler 36](#_Toc523038215)

[Making the Handler More Sane With Settings 36](#_Toc523038216)

[More Adjustments 37](#_Toc523038217)

[Conclusion 37](#_Toc523038218)

[Further Reading 38](#_Toc523038219)

[**Taking Advantage of the Embedded Omnibus Ruby** 38](#_Toc523038220)

[Intro 38](#_Toc523038221)

[Advantages 38](#_Toc523038222)

[Method 1: EMBEDDED\_RUBY=true 38](#_Toc523038223)

[Method 2: Hard-coding Shebangs (#!) 39](#_Toc523038224)

[Looking at the ruby 39](#_Toc523038225)

[Using the Ruby 39](#_Toc523038226)

[Conclusion 40](#_Toc523038227)

[**Standalone, Safemode, and Subscriptions** 40](#_Toc523038228)

[Sensu's Scheduling Model 40](#_Toc523038229)

[Subscription Checks 40](#_Toc523038230)

[Security Implications / safe\_mode 41](#_Toc523038231)

[Standalone Checks 41](#_Toc523038232)

[Aggregate Notes. 42](#_Toc523038233)

[Conclusion 42](#_Toc523038234)

[**Handlers, Filters, and Subdued Checks** 43](#_Toc523038235)

[How Filters Work 43](#_Toc523038236)

[Custom Client Attributes 43](#_Toc523038237)

[Subdue 45](#_Toc523038238)

[Conclusion 45](#_Toc523038239)

[**Securing RabbitMQ With SSL** 45](#_Toc523038240)

[Making a CA, Signing Some Certs 45](#_Toc523038241)

[Adding Certs to RabbitMQ 46](#_Toc523038242)

[Adding Certs to Sensu 46](#_Toc523038243)

[Alternatives 47](#_Toc523038244)

[**Hardening the Rest of Sensu** 48](#_Toc523038245)

[RabbitMQ 48](#_Toc523038246)

[Redis 48](#_Toc523038247)

[Sensu API 49](#_Toc523038248)

[Sensu Client 49](#_Toc523038249)

[Sensu Server 49](#_Toc523038250)

[Dashboards 49](#_Toc523038251)

[Conclusion 49](#_Toc523038252)

[**Installing and Using the Sensu-cli** 50](#_Toc523038253)

[Intro 50](#_Toc523038254)

[Installation 50](#_Toc523038255)

[Background Prep 50](#_Toc523038256)

[Usage 50](#_Toc523038257)

[Doing more interesting things 51](#_Toc523038258)

[Having a Server Silence Itself 52](#_Toc523038259)

[Silencing Clients 53](#_Toc523038260)

[Emitting Alerts 53](#_Toc523038261)

[Resolving Alerts 53](#_Toc523038262)

[Conclusion 54](#_Toc523038263)

[All Examples 54](#_Toc523038264)

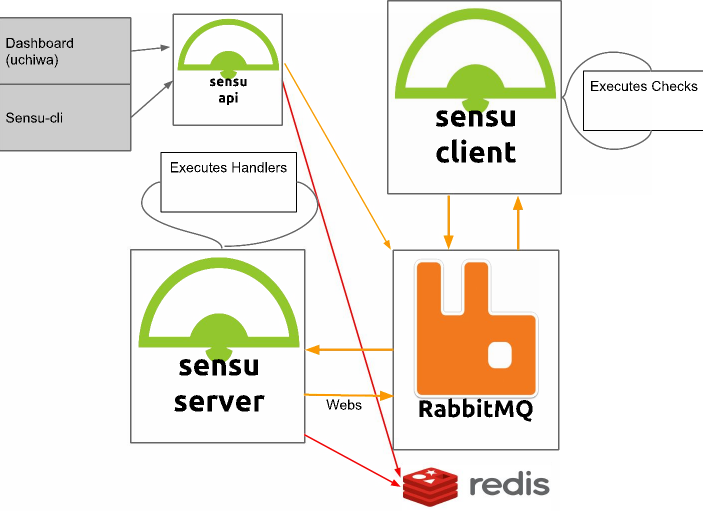
[Have a host silence itself 54](#_Toc523038265)

[Silence any client that has the word "test" in the name 54](#_Toc523038266)

[Delete sliences older than 3 days 54](#_Toc523038267)

[Purge any checks that haven't checked in in a month 54](#_Toc523038268)

# **First Steps**

Before we install anything or make any config files, we have to start at some core concepts and talk about Sensu's architecture. Most of the misconceptions I see people have on the Sensu mailing list are a result of assumptions that users make about Sensu that are incorrect, because they assume that Sensu is like Nagios or other types of monitoring systems. It isn't. Lets talk about why that is. 

**The Sensu client**

Let's start at the Sensu Client. The Sensu Client is a daemon or agent that runs on every server that you want to monitor. It is written in Ruby, and can run on Linux, BSD, or Windows.

The Sensu Client the process that *actually* executes checks. Keep that in mind this is the only place where actual check scripts run. Now, the Sensu Client can also *schedule* its own checks. These are called "standalone checks", because the client isn't dependent on getting instructions from the Sensu Server on what and when to run.

Now, the Sensu Client can't be that standalone, it has to do something with the results of the checks. For that, the Sensu Client puts its results onto the transport, which is RabbitMQ.

**RabbitMQ**

Here is where you might be thinking, Kyle, I'm hear to learn about Sensu, what does RabbitMQ have to do with it? Sensu uses a Queue to separate the actions between the clients and the server. Its a good thing, and the fact that Sensu doesn't reinvent the wheel here. It is true that this external component makes the overall architecture more complex, but it buys flexibilty that hopefully you will see exposed later.

But, back to RabbitMQ: the Sensu Client places result data onto RabbitMQ. Good.

**Sensu Server**

The Sensu Server is the daemon that pulls result data off of the queue and acts on that data. That action is usually in the form of executing *handlers*. Handlers are programs that read in the event data from stdin, and do something like email you or talk to pagerduty.

The Sensu server can *also* ask for checks to be executed on the Sensu clients. These are called subscription-based checks, because the clients have to "subscribe" to a particular tag, like "webserver". The Sensu server puts a request for a particular check, say check\_http to be executed by all clients that are subscribed to the "webserver" tag. Remember this is contrast to "standalone" checks where the Sensu-client does the scheduling.

The differences between subscription-based checks and standalone checks is also a common source of confusion amongst new Sensu users. I'll cover this in-depth in a different course.

**Redis**

Sensu has to keep state somewhere. The state includes things like check history, timestamps, what’s up, what’s down, etc. Sensu is designed to store that state in Redis. Redis, if you haven't heard of it, is a key-value store. The disadvantage to you as a user is that you must run a Redis instance. The advantage is that the Sensu-server itself is stateless, and easier to scale, and it doesn't have to re-invent the wheel for storing state, which makes the Sensu codebase smaller.

**Sensu API**

While Sensu-server does it's one thing (operate on events and execute handlers), and Redis does its state-storing thing, there is also the Sensu-API process. Its job is to just talk to Redis and RabbitMQ and provide a rest interface to what is going on. Handlers themselves often use this API, as well as dashboards and external tools.

**Dashboard (Uchiwa)**

With any monitoring system, it is nice to have a dashboard. Would you believe that Sensu doesn't have a built in dashboard? It might be surprising if you are coming from a monolithic monitoring system where all this stuff is contained in a single process. But in Sensu it is separate. The canonical Sensu dashboard is Uchiwa. Uchiwa is written in Go, and provides a nice "plane of glass" to expose what is going on with your systems. It only needs to connect to the Sensu API. We'll be investigating this component in a later lecture.

# **Installing Sensu Itself**

First install erlang, rabbitmq and redis.

### Repos

Like many projects, Sensu provides packages and repos to make it easy to install.

Here we'll install the Debian package repos per the docs:

wget -q http://repos.sensuapp.org/apt/pubkey.gpg -O- | sudo apt-key add -

echo "deb http://repos.sensuapp.org/apt sensu main" | sudo tee /etc/apt/sources.list.d/sensu.list

apt-get update

### Actually Installing It

Now that we have the repo installed and up to date, we can actually install the Sensu package.

apt-get install sensu

### About the package

As an engineer, you need to know that the Sensu package itself is in "omnibus" form, which means it contains everything it needs to run, including its own ruby interpreter. The same package can function as a Sensu Client, or Server, or API.

Lets look a bit at the package contents:

dpkg -L sensu | less

You can see some example config files, a README in that conf dir, lets check that out next.

Lots of stuff in /opt/sensu/embedded, that is the embedded ruby and gems. It does make the package bigger, but is sure makes easier to deploy, especially considering all the strange and old system rubys that exist everywhere. I promise, you will learn to love the omnibus package.

Lets take a look at that readme:

vim /etc/sensu/conf.d/README.md

You will eventually pick up here that Sensu's configuration is JSON in /etc/sensu/conf.d.

True story, Sensu reads in all the json in these directories recursively and then does a big merge. So the filenames are actually irrelevant. The JSON format should be a hint to you that it is designed for a computer program to spit out these files, aka a configuration management tool. While out of the scope of this introductory video, using configuration management is highly recommended, it is pretty easy for us humans to make mistakes writing out json.

### Configuring Sensu Server

I'm going to use the same instructions from the official documentation for this introduction, and talk a little bit about what is going on behind the scenes.

Let’s download the starter sensu-server config file:

wget -O /etc/sensu/config.json http://sensuapp.org/docs/0.20/files/config.json

vim /etc/sensu/config.json

You can see for this example we are running all of these components on localhost on the same server. That is fine for this example. You can see some of Sensu's flexibility here, as you can scale out the components by scaling each in their own way, splitting out things to different hosts, etc. But for most installations, having everything on one host is just fine.

### Installing the First Check

We are going to go ahead and follow the official docs and install a memory check:

wget -O /etc/sensu/conf.d/check\_memory.json http://sensuapp.org/docs/0.20/files/check\_memory.json

vim /etc/sensu/conf.d/check\_memory.json

As hinted in the early architecture lecture, this is a "subscription" check, as you can tell because there are subscribers in that array, and there is no "standalone" key. (on standalone checks there would be a standalone: true) So this will be a check that will be executed by the client, but requested by the server. It just so happens the client and server on the same machine here.

### Configuring a handler

If you remember from the first lecture, Handlers are the things that the sensu Server runs in response to Events. They are just scripts that read event data in json form from stdin and then do something. In this example we are just going to install a very trivial handler, "cat":

wget -O /etc/sensu/conf.d/default\_handler.json http://sensuapp.org/docs/0.20/files/default\_handler.json

vim /etc/sensu/conf.d/default\_handler.json

Can you imagine in your head what this handler will do when it gets event data? Thats right, it will just, spit the same thing back out again? Where will it spit it out? Well, not pagerduty or anything, but just in the Sensu-server log. Remember that the Sensu Server is responsible for executing handlers.

We'll see it in action in a bit.

### Big Chown

sudo chown -R sensu:sensu /etc/sensu

Just in case we made a mistake as root and sensu can't read any of these files.

### Start The Server and API

Lets start up the sensu server and the api component and check it out:

sudo /etc/init.d/sensu-server start

sudo /etc/init.d/sensu-api start

You can see how its doing:

tail -f /var/log/sensu/sensu-server.log

tail -f /var/log/sensu/sensu-api.log

The Sensu logs can be a little tricky to parse, as they themselves are in json format. Sometimes I like to use the jq command to make it a little prettier:

tail -f /var/log/sensu/sensu-server.log | jq .

jq deserves a whole course in itself. Maybe I'll make a jq course... But for now jq . is good printing jq command to know.

# **Tinkering With Uchiwa**

Welcome to "Tinkering with Uchiwa". If you are installing Sensu for the first time, then this dashboard will be new to you, so I want to show you around a bit.

As a reminder, Uchiwa is the open source dashboard. There is the enterprise dashboard too, but it operates very much in the same way.

### Make Some Events

wget https://raw.githubusercontent.com/solarkennedy/sensu-shell-helper/master/sensu-shell-helper

chmod +x sensu-shell-helper

./sensu-shell-helper -n "check\_http" /bin/false

./sensu-shell-helper -n "check\_ssh" /bin/false

./sensu-shell-helper -n "check\_ssh" -j '"source": "some-other-host1",' -- /bin/false

./sensu-shell-helper -n "check\_snmp" -j '"source": "some-other-host2",' -- /bin/false

Do you notice anything in particular about this first view? It is a little tricky, how about what you don't notice?

This might surprise you, and this is another very common misconception about Sensu that people have, especially coming from other monitoring systems. Sensu is not a "green light" dashboard.

Green-light dashboards don't scale very well. If you have 1000 sensu clients each with 1000 checks that are all green, that is 1 million events that sensu and the dashboard would have to process.

Sensu's event-driven architecture means that it only shows things that are red. This is a good thing, hopefully you can just get used to it.

While it is true that sensu can record the event output of green checks, by default it does not: You only see the things that are failing.

#### More Events Page

Lets talk about the columns on this page.

Why the "Source" column? Why not "hostname"? Although this is and advanced topic, Sensu can get alerts for hosts that don't really exist. This is for things like switches or clusters or things like that. I've made some fake hosts here for demonstration purposes.

Check is the check name. Output would be the last outout of the check, for these injected events I didn't add any output data, but this would be where it would say "Critical, only 100M free mem" or whatever"

This "Occurences" column represents how many times this event has fired. It doesn't count how many times it has been green.

This cloudy thing is "datacenter", remember that uchiwa references each different Sensu cluster as a datacenter, so this would allow you to easily sort by datacenter if you wanted to. And this "issued" column, this represents how long ago the event came in.

Lets click on an event to get more data.

#### An Event

You can see in this modal we have some more data about the host that made it and the event output. This is mostly informational, lets talk about the buttons you can click on.

These speaker icons are for silencing the host over here, or the check over here. If you silence a host, most Sensu handlers will also suppress notifications for other checks on that host as well. You can also silence the check as well. If you silence something, it won't make alerts, as you might expect.

One subtle thing to note here is that the handlers that sensu executs have to be "silence-aware", our simple cat handler is going to continue to operate and doesn't know anything about silences.

How about this X? What does it mean to "delete" a client? I'll talk about that in a bit on the clients page. How about this Checkmark? You can click this checkmark to manually resolve the check, as if it was just fine. Granted, if the thing that spawned the alert alerts again, it will come right back of course. But remember that in Sensu everything is event driven. Maybe this was a one-time event? Maybe there is no such thing as check\_snmp anymore and it will never resolve by itself? It happens sometimes that you have to manually resolve things.

### Clients

You can see the client here. Sensu does track the status of the clients.

You can Silence a client if you don't want it to make any noise from the handlers by clicking this volume icon. Again, this willsilence all the checks and the client itself. Which is what most people see as the expected behavior.

If you click on a client, you can see more data. This is very similar to the check modal, except here sensu shows you all of the checks it knows about.

They are green, but this listing is not necissarily exhaustive or fresh. It just represents checks that it has a history for, and if the last return code for the history of a check is 0, then it shows up as green here.

#### Delete a client

Lets go ahead and delete this client. What happened? I thought I deleted it? Well deleting it did auto-resolve any of the failing checks it had before. But the sensu client running our our host auto-re-registered itself within a few seconds! Remmeber Sensu does not have some big master list of hosts and checks that belong to them. It operates on events, and this host just decided to register itself now.

Hopefully this is getting the gears turning for you on how useful this would be in a dynamic environment. Remember all of this is happening through a an api. And there are also good command line tools. Just think about what you can build when you work with a system that doesn't have to be "told" that a host exists, and it can just come into existance!

But you saw that me deleting the client didn't make it go away. Remember my metaphor about the map and the territory? Sensu is always in a constant state of reconstructing its map to match the territory.

### Checks

Checks will show only server-side checks. The Sensu-API is not aware of standalone checks that are only defined on the clients. This can also be confusing to newcomers, because they expect to see every check ever here.

But, if you are running a Sensu-client on the same host as the Sensu Server, then you will see those checks here.

This screen is mostly read only, to give you an idea of the Subscription based checks and who subscribes to them.

### Stashes

Stashes are Sensu's kinda free-form key-value store. Sensu handlers use the slience stashes to mark a host or check as silenced. But they can be anything! This makes Sensu very flexible, but taking advantage of this feature is a little outside the scope of this video.

The important thing to know is that Silences are implemented by making a stash here. And in this view you can see all the stashes.

Lets silence a few things and see how they show up here.

Stashes are just keys and values. You can use them for anything, but silencing is the most popular way to use them. Plus remember they have an api. What if you have a provisioning system that launches new hosts and you want them silenced for a few hours? Can you imaging making that api call or using the cli tool in a script? With the reason field being "The host was newly provisioned" and the "source" was "your provisioning tool"? Could be pretty cool.

### Aggregates

Aggregates are an advanced topic, they have to do with executing a check over the course of many hosts and doing a tally or aggregate of the results and acting on it.

I'm not going to talk about it in this introductory course, but I'll link to it in the external resources section of this lecture.

### Datacenter

Here is where we see that multi-site capability show up. We only defined one site, and it shows up here. If you were building a multi-site setup, the health of the individual endpoints would show up here.

## **Conclusion**

That is about it for the dashboard. Uchiwa is a modern dashboard. It is pretty responsive and gets the job done.

This is just an introductory course, but you can do a lot more. Uchiwa will actually automatically embed links and images, which can give you a really rich experience if you want to embed graphs from an external datasource.

And of course, remember the API. The dashboard is just one way to visualize events. On my github I have a script that interacts with the API and prints a list of events in the message of the day, so you see it when you log in.

# **Installing a Real Handler**

## **Intro**

If you remember from the architecture lecture, the job of the Sensu Server is to respond to event data and act on it. The actions are usually in the form of "Handlers".

A Sensu "Handler" is a script that does something based on the event data that comes from standard in. This makes them pretty easy to write, test, and understand.

Remember our "cat" handler?

cat /etc/sensu/conf.d/handlers.json

It is pretty naive, and just spits out the event data right back at us. You can see it in the logs:

tail -f /var/log/sensu/sensu-server.log

Handler output goes in the sensu-server log, because that is the thing that is executing handlers.

Handlers docs: <https://docs.sensu.io/sensu-core/1.4/reference/handlers/>

## **Exploring Community Handlers**

Lets go find a handler to send us some email. Sending email is probably the most popular thing to do with monitoring systems. Where should we go?

Well you should be aware of this sensu-plugins github project:

xdg-open https://github.com/sensu-plugins/

This github project hosts lots and lots of Sensu Plugins. A Sensu plugin is just a general term for anything that Sensu runs. Handlers are one kind of Sensu plugin, but there also checks and other things.

Well, lets type in the search box for mail...

sensu-plugins-mailer sounds legit. sensu-plugins-qmail is probably plugins for checking qmail, not sending email. Sensu-plugins-ponymailer is a more advanced email handler using the "pony" rubygem. Lets start with the first one: sensu-plugins-mailer

Lets see.. some require configuration, some example json. So it Sends some email, probably what we want. Now what?

## **Installation**

This is a ruby gem, so to install it you use the gem command. These are just general docs, nothing mailer specific. If you already know how to handle rubygems, you may be right at home. Not everybody does though. Does this mean you have to install ruby to send email?

Well, remember earlier on when I talked about how Sensu was packaged? Remember that it is an omnibus package, meaning it comes with its own version or ruby with it!

This is pretty convenient. We don't have to install another ruby, who knows what version of ruby it would be or whatever. If we can just use Sensu's ruby, lets do that. Where is it again?

ls /opt/sensu/embedded/bin/

The gem command is right there. Will it work?

/opt/sensu/embedded/bin/gem install

sudo /opt/sensu/embedded/bin/gem install sensu-plugins-mailer

Ok still didn't work. Missing gcc?

sudo apt-get install build-essential

sudo /opt/sensu/embedded/bin/gem install sensu-plugins-mailer

But where did it put it?

/opt/sensu/embedded/bin/gem contents sensu-plugins-mailer

Ok, so it is installed in the gems folder in the embedded ruby stuff. Ok. But what about the binaries?

ls /opt/sensu/embedded/bin/ | grep mailer

And checkout the shebang:

head /opt/sensu/embedded/bin/handler-mailer.rb

This is important because we don't want this handler using the system ruby, if we even have any. Does it work?

/opt/sensu/embedded/bin/handler-mailer.rb

Is it working? No news is good news? Watch I'll press cntrl-d.

error reading event: A JSON text must at least contain two octets!

See, this is what handlers do, they wait for event data in json form to come in to stdin. So it is working. Lets configure it.

cd /etc/sensu/conf.d

sudo vim default\_handler.json

You can see the leftover cat handler here. Lets just replace the path.

Now what about configuration for the handler? Lets check back at the docs:

{

"mailer": {

"admin\_gui": "http://admin.example.com:8080/",

"mail\_from": "sensu@example.com",

"mail\_to": "monitor@example.com",

"smtp\_address": "smtp.example.org",

"smtp\_port": "25",

"smtp\_domain": "example.org"

}

}

It doesn't say what file this goes in right? Well Sensu just does a big merge of the json dictionaries in this thing, so it actually doesn't matter what file it goes in. I like to have the handler and config in close by, so lets just stick it in the same file

"mailer": {

"admin\_gui": "http://localhost:3000/",

"mail\_from": "sensu@localhost",

"mail\_to": "root",

"smtp\_address": "localhost",

"smtp\_port": "25",

"smtp\_domain": "localhost"

}

So we changed a config file. Do we need to do anything else? Yes, we do need to restart the sensu-server. Remember that handlers are executed and configured by the sensu server, so in this case it is the only thing that needs to be restarted:

sudo /etc/init.d/sensu-server restart

Now lets watch it:

tail -f /var/log/sensu/sensu-server.log

Is it working? Try to think back about the uchiwa lecture and think to yourself, why don't I see anything about email? Of course, Sensu is not a green light dashboard. And it is anything actually failing? Cause if there isn't anything failing, no events are going to get processed and no handlers are going to fire. This seems a little obvious in retrospect. Lets make our check fail.

sudo vim check\_memory.json

Now, do I restart the sensu server or the sensu client? Or what?

Well this particular check is a subscription check, which is defined and published by the server, so you should restart the sensu server

sudo /etc/init.d/sensu-server restart

Lets watch how the client reacts:

tail -f /var/log/sensu/sensu-client.log

{"timestamp":"2015-08-15T22:36:00.230324+0000","level":"info","message":"received check request","check":{"name":"memory","issued":1439678160,"command":"exit 2"}}

{"timestamp":"2015-08-15T22:36:00.238223+0000","level":"info","message":"publishing check result","payload":{"client":"mycoolhost","check":{"name":"memory","issued":1439678160,"command":"/etc/sensu/plugins/check-mem.sh -w 128 -c 64","interval":10,"subscribers":["test"],"executed":1439678160,"duration":0.008,"output":"MEM OK - free system memory: 296 MB\n","status":0}}}

Why is this happening? The server is telling it to run a particular command, but the client uses its local version of check-mem. Whats up wit that?

This behavior is surprising, but an artifact of the fact that we have the sensu server and sensu client on the same server. If the Sensu client sees a check definition, it will assume that the check definition contains special overrides that are designed to take precedence over the generic check-mem that comes from the server. For example, you can imaging you have a fleet of servers that have different memory capacities, and on a special server with lots of ram you set a different threshold for how much ram is critical. That is what is happening to our example here. The solution is to restart the sensu client.

sudo /etc/init.d/sensu-client restart

tail -f /var/log/sensu/sensu-client.log

This is one of the reasons why I personally prefer client-defined checks only. I trade the centralized control in favor of having each client responsible for their own checks. This is easier to do with configuration management, like Puppet or Chef.

Anyway, is it working now? Are we getting emails?

tail -f /var/log/sensu/sensu-server.log

Its doing "something" Lets look at the logs really carefully:

{"timestamp":"2015-08-15T22:43:20.244902+0000","level":"info","message":"processing event","event":{"id":"185a9dbe-c132-4aae-9cef-4733c988d30b","client":{"name":"mycoolhost","address":"localhost","subscriptions":["test"],"version":"0.20.2","timestamp":1439678580},"check":{"command":"exit 2","interval":10,"subscribers":["test"],"name":"memory","issued":1439678600,"executed":1439678600,"duration":0.004,"output":"","status":2,"history":["0","0","2","2","2","2","2","2","2","2","2","2","2","2","2","2","2","2","2","2","2"],"total\_state\_change":4},"occurrences":19,"action":"create","timestamp":1439678600}}

{"timestamp":"2015-08-15T22:43:20.760092+0000","level":"info","message":"pruning check result aggregations"}

{"timestamp":"2015-08-15T22:43:20.908157+0000","level":"info","message":"handler output","handler":{"type":"pipe","command":"/opt/sensu/embedded/bin/handler-mailer.rb","name":"default"},"output":["only handling every 180 occurrences: mycoolhost/memory\n"]}

It is getting an event. But the handler is doing this, "only handling..." What does this mean? Well occurrences is how many times the event has occurred. You can see how many occurrences we have had here. (19). And by default this handler only does something every 180 occurrences. This is a filter mechanism so you don't get an email every 30 seconds.

There are lots of things you can do to tune this filter, but they are out of scope for this introductory course. For now you should just be aware of them and know that they will suppress handler activity.

This check was already failing 19 times before we event restarted the server, so we are going to have to wait before this handler is activated...

You know what we can do though. We can use Uchiwa to manually resolve this check which will reset the occurrences back to zero, as if it was fresh.

And now we made it do something. A little messy here, but connection refused? Thats because on this vagrant box I don't have a mail server running to use. Let me install one real quick.

sudo apt-get install postfix mailutils

And now a manual resolve....

And now the handler says it did something:

"output":["mail -- sent alert for mycoolhost/memory to root\n"]}

Lets read it:

sudo mail

It works! We got an email from that manual resolve and from the alert.

Obviously we have just scratched the surface here, but at least you have a Sensu installation that can email you, which is not bad.

There are many more things you can do here, there are tons of community handlers available out there, and of course you can write your own. Remember that handlers are just scripts that take in the event data from stdin and then do something.

Check the external resources section of this lecture for show notes and links to the documentation on handlers, as well as all the commands I used in this lecture.

# **Adding a Real Check**

### Intro

So far we have been using just been using the toy check-mem from the tutorial on the official documentation. Let's see if we can do something more realistic and useful.

### Going for a Nagios Plugin

You know, not everybody has the luxury of setting up a greenfield monitoring system. Let's say that you are migrating from Nagios and you want to re-use your existing check definitions and plugins.

I think this is great. I love re-using components. Let's get them

sudo apt-get -y install nagios-plugins-basic

Now we have those familiar nagios checks:

dpkg -L nagios-plugins-basic | grep check\_

Let's use the good ole-fashion check\_disk.

/usr/lib/nagios/plugins/check\_disk /

Just for fun I'm going to make a "ballast" file that we can delete to clear the alert.

dd if=/dev/zero of=ballast bs=1024 count=1024000

Let's crank it till it fails

/usr/lib/nagios/plugins/check\_disk /

Now let's look at how we might use thing and configure it.

### Configuration

cd /etc/sensu/conf.d/

We can peak at the other example check definition, or we can make our own. We can decide if this is going to be a standalone check or a subscription check. Remember that standalone checks are defined and scheduled by the client. Subscription checks are defined and scheduled on the server.

I like standalone checks myself. And disks are one of those things that are intrinsic to the server, sorta. So let's pretend that we are going to deploy these checks with configuration management, a provisioning tool, or something like that.

sudo vim check\_slash.json

Sensu configuration files are JSON, and they are dictionaries of whatever the thing is you are defining. This is a check so it looks like this:

{

"checks": {}

}

Now inside this dictionary we can define our check:

{

"checks": {

"check\_slash": {

"command": "/usr/lib/nagios/plugins/check\_disk -c 50% /",

"interval": 10,

"standalone": true,

}

}

}

Not that bad. The check\_disk in command we came up with kinda experimentally but in real life you might copy+paste from an existing setup. Sensu checks are always executed by the client, so there is no need for an extra remote execution program. The Sensu user has permission to run this command, so we don't need sudo.

Interval is how many seconds between checks. So this check will run once per minute. With Sensu, because all the checks are executed by the client, it is relativly cheap to have quick execution times. And standalone: true because this is a check scheduled and defined by the client, the Sensu server doesn't know about it.

So we save the file, let's verify it is valid syntactically:

jq . check\_slash.json

Now we only need to restart the sensu client to pick up on it, because it is a standalone check.

sudo /etc/init.d/sensu-client restart

tail -f /var/log/sensu-client.log

Seems to be working, our sensu-client is executing this check every 10 seconds. It is currently failing. Check our mail?

sudo mail

Now lets clear our ballast file to free up some disk space.

rm ballast

Now check our mail?

sudo mail

### Installing Community Plugin

Now let's say that you were not content with the old fashion plugins and you would like to try something new out of the community plugins repo. You see something that catches you eye and you say, "oh I really want some rabbitmq checks", so you go look for them, and you find some good ones!

Great! By all means take advantage of the existing work that has been open-sourced for you to use! Now remember how to install this using the embedded ruby:

sudo /opt/sensu/embedded/bin/gem install sensu-plugins-rabbitmq

Now where did it put the scripts:

ls /opt/sensu/embedded/bin/ | grep rabbit

Does it work?

/opt/sensu/embedded/bin/check-rabbitmq-alive.rb

Amazing. As long as you use the full path to this thing, you don't have to mess with any environment variables or anything, the embedded ruby is self-contained and the script already has the full path to the correct ruby interpreter in the she-bang line like we saw before.

Lets setup the check:

{

"checks": {

"check\_rabbitmq\_alive": {

"command": "/opt/sensu/embedded/bin/check-rabbitmq-alive.rb",

"interval": 10,

"standalone": true,

}

}

}

And restart the sensu client

sudo /etc/init.d/sensu-client

Now what will actually happen when I stop rabbitmq?

sudo /etc/init.d/rabbitmq stop

tail -f /var/log/sensu/sensu-client.log

The sensu client cannot talk to RabbitMQ, and therefore can't report that RabbitMQ is down. This is kind a strange situation. In later courses I'll discuss how to monitor Sensu itself as a whole, but with RabbitMQ down, the Sensu client cannot operate.

sudo /etc/init.d/rabbitmq start

tail -f /var/log/sensu/sensu-client.log

And you can see that the Sensu client automatically reconnects and begins operating again, but the Sensu server never got a message that RabbitMQ was down.

But what notifications did we get while RabbitMQ was stopped?

sudo mail

The Sensu client periodically sends keepalive heartbeats via RabbitMQ for the Sensu server to pick up on. When RabbitMQ was down, the Sensu-client was not able to send its heartbeats, and the Sensu-server picked up on that, and alerted us. It does not mean that the Sensu-server couldn't talk to the client, at least not directly.

Anyway, that was just a small diversion for installing checks. We covered using existing nagios plugins and modern Sensu Community Plugins from Github. Feel free to review the external resources of this lecture to see the official documentation on Sensu check definitions, as well as all the commands I typed in this lecture.

### Further Reading

* <https://sensuapp.org/docs/0.20/checks>

# **Install sensu with chef**

### Installing the required cookbooks

Let's take a look at the cookbooks we are going to use. The major one is of course, [Sensu](https://github.com/sensu/sensu-chef).

For the purposes of this lecture I'm actually going to skip SSL configuration.

Looking through the provided recipes, it looks like we are going to need to include sensu::default first. And then for configuring a server we are going to need rabbitmq, redis, server\_service, and api\_service. We'll install a client and a dashboard a bit later.

I'm hoping to get away with most of the sane default here, but these attributes will become important to you as you actually integrate Sensu with your existing chef infrastructure.

Let's get the cookbook and its dependencies so we can use it using the knife tool:

knife cookbook site install sensu

...

ls

Look at all of these cookbooks we didn't have to write!

### Making our sensu\_server wrapper cookbook

But we will need to write at least one "wrapper" cookbook to kinda put things together for us. I'm going to call this first wrapper cookbook, sensu\_server:

knife cookbook create sensu\_server

Now let's begin to put together our first sensu\_server recipe and try to put all these pieces together. First let's include that default recipe the documentation said we need to include:

include\_recipe "sensu::default"

Now lets try to include all the other components that normally go on the Sensu server:

include\_recipe "sensu::rabbitmq"

include\_recipe "sensu::redis"

include\_recipe "sensu::api\_service"

include\_recipe "sensu::server\_service"

Now let's use chef-solo and see what happens. I'm going going to make a role or anything, I'll just run this cookbook directly

chef-solo -o sensu\_server --config solo.rb

What do we get? Our first error:

Chef::Log.debug 'apt is not installed. Apt-specific resources will not be executed.' unless apt\_installed?

Kinda strange. Of course apt is installed, but this is chef telling us that we have to let it know that we need apt. But really we only need to include the sensu cookbook as a dependency of our wrapper:

vim cookbooks/sensu\_server/metadata.rb

depends 'sensu'

chef-solo -o sensu\_server --config solo.rb

Our next error is about databags:

40>> raw\_hash = Chef::DataBagItem.load(data\_bag\_name, item)

By default the sensu cookbook uses databags to share ssl certs between clients and servers. For the purposes of this lecture I'm not going to go through this particular procedure, so I'm just going to set the attribute to disable ssl for now, just like we were actually doing in all the previous examples.

vim cookbooks/sensu\_server/attributes/default.rb

default["sensu"]["use\_ssl"] = false

chef-solo -o sensu\_server --config solo.rb

Wow, lots of stuff. But now you can see the sensu things are running:

ps -ef --forest

## **Installing a Client**

A lone Sensu server is no fun, let's also make sure there is a client ready to do stuff:

vim cookbooks/sensu\_server/recipes/default.rb

sensu\_client 'localhost' do

address '127.0.0.1'

subscriptions []

end

include\_recipe "sensu::client\_service"

Now we can apply it

chef-solo -o sensu\_server --config solo.rb

And you can see it is running, although we haven't configured any checks.

## **Installing Uchiwa**

Let's install a dashboard so we can visually see what is going on with Sensu. To do that we'll need the [Uchiwa cookbook](https://github.com/sensu/uchiwa-chef):

knife cookbook site install uchiwa

Now we can add this to our recipe:

vim cookbooks/sensu\_server/recipes/default.rb

include\_recipe "uchiwa"

Now we will need to make sure our sensu\_server wrapper depends on it:

vim cookbooks/sensu\_server/metadata.rb

depends 'uchiwa'

chef-solo -o sensu\_server --config solo.rb

Opening it in a browser:

xdg-open [http://localhost:3000](http://localhost:3000/)

We have a login page because the default chef attributes setup a username of admin and a password of supersecret

https://github.com/sensu/uchiwa-chef/blob/master/attributes/default.rb

### Installing a Disk Check

Just like in the introductory course, I would like to install a disk check from from Sensu plugins github project. We are going to do the same thing as before were we install it using the Sensu omnibus ruby.

With Chef we can use the sensu\_gem provider to get it installed. Remember that only the Sensu client executes checks, so this code should be associated with the chef recipe that does client stuff.

vim cookbooks/sensu\_server/recipes/default.rb

sensu\_gem 'sensu-plugins-disk-checks'

With the check gem installed, we can configure the check:

sensu\_check 'check-disk' do

command "/opt/sensu/embedded/bin/check-disk-usage.rb"

standalone true

end

chef-solo -o sensu\_server --config solo.rb

Lets inspect the actual configuration file that Chef made for that:

cat /etc/sensu/conf.d/checks/check-disk.json

In your infrastructure you will have to make the call between defining standalone checks on the clients like this, or doing subscription checks and configuring clients to subscribe to certain tags. I like standalone checks myself, as it makes it more straightforward to add Sensu checks with particular wrapper recipes.

You could imagine a cookbooks that wraps apache stuff together. That cookbook might have a monitoring recipe included with it. This would keep the recipe kinda "self-contained", and any role that included that cookbook would get the monitoring with it, regardless of the tags it was subscribed to.

## Installing a Email Handler

Just like in the introductory course, lets get this sensu-server up and running with an email handler so you can get email alerts.

Handlers are defined and executed on the server, so this code will go along with the other server stuff.

We'll need the gem and the sensu-handler config:

vim cookbooks/sensu\_server/recipes/default.rb

sensu\_gem 'sensu-plugins-mailer'

sensu\_handler "mailer" do

type "pipe"

command "/opt/sensu/embedded/bin/handler-mailer.rb"

end

Now we need the configuration for the mailer plugin. For that we use the sensu\_snippet construct to add in some arbitrary config:

sensu\_snippet 'mailer' do

content(

'admin\_gui' => 'http://localhost:3000/',

'mail\_from' => 'sensu@localhost',

'smtp\_address' => 'localhost',

'smtp\_port' => '25',

'smtp\_domain' => 'localhost'

)

end

And now a new error:

STDERR: ERROR: Error installing sensu-plugins-mailer:

ERROR: Failed to build gem native extension.

/opt/sensu/embedded/bin/ruby extconf.rb

If you see this error, it might me one of two things. Maybe you are missing the build-essential package, which contains make, a compiler, and other things required to build native gem extensions. Or maybe you are like me and you were running a virtualbox image with the default 512MB of ram, which is not enough to run chef and gcc at the same time.

## **Conclusion**

I've kinda just scratched the surface of what you can use with Chef. Like any configuration-management tool, the big idea is to get your Sensu configuration down to something programmatic and reproducible.

Here we have reproduced the setup we had from the introduction course, with only a few lines of code that we had to write. Obviously we are standing on tall shoulders of those who wrote all the supporting cookbooks.

But, it will be up to you to make wrapping cookbooks that combine stock cookbooks with sensu monitoring and supporting checks and plugins.

Check out the external resources of this lecture for more information on this as well as all the commands and cookbooks used in this example.

# **How to Tune Sensu Check Parameters**

## **Anatomy of a Sensu Check Definition**

Once you have Sensu setup and running in production, you need to figure out how you are going to rate limit and throttle alerts that you get.

Say you have setup an email handler in combination with a disk check that runs every minute. You certainly don't need an email every minute reminding you that you are out of disk space!

This is where check tunables come in, let's talk about what they are and how to use them.

### How Checks Are Defined

Sensu checks can be defined with lots of different attributes. Now-a-days the Sensu [documentation](https://sensuapp.org/docs/latest/checks#definition-attributes) is excellent, and describes every possible attribute in detail.

Most of these attributes are self-explanatory, it isn't really necessary for me to just read these two you.

But the more-interesting aspect of these check definitions are those attributes that are not used by Sensu! That's right, the Sensu check definition language is extremely flexible you can add arbitrary key/values in the check definition and the Sensu core will simply ignore them, and not through a syntax error.

This is a big deal, and it means that you can extend Sensu to do really interesting things. Let me show you a few things that become possible when you allow this kind of arbitrary data passing.

### Sensu-plugin Attributes

Let's inspect some very popular [extra attributes](https://sensuapp.org/docs/latest/checks#sensu-plugin-attributes) that you can set for Sensu Handlers to use. To reiterate, these are extra setting you can just stick in your check definition for Sensu handlers to pick up on. Remember that Handlers are just scripts that are executed by Sensu when events fire. If the handler you are using utilizes the sensu-plugin gem (and almost every handler on the sensu-plugins Github project does), then they will respect these parameters.

Let's look at a few:

#### Occurrences

Occurrences: The number of event occurrences that must occur before an event is handled. The default is 1 here. That means by default handlers will begin to fire on first event. So if you are running check-http, and the website goes down, then the firsttime that check fails, the Sensu handler will do something. To be clear here, Sensu is calling this handler every time. For example if you are using the mailer handler like in our examples, Sensu is just running that mailer script every single time the check fails. But, the handler will filter itself based on this occurences setting.

For another example, let's say that you have a ping check, but you know that sometimes the network can be flaky, and you don't want to get an email unless the network is really down. Occurences can be one way to gloss over these failures. If this check has an interval of 60 seconds, and you set the occurrences setting to 5, then on the fifth time this check fails on that 5th minute, the handler will actually do something about it. If the check passes before that 5th time, then the check will reset, and no email would have been sent.

#### Refresh

Refresh is a time unit, in seconds, that represents the next layer of filtering. The default refresh is 1800 seconds, which is 30 minutes. This means that you should get 30 minutes between when the mailer handler will send you emails.

Keep in mind that this setting is independent of the occurrences setting. In other words, if you have a check that runs once a minute, and an occurrences setting of 15, that means you will get an email alert 15 minutes after the check started failing. You will then get another email on minute 30, because that is what refresh defaults to. You won't get another email till minute 60.

This is probably fine, but you should just be aware of how these parameters interact.

If you don't believe me or you would like to see for yourself, you can always look at the [sourcecode](https://github.com/sensu-plugins/sensu-plugin/blob/aa59019a584eae88f3e784d7079f59a762879418/lib/sensu-handler.rb#L108-L119) that controls this filtering behavior.

#### Dependencies

Dependencies is a totally underutilized parameter. With dependencies you can automatically not fire an email if a different check is already failing. You can even reference other hosts with this '/' notation.

A real world example of this might be something that operates via cron, and you know that if cron isn't running, then you don't need an alert on this thing too. If the name of your cron check is simply check\_cron, then you can just add dependencies: ['check\_cron'] to your check definition. If cron is already failing, then the alerts for the thing that depends on cron will be suppressed. This is especially helpful with network topologies and vpns. You don't need a billion alerts if you already know the vpn link is down. In sensu these dependencies are so easy, compared to other monitoring systems, because they are checks in a lazy way. It does mean that you do have to double-check your spelling. If you mis-spell your dependency name, Sensu doesn't know and it won't catch your typo.

### Custom check attributes

We've covered a few check tunables that you can use out of the box to help customize your alerting experience for you and your teams. Let's take moment to talk about taking this to a really interesting level: custom check attributes.

Say we have a check definition on disk already:

{

"checks": {

"check\_mysql\_replication": {

"command": "check-mysql-replication-status.rb --user sensu --password secret",

"subscribers": [

"mysql"

],

"interval": 30,

"playbook": "http://docs.example.com/wiki/mysql-replication-playbook"

}

}

}

This example is right out of the official [Sensu Docs](https://sensuapp.org/docs/latest/checks#custom-definition-attributes) Do you recognize any unusual check parameters? command, subscribers, and interval are all normal parameters for a check, but what is playbook? Playbook is a totally custom, random parameter that someone decided to add in here. You know what, Sensu does not see this as a syntax error. It liberally accepts check definitions as long as they are syntactically correct. Anything extra it just passes along in the event data.

But once it is in the event data, handlers can see it! In fact the stock sensu mailer handler [does intepret this parameter](https://github.com/sensu-plugins/sensu-plugins-mailer/blob/a8355875b5f732c212d5eeeb51f7188b836773e5/bin/handler-mailer.rb#L95)

The code here says, if there is a playbook defined in this check, we'll go ahead and add it into the body of the email.

So in this way you can annotate your check definitions with extra metadata about the check. The playbook field is a great example, but you could do anything! You could add datacenter, or team name, or SLA, the limit is only your imagination. You could either just store that in the check data, or you could go further and adjust your handlers to use that data.

In this case, we are just printing the playbook in the email of the body, but you could do a JIRA handler that takes in tags, or an EC2 handler that understands AZs, or maybe a multi-tenant environment that understands customer names. Maybe you work in a datacenter environment and you want all alerts to have an asset tag. Maybe you have a special role that each of your servers have, or you want servers that are in production to have a special prod tag. Once you have this tool in your toolbox, a lot of very interesting possibilities show up.

## **Conclusion**

Anyway, this to me is one of the more interesting aspects of Sensu that I've never really seen in any other monitoring system. Most monitoring systems have ways to tune how often you get an email, but not many allow arbitrary key/values that you can use for whatever you want.

Look at the external resources section of this lecture for more interesting examples of custom check attributes. I'll try to keep a good currated list of creative ways to use this feature.

List:

* [mailer-handler](https://github.com/sensu-plugins/sensu-plugins-mailer/blob/a8355875b5f732c212d5eeeb51f7188b836773e5/bin/handler-mailer.rb#L95) using the playbook field and adding it to the email body
* [remediation-handler](https://github.com/sensu/sensu-community-plugins/blob/master/handlers/remediation/sensu.rb#L27-L66) uses a special remediation attribute to describe ways to programmtically "fix" (remediate) an alert.
* [statuspage-handler](https://github.com/sensu-plugins/sensu-plugins-statuspage/blob/b5a8c4940536c4e0f0e51d980fea278e6d4075cf/bin/handler-statuspage.rb#L30) Uses a custom component\_id to automatically annotate status pages.
* [ansible-handler](https://github.com/sensu/sensu-community-plugins/blob/f807971cee35bfc59f2217073f1cca25f7236e2e/handlers/other/ansible.rb#L30) Takes a playbook attribute to execute a particular playbook
* [pagerduty-handler](https://github.com/sensu-plugins/sensu-plugins-pagerduty/blob/df80a30ce3705852c2f9eb25b6ad967b64aaa553/bin/handler-pagerduty.rb#L43) Understands a pager\_team for different pagerduty Teams.
* [jira-handler](https://github.com/Yelp/sensu_handlers/blob/5743cd89e9b4b9af9c3b0a45e3ac9e0ce801e569/files/jira.rb#L10) Builds up tags for helping collect related alert tickets in JIRA.

Also I'll like to the official docs on the tunables I mentioned in the lecture for adjusting the frequency of your alerts. They will definitely come in handy and any production Sensu engineer needs to have a firm grasp of them and what they do.

# **Writing Your Own Sensu Checks**

There comes a time in every engineers' life where they have a need to monitor something, where no existing monitoring script exists.

Luckily, writing check scripts is relatively easy, because the contract you have to follow is so straightforward and hasn't changed for... decades?

## **Your First Check Script**

Let's write a very basic script in bash at first, and then we'll port it to Ruby before it gets too complex.

For the lecture we are going to build a Sensu check to check to see if a file exists. There are certainly existing checks like this that already exist, but I find it helpful to do this exercise when we all already know what this check is supposed to do.

But I would like to actually get my TDD on and write some tests first for this thing:

#!/bin/bash

function assert {

if [[ $1 -eq $2 ]]; then

echo Pass

else

echo "Fail. Expected $2 but got $1"

fi

}

touch test\_file

./check\_file\_exists test\_file

assert $? 0

rm -f test\_file

./check\_file\_exists test\_file

assert $? 2

So we have a little bit of a test framework to make sure our code will do what we think it should do. On the first test case where the test\_file exists we assert that after the check runs the return code should be 0, which means "ok"

On the second case we remove the file and run our script and assert that the return code is 2, which means critical. If you are curious about these return code conventions, they are pretty old and very standard:

https://sensuapp.org/docs/latest/checks#what-are-sensu-checks

Pretty much every monitoring system that works in the linux world follows these same conventions.

So let's run our tests!

chmod +x test\_check\_file\_exists

./test\_check\_file\_exists

Of course our tests fail, as we haven't written any code yet. 127 is the shell's return code when the script doesn't exist. Let's write some code to make our tests pass.

vim ./check\_file\_exists

#!/bin/bash

Now for for the most naive version of this check, we can simply use the basic test functionality of the shell:

[[ -f $1 ]]

We this might work. In bash the final return code here will be whatever the result of this test is. Does it pass our tests?

chmod +x check\_file\_exists

./test\_check\_file\_exists

It doesn't. The second test case fails because this test returns 1 when the file doesn't exist, and we were expecting a 2.

Maybe we can write more than one line of code and make this check a tad more user friendly:

#!/bin/bash

if [[ -f $1 ]]; then

echo "OK: File '$1' exists and is a normal file"

exit 0

else

echo "Critical: File '$1' doesn't exist or is not a normal file"

exit 2

fi

This is a bit more readable. Let's run the tests:

./test\_check\_file\_exists

Both pass now. You can see that writing Sensu checks is really not that bad, all you have to do is conform to the return code conventions. Adding extra output for humans is nice too. I really like it when checks return as much human-readable friendly output as possible.

## **Porting to Ruby**

Bash is fine, but what if we wanted to make this check be more flexible, and have options like "negating" the check, so it is ok if the file doesn't exit?

You can do this in bash, but more complex logic is easier to do in a language with more constructs. But guess, what: you have a language interpreter at your fingertips that is guaranteed to be deployed with every Sensu client that you deploy: the Sensu embedded ruby!

You get a modern version of ruby with lots of goodies.

Let's port our check to use this Ruby so we can expand it's functionality more easily.

#!/opt/sensu/embedded/bin/ruby

require 'sensu-plugin/check/cli'

Now this is a bit controversial: hard-coding the she-bang here to use the embedded Sensu ruby is a bit inflexible. It is true. If a user wanted to use a different ruby, they would have to literally edit this script and modify it.

On the other hand, it is explicit about the exact ruby you are expected to use. If you didn't have this she-bang here, a user might execute it using the system ruby, which probably doesn't have the sensu-plugin gem that you requested.

It is a tossup. For open-source plugins it makes Sensu to just be generic and use whatever ruby is set in the environment, but for plugins I develop in production, I hard-code the path because I never want it to be accidentally run, by a machine or by a human, with the wrong ruby.

Anyway, let's run our tests just with what we've got.

./test\_check\_file\_exists

Sensu::Plugin::CLI: ["Not implemented! You should override Sensu::Plugin::CLI#run."]

So now we need to talk a little bit about this class. The sensu-plugin gem comes with a lot of helpers to help build Sensu-plugins and handlers and stuff.

Here we want to build a command line check script. Here is how you use this class to build this check:

#!/opt/sensu/embedded/bin/ruby

require 'sensu-plugin/check/cli'

class CheckFileExists < Sensu::Plugin::Check::CLI

def run

end

end

To use this class we have to inherit from it, and have a run method. This run method is what is invoked when someone executes it from the command line.

./test\_check\_file\_exists

CheckFileExists WARNING: Check did not exit! You should call an exit code method.

Closer. We haven't told the check to exit properly. Let's do what it says and use an exit code method.

What methods can we call? Well:

https://github.com/sensu-plugins/sensu-plugin/blob/b679e239a63d7c206bada044f67f43834d44e33f/lib/sensu-plugin/cli.rb#L26

Although this is a bit meta, we can call a method for every type of exit code. What were those exit codes again?

https://github.com/sensu-plugins/sensu-plugin/blob/69ac44f539d07bf044eb2b1370c36230fd00524f/lib/sensu-plugin.rb#L4

Now we should have enough data to write our check:

#!/opt/sensu/embedded/bin/ruby

require 'sensu-plugin/check/cli'

class CheckFileExists < Sensu::Plugin::Check::CLI

def run

filename = argv[0]

if File.exists?(filename)

ok "File '#{filename}' exists!"

else

critical "File '#{filename}' doesn't exist!"

end

end

end

This isn't supposed to be a ruby lesson, but just a demonstration of what it looks like to write sensu checks in ruby using the standard sensu-cli plugin constructs. It reads pretty well I think, but currently just implements our bash version. Do our tests pass?

./test\_check\_file\_exists

So yea, pretty much the same functionality as the bash version. What have we gained? Well, in theory this check would work on systems, that maybe don't have bash? Like Windows or BSD systems?

Also we are in a better position to add more logic and integrate with other ruby libraries that interface with more interesting things. For example in ruby you can get access to really good AWS libraries or http libraries that might be cumbersome to replicate in bash.

### Expanding the check with negation

Let's expand this check in ruby to include the option to negate the check, that is return "ok" the file isn't there, and critical when it is there.

Let's write our tests first:

touch test\_file

./check\_file\_exists --inverse test\_file

assert $? 2

rm -f test\_file

./check\_file\_exists --inverse test\_file

assert $? 0

I can't think of a much better term than just "inverse", as we are checking the inverse of what we normally would be checking.

./test\_check\_file\_exists

Invalid check argument(s): invalid option: --inverse, ["/opt/sensu/embedded/lib/ruby/gems/2.0.0/gems/mixlib-cli-1.5.0/lib/mixlib/cli.rb:191:in `parse\_options'", "/opt/sensu/embedded/lib/ruby/gems/2.0.0/gems/sensu-plugin-1.2.0/lib/sensu-plugin/cli.rb:13:in `initialize'", "/opt/sensu/embedded/lib/ruby/gems/2.0.0/gems/sensu-plugin-1.2.0/lib/sensu-plugin/cli.rb:55:in `new'", "/opt/sensu/embedded/lib/ruby/gems/2.0.0/gems/sensu-plugin-1.2.0/lib/sensu-plugin/cli.rb:55:in `block in <class:CLI>'"]

Our test fail of course because such an option doesn't exist. Let's add it.

To add this option, let's read up on the docs! The Sensu-plugin cli construct uses the milib-cli gem to do command line parsing.

This adds a very easy to use dsl for adding command line options:

https://github.com/chef/mixlib-cli

option :inverse,

:long => "--inverse",

:description => "Return OK if the file doesn't exist, Critical if it exists",

:boolean => true

You can call this magic, or you can just call it abstraction, but this construct takes most of the gotchas out of command line parameter parsing.

Now we can use this option in our code to invert our logic:

if File.exists?(filename)

if not config[:inverse]

ok "File '#{filename}' exists!"

else

critical "File '#{filename}'exists!"

end

else

if not config[:inverse]

critical "File '#{filename}' doesn't exist!"

else

ok "File '#{filename}' doesn't exist!"

end

end

There is almost certainly a better way to do this, but let's see if this works. Notice how the command line parameters show up in this config dictionary for use.

With this new flag we automatically get some command line help options now:

$ ./check\_file\_exists --help

Usage: ./check\_file\_exists (options)

--inverse Return OK if the file doesn't exist, Critical if it exists

Pretty cool. And do our tests pass now?

./test\_check\_file\_exists

They all pass, so now we have expanded our check to include the inverse logic in case you need to be sure a file doesn't exist.

## Conclusion

In conclusion, writing Sensu checks is easy. You can use any language, even just plain bash, but you also have Ruby and some sophisticated constructs to reach for you want to.

But don't forget to not-reinvent the wheel. Not only are there plenty of existing community Sensu plugins available for you to re-use, but there are also tons of other monitoring plugins that written for other tools that are also compatible, thanks to the Nagios-compliant return-code api.

So go forth and write Sensu checks like a pro!

### Further Reading:

* [Sensu-plugin documentation](https://github.com/sensu-plugins/sensu-plugin)
* <https://github.com/chef/mixlib-cli>
* <https://docs.sensu.io/sensu-core/1.4/guides/intro-to-checks/>

### Final version of the check\_file\_exists

#!/opt/sensu/embedded/bin/ruby

require 'sensu-plugin/check/cli'

class CheckFileExists < Sensu::Plugin::Check::CLI

option :inverse,

:long => "--inverse",

:description => "Return OK if the file doesn't exist, Critical if it exists",

:boolean => true

def run

filename = argv[0]

if File.exists?(filename)

if not config[:inverse]

ok "File '#{filename}' exists!"

else

critical "File '#{filename}'exists!"

end

else

if not config[:inverse]

critical "File '#{filename}' doesn't exist!"

else

ok "File '#{filename}' doesn't exist!"

end

end

end

end

# **Writing Your Own Sensu Handlers**

A "handler" is the name for the piece of code that Sensu executes in response to recieving and event.

In the introductory course I mentioned that handlers simply take input from stdin and then act on that input. And that is true!

Handlers can be as simple as just the cat command, that take the input and spit it back out. For this lecture will make a more sophisticated handler.

## What Language Handlers are Written In

Although you could write a Sensu handler in any language, most of them are written in Ruby. Handlers are written in ruby to give them access to a lot of helping ruby methods that come with the sensu-plugin gem. These are helper for things like reading config out of /etc/sensu, or handling the situation where checks are silenced.

Still though, when writing handlers in Ruby, you get a lot of boilerplate for free. Similar to how when we wrote the check script in ruby, we got a lot of similar command line parsing things that seems to be kinda "magical", when we use the sensu-plugin class for handlers, we will also get a lot of cool stuff.

## Getting Started

There isn't a ton of [existing documentation](https://github.com/sensu-plugins/sensu-plugin#handlers) on how to write handlers, but there are lots of examples and prior art.

Let's start by copy/pasting the [show handler](https://github.com/sensu/sensu-community-plugins/blob/master/handlers/debug/show.rb) , which is kinda the most minimal handler there can be:

#!/opt/sensu/embedded/bin/ruby

require 'sensu-handler'

class Show < Sensu::Handler

def handle

puts 'Settings: ' + settings.to\_hash.inspect

puts 'Event: ' + @event.inspect

end

end

Like before I'm going to make the call to hard-code the Sensu embedded ruby in my shebang for the lecture. You will have to decide for yourself if this is what you want in your own environment.

How are we going to actually test this handler? Certainly we could reproduce a server environment and a purposely failing check. But I have a better idea. Remember that handlers just take input from std-in and then operate on it? Let's just give it some stdin!

We can just steal some example [event data](https://sensuapp.org/docs/latest/events#sensu-event-data) from the official sensu documentation page.

cat > test-data.json

...

cat test-data.json | ./handler.rb

And look! The Show handler did what it says it does, it prints out the settings and prints out the event data.

## Writing a Real Handler

So what kind of handler are we going to write? I toyed with the idea of writing a trivial handler, but this time I'm actually going to take this opportunity to write a handler that I've actually always wanted: a "Notify My Android" handler.

Notify My Android is a service for android phones to receive arbitrary push notifications. It is extremely easy to use. I would like to get a NMA alert from Sensu, so let's build a handler.

Building this handler is going to be very easy thanks to this existing [NMA rubygem](https://github.com/slashk/ruby-notify-my-android#usage-as-a-gem)

I'm just going to copy in this example to get started:

#!/opt/sensu/embedded/bin/ruby

require 'sensu-handler'

require 'ruby-notify-my-android'

class Show < Sensu::Handler

def handle

NMA.notify do |n|

n.apikey = "9d1538ab7b52360e906e0e766f34501b69edde92fe3409e9"

n.priority = NMA::Priority::MODERATE

n.application = "NMA"

n.event = "Notification"

n.description = "Your server is under attack!!!"

end

end

end

Of course I need this gem installed, so I'm going to use the Sensu omnibus ruby to install it:

/opt/sensu/embedded/bin/gem install ruby-notify-my-android

## Making the Handler More Sane With Settings

Much of this will need to be replaced with real values from Sensu. Let's start with the API key. I could certainly hard-code an API key in here, but that would not be very flexible.

You saw in the show handler that we have the ability to simply read in all of Sensu's configuration. This makes it really easy to add config for a handler!

Let's make the config file we want, and then have the handler read it:

cat > /etc/sensu/conf.d/notify\_my\_android.json

{

"notify\_my\_android": {

"api\_key": "abc123"

}

}

Behind the scenes I'm going to replace that with a real API key so this will actually work of course.

But you can see how straight-forward this is, it is just a config file with a unique top-level key. In this case the key is notify\_my\_android. It is customary to have the name of the key be the name of your handler.

mv handler.rb notify\_my\_android.rb

Now to use this config, we can just use the settings hash:

n.apikey = settings["notify\_my\_android"]["api\_key"]

So great. Really the ruby glue provided by the sensu-plugins gem makes everything available to you as you need it.

## More Adjustments

We can change our application to Sensu:

n.application = "Sensu"

What should the event key be? Probably something about what is wrong on what host:

n.event = @event['client']['name'] + '/' + @event['check']['name']

For the description we can use this hander [event summary](https://github.com/sensu-plugins/sensu-plugin/blob/aa59019a584eae88f3e784d7079f59a762879418/lib/sensu-handler.rb#L61) method.

n.description = event\_summary

Well, it isn't many lines of code, but I don't think we need many? Does it it work?

cat event-data.json | ./notify\_my\_android.rb

It is kind strange to see no command line output though, that will make this handler pretty difficult to debug when things go wrong. Let's simply print the actual response we get back from the NMA method:

def handle

response = NMA.notify do |n|

n.apikey = settings["notify\_my\_android"]["api\_key"]

n.priority = NMA::Priority::MODERATE

n.application = "Sensu"

n.event = @event['client']['name'] + '/' + @event['check']['name']

n.description = event\_summary

end

puts response.inspect

end

cat event-data.json | ./notify\_my\_android.rb

That is better. It could be made more pretty, but at least we get something.

## Conclusion

You can see that there is really not much here, and that is because we get so much from Sensu::Handler class that we are inheriting. Getting settings for our handler, like an api key is super easy, thanks to the automatic settings variable we have access to. All of these event an client variables are given to use in a sanitized way from the standard-in, but I could imagine we could do fancier things with them.

Speaking of fancy, on my lecture on tuning alerts, remember the refresh and occurences settings? This handler already respects those. That is part of the filtering logic of the Sensu::Handler class that we inherited.

However, if you needed to you could override that method with your own, if notify-my-android need special filtering.

But even with such little code our handler is mostly functional! When you take the existing methods you get from the Sensu::Handler class and you combine them with existing ruby gems that exist for just about anything, you get a great combination that allows you to build new integrations with Sensu, with very little effort.

I am by no means a "Ruby guy", but I can handle this.

So go forth and do not be afraid to build custom integrations with Sensu, as you can see you can get very far with a very minimal product. With a little glue, Sensu can integrate with just about anything!

## Further Reading

* [Official Documentation](https://github.com/sensu-plugins/sensu-plugin#handlers)
* [Notify My Android Rubygem](https://github.com/slashk/ruby-notify-my-android)
* [Sensu Show Handler](https://github.com/sensu/sensu-community-plugins/blob/master/handlers/debug/show.rb)

# **Taking Advantage of the Embedded Omnibus Ruby**

## Intro

You may remember me mentioning before in the introductory course that Sensu is distributed as an "omnibus package". That means it comes its own isolated ruby interpreter and gems, completely isolated from the system-installed Ruby, if any if the system even has Ruby installed.

This is a good thing for Sensu, it means you have predictable deployments, regardless of the distro you are on.

In previous lectures I've demonstrated how to use this Ruby to install handlers and checks. But say you really want to use this Ruby for everything, so much that whenever Sensu runs anything Ruby related it will use the embedded Ruby.

## Advantages

### Method 1: EMBEDDED\_RUBY=true

There are two ways you can take advantage of this. The first way is by setting EMBEDDED\_RUBY=true in /etc/default/sensu:

vim /etc/default/sensu

If you set this to true, then whatever service that runs on this host, either the sensu-client or the sensu-server, will have the omnibus-ruby in the path first. This means any ruby-based plugins will use it instead of whatever system ruby is installed.

#### Mechanics of EMBEDDED\_RUBY=true

But how the heck does EMBEDDED\_RUBY=true actually work? Let's find out where that actually activates.

https://github.com/sensu/sensu-build/blob/f872cc5f3d0345b636e73505b49e8230c836b0a8/sensu\_configs/default/sensu

This is where EMBEDDED\_RUBY is actually set. You can see right now it defaults to false. This is definitely an opt-in kinda thing, it would be a little surprising if the Sensu-ruby took over by default.

But if you know you want to use Sensu's Ruby as the default ruby, and you set this to true. What does that actually do?

https://github.com/sensu/sensu-build/blob/f872cc5f3d0345b636e73505b49e8230c836b0a8/sensu\_configs/init.d/sensu-service#L166

This is an example of one of the init scripts. You can see that if this is true, the init script will set the embedded Ruby's bin folder to be first in the path. Also it will set the GEM\_PATH to use the Sensu-embedded-ruby's Gems first, instead of what might be on the system.

That's it, no magic, just bash. Nothing specific to the Sensu user or anything, it just sets the PATH and GEM\_PATH.

On the one hand this can be convenient, but it does mean that if you need to reproduce exactly how Sensu is executing checks or handlers, you need to remember to do the same thing. It isn't very explicit, which is I prefer the second method...

## Method 2: Hard-coding Shebangs (#!)

The other way to use this ruby is to make the she-bang directly invoke it. This is a little easier to do if you control the script, and it is not a host-wide setting, which is kinda nice.

This is the method I've used in all the other lectures so far, mostly because it is so reproducible. It doesn't matter what your environment variables are, you can run the command with the full path and it will always use the right ruby.

## Looking at the ruby

Lets see what we have to work with:

/opt/sensu/embedded/bin/ruby --version

So Ruby 2.0.0, which is decently new. Lets see what gems are available:

/opt/sensu/embedded/bin/gem list | less

These are the actual gems that Sensu uses, so be careful when adjusting these.

## Using the Ruby

Of course if you want to use the embedded ruby on a script you can simply put that ruby into the she-bang of the script.

#!/opt/sensu/embedded/bin/ruby

And of course you can simply invoke a ruby script with that interpreter directly:

/opt/sensu/embedded/bin/ruby my-script.rb

It may be more verbose, but it is explicit about exactly which ruby to use.

## Conclusion

That is all I really wanted to say about the embedded ruby. In the other lectures you've seen me use it to install and run handlers and checks, but I just wanted to touch now what it means to set EMBEDDED\_RUBY=true. You don't need to set it to be true to take advantage of the omnibus ruby, you can simply call scripts by their full path instead. You should only really setEMBEDDED\_RUBY=true if understand what it means to set it. And of course all it does is change the paths in the init script, no magic. But now you know!

# **Standalone, Safemode, and Subscriptions**

When running Sensu in production, you need to be aware of how Sensu schedules checks. The ideas of "standalone checks", "subscription checks", and "safemode" are all related ideas that I think are very important, and worth devoting a small lecture to explain what they mean.

Let's go to the whiteboard to explain...

## Sensu's Scheduling Model

In order to talk about how Sensu's check scheduling model works, we have to talk about RabbitMQ. If you have a Sensu client and a Sensu server, RabbitMQ is the only method of communication between the two, that is it.

But you have to remember that RabbitMQ is a two-way street here. Not only can Sensu clients put check results onto the queue for processing, but the Sensu server can also put check requests onto the queue for clients to act on.

Your Sensu configuration and your check configuration will determine how this behaves exactly.

## Subscription Checks

Let's talk about "Subscription" checks first. Subscription checks are checks that are scheduled by the Sensu Server. In order for them to work, you need at least two pieces of configuration in place. This will be more obvious with an example.

Let's say you have a Sensu server and a bunch of Sensu clients running on your cluster of webservers. To setup a subscription-based check for something like check\_http, you need the actual check defined on the Sensu server. It will have a flag for which subscribers should run this check, you might say "webserver" need to subscribe to it. Once this is in place, on the regular schedule, the Sensu server will begin to put out check requests onto RabbitMQ, calling our for anything subscribed to "webserver" to execute this check.

But who is listening? Any server client that has been setup with "webserver" in the client subscriptions configuration will be listening for these check requests. When the server puts out that call for check\_http, the clients that are subscribed will pick up on it, execute the check, and put the results back onto the queue for the server to process. The advantage to this model is that you have central configuration and control over the checks, you only have one place where this check is defined, and conceptually it is similar to how most monitoring systems work, more or less. In a sense the Sensu clients are kinda acting like just "dumb" agents that are executing checks on the host, for the server.

## Security Implications / safe\_mode

Now if you are like me, with a decent amount of sysadmin experience, your first reaction to this might be, "wait, the Sensu clients just do whatever the server tells them to do? This sounds like a "remote execution exploit as a service" setup.

And it is true, anyone with access to RabbitMQ in this setup could ask the Sensu clients to do anything, really. Granted processes are spawned under the Sensu user, it still seems like a pretty relaxed security model. Personally, I would not run Sensu like this in a production setup, without what is called: "safe\_mode".

With safe\_mode on, Sensu clients will refuse to run any check that has not already been defined in their local configuration. This is certainly a little safer, but now it means that you must pre-configure your clients with the checks that they are going to run. That means now they each have to have a file with the check\_http check defined on them.

If you ask me this should be the default, but I understand why it is not: out of the box it restricts how useful Sensu is in this subscription mode. But this is certainly something engineers need to be aware of when they are integrating Sensu with their infrastructure.

I personally don't see a problem with requiring a config file with this check definition, do you know why? It is because I use configuration management! If you haven't watched any of the lectures on using configuration management tools with Sensu, I encourage you to do so. If you are using configuration management to deploy Sensu and setup the list of subscriptions, then it isn't that big of a deal for the configuration management tool to also deploy the config file for the check\_http check at the same time. After all, that is really what configuration management tools do best.

## Standalone Checks

But there is another option: standalone checks. Standalone checks are defined on the client only. Also the client is responsible for actually scheduling this check on itself. In fact, even if the Sensu server is down, standalone checks will continue to run on their intervals, because it is the job of the Sensu client.

You can define a standalone check\_http check on any server, regardless of it's subscriptions, and the Sensu client will see it, schedule and execute it, and that is it. The nice thing about this setup is that no configuration on the Sensu server is needed at all. Any client with standalone checks just operate on their own, and the Sensu server just operates on the results of those checks.

The downside is that now you must deploy this configuration file to every webserver, you can no longer centrally control it. Again though, this is exactly what configuration management is for. If configuration management is setting up your webservers, it can deploy this Sensu config file too. In this world, safe\_mode doesn't apply. You should certainly turn it on, but it doesn't affect the behavior of standalone checks.

As I've said before, I like standalone checks myself, I think they are actually easier to understand than subscription checks, and I like how easily they fit with the configuration management model. If you are using configuration management to deploy a webserver on a particular port, and you wanted to change that port, the configuration management tool could change the port of the webserver and update the monitoring configuration on the same host, using standalone checks, and that is it.

With the subscription model you might update your webservers, but then you have to update the Sensu server too. But what if the rollout is slow? Well in the standalone world, the app and the monitoring config are on the same server, so they can change kinda atomically together. I think this is cool. You could do this with subscriptions as long as you had the check defined on the server already, and the local settings will override the check definition that the server sends out, but at that point it seems like you might as well just do a standalone check.

## Aggregate Notes.

I should note that there is an advanced Sensu topic called "aggregates". Aggregates are a way for the Sensu server to schedule a check across a set of subscribers and then tally the results to give you a kind of "aggregate" view. Aggregates don't work on standalone checks because they are not centrally scheduled.

## Conclusion

So in conclusion, "subscription" checks are defined and scheduled on the server, and then clients that are tagged with the same subscription pick them up, execute the check, and then put the results back on the queue.

With safe\_mode on, these clients will only execute those checks if the check is also defined locally, for safety.

"standalone" checks are checks that are defined locally on the client-only, they don't need any server-side configuration. safe\_mode doesn't affect them.

I hope this makes this topic very clear. It is an idea unique to Sensu's architecture, so I thought it was worth clarification. If you are using configuration management, I find that standalone checks work well for that. If you are setting up checks by hand, subscription checks are easy with safe\_mode off, but you do need to be aware of the security implications.

Like always, I'll have additional documentation on these topics as well as show notes in the external resources section of this lecture.

# **Handlers, Filters, and Subdued Checks**

Let's talk about filters. Eventually when you scale out any monitoring system, you come to the conclusion that not all alerts are created equal. Sensu's filter language is one tool you have to describe certain policies, to filter out certain alerts.

## How Filters Work

A Sensu filter will limit what types of events a handler will see:

https://sensuapp.org/docs/latest/getting-started-with-filters#create-an-event-filter

The official documentation on this subject is actually really good, so I don't feel the need to go over this subject too much, but I think it is worth doing an example.

The first thing you need to understand about filters, is that they always operate on event data. Let's look at some example event data to refresh our memories:

https://sensuapp.org/docs/latest/events#sensu-event-data

There are lots of interesting things in this dictionary to use for filters.

Let's look at the example in the docs:

https://sensuapp.org/docs/latest/getting-started-with-filters#inclusive-filtering

You can see that filters are defined in the filters configuration namespace, then next is the name of the filter, in this case the name of the filter is production. Look at the next key, attributes. Here we are defining the attributes of the event data that are relevant for filtering. Next is client, so this filter has something to do with the client section of the event data, and then environment: production. But.. I don't see `environment in our example event data:

<https://sensuapp.org/docs/latest/events#sensu-event-data>

## Custom Client Attributes

This hints on a topic that we haven't really covered, and that is: Custom client attributes. Remember in the lecture about turning Sensu checks, we said that you can add custom data into the check for handlers to use? These were things like "playbook", and if the "playbook" key was in the check definition, the email handler would see that and stick it in the email output.

In the same way that you can define custom check attributes, you can also define custom client attributes:

https://sensuapp.org/docs/latest/clients#custom-definition-attributes

Any custom fields here will be ignored by Sensu and just passed on to handler and filters for use. The example in the docs here is for MySQL, but environment is a good one too. Annotating your clients with metadata likes this means that handlers and filters can use it. Let's take our client on this test server and annotate it with environment: production.

cd /etc/sensu/conf.d

vim client.json

And now let's restart the sensu-client to pick up that change:

sudo restart sensu-client

Sensu pretty much will just ignore this extra data, it is not a syntax error.

Now our filter kinda makes more sense. The filter will make it see only those events with environment: production in the client section of the event data.

Well we just made our local client have this environment attribute, let's apply this filter and see if we can filter it out anything non-production:

{

"filters": {

"production": {

"attributes": {

"client": {

"environment": "production"

}

}

}

}

}

Filters are consumed by the sensu-server, so we have to restart the sensu server to pick up on it.

sudo restart sensu-server

Now, is our the checks being filtered?

tail -f /var/log/sensu/sensu-server.log

It isn't. Why not? Because we haven't told sensu to connect the pieces between this filter we created, and the handler that is executed. In this case, it is our mailer handler. We need to tell our mailer handler to use the filter we created, so that we don't get emails for things in the production environment.

"filters": ["production"]

Note here that I'm calling out this filter by name, and "production" was the arbitrary name of the filter. Now we should restart the sensu server one more time, and then see if it worekd.

https://sensuapp.org/docs/latest/filters

sudo restart sensu-server

tail -f /var/log/sensu/sensu-server.log

Now if we set our client to have the production environment attribue, the filter should start to see it and execute the handler.

vim /etc/sensu/conf.d/client.json

sudo restart sensu-client

tail -f /var/log/sensu/sensu-server.log

Now that our client now has the production attribute, the handler sees it and sends us email again.

## Subdue

The "Subdue" mechanism is another tool that you can use, to quiet certain checks during specific time period. This is useful for checks during what you might call "quiet hours":

https://sensuapp.org/docs/latest/checks#subdue-attributes

All of the subdue attributes are time-centric, they don't have anything to do with client attributes. Also you should note that this kind of filtering is on a per-check basis, they are not tied to a particular client or client attribute.

This is not quite the same as filter, but it kinda acts like one, so I figured I would mention it. Sometimes you just want to subdue checks that are not relevant to check during say, non business hours. Other times you want to filter our checks that operate on custom client or check attributes, like things that are in the production environment.

## Conclusion

As always, be sure to check out the external resources section of this lecture for exact links to the Sensu documentation on these features for filtering out Sensu alerts.

# **Securing RabbitMQ With SSL**

Let's talk a bit about why we care about SSL with RabbitMQ. In Sensu, RabbitMQ is the primary way that messages are sent between the client and servers. This whole time we have not been using SSL, which means all traffic between the Sensu client and server has been unencrypted. This is a potential security risk, especially if you are going across untrusted networks. This is especially important if you are not using safe\_mode, and the Sensu clients will execute arbitrary code upon a check request.

SSL encrypts this traffic. But, enabling SSL means we have an additional burden of getting SSL certs. For this lecture we will be creating our own certificate authority and issuing our own self-signed certs. At the end of the lecture I'll discuss the pros and cons to this approach, and talk about some other options.

## Making a CA, Signing Some Certs

The official Sensu documentation actually comes with some helping scripts to create our own Certificate Authority and sign our own certs.

https://sensuapp.org/docs/latest/ssl

The docs are a little sparse, but let's download the helper script and see how far we can get...

cd /tmp && wget http://sensuapp.org/docs/latest/tools/ssl\_certs.tar && tar -xvf ssl\_certs.tar

cd ssl\_certs

ls

vim ssl\_cert.sh

./ssl\_certs.sh generate

find

As you can see the script has generated some certs and keys for us to use, including one for a server, a client, and the CA. The are not anything really fancy, just basic self-signed certs:

openssl x509 -in server/cert.pem -text -noout

openssl x509 -in client/cert.pem -text -noout

## Adding Certs to RabbitMQ

Now that we have some certs and keys to work with, let's copy them into RabbitMQ's config directory for use:

mkdir /etc/rabbitmq/ssl

cp server/key.pem server/cert.pem sensu\_ca/cacert.pem /etc/rabbitmq/ssl/

Now we can configure RabbitMQ to use them. I'm for refererence, the [RabbitMQ documentation page](https://www.rabbitmq.com/ssl.html) has exact instructions on how to enable SSL listeners:

cd /etc/rabbitmq

find

vim rabbitmq.config

{rabbit, [

{ssl\_listeners, [5671]},

{ssl\_options, [{cacertfile,"/etc/rabbitmq/ssl/cacert.pem"},

{certfile,"/etc/rabbitmq/ssl/cert.pem"},

{keyfile,"/etc/rabbitmq/ssl/key.pem"},

{verify,verify\_peer},

{fail\_if\_no\_peer\_cert,true}]}

]}

/etc/init.d/rabbitmq-server restart

And let's look at the logs:

tail -f /var/log/rabbitmq/rabbit@vagrant-ubuntu-trusty-64.log

You can see that we haven't adjusted Sensu's configuration yet, so those clients are still connecting to the non-ssl port, 5672.

## Adding Certs to Sensu

Now that RabbitMQ is listening and ready to accept SSL connections on 5671, we are ready to give client certs to Sensu and adjust its configuration:

cd /tmp/ssl\_certs/

mkdir /etc/sensu/ssl/

cp client/key.pem client/cert.pem sensu\_ca/cacert.pem /etc/sensu/ssl/

vim /etc/sensu/config.json

"rabbitmq": {

"ssl": {

"cert\_chain\_file": "/etc/sensu/ssl/cert.pem",

"private\_key\_file": "/etc/sensu/ssl/key.pem"

},

"host": "localhost",

"port": 5671,

Remember both the Sensu server, client, and API connect to RabbitMQ for communication, so they will all need to be restarted to pick up this configuration:

/etc/init.d/sensu-client restart

/etc/init.d/sensu-server restart

/etc/init.d/sensu-api restart

This config file is shared by all three on this host, in practice you might have different components on differnet hosts, and they will all need this treatment. Let's look at the RabbitMQ log:

tail -f /var/log/rabbitmq/rabbit@vagrant-ubuntu-trusty-64.log

Now you can see connections are coming in on port 5671. And let's verify that the Sensu components are still connected:

tail /var/log/sensu/sensu-server.log -n 200

And that uchiwa still works:

http://localhost:3000/#/datacenters

So we did it, we got Sensu and RabbitMQ to talk over SSL, with no downtime. Normally I would do something where I would enable SSL on RabbitMQ, disable non-ssl, watch Sensu fail, then enable SSL on Sensu to watch it work again. But in this case I wanted to demonstrate that this configuration change can be done in a production environment with no downtime, as long as the steps are followed in this order.

## Alternatives

Setting up SSL can be a pain, but once you have it setup, it is done. I don't mind so much that it requires this effort up front and Sensu doesn't do it automatically for you. Honestly, I wouldn't want Sensu to do it magically for me, just like I wouldn't want Apache or Nginx to setup SSL certs for me.

The fact that it is self-signed also doesn't bother me too much, is isn't like someone is going to see this in a browser.

Alternatively though, if you already have a self-signed SSL cert setup in your environment, say for an existing setup like Puppet or Chef, you could totally use that. You already have a certificate authority, and each server has it's own signed private key, it would totally work.

On the other hand, you may want to just go through the work of setting up a different SSL cert for Sensu, just so you don't risk the possibility of breaking your monitoring and your configuration management at the same time.

SSL adds extra protection to the actual traffic on the wire for Sensu, and I think it is absolutely worth it for any production environment. Of course I always recommend using a configuration management tool like Chef/Puppet/Salt/Ansible, to make it easy to deploy this configuration in a reproducible way.

As always, look to the external resources section of this lecture for links to the official documentation on how to configure SSL with RabbitMQ for Sensu, as well as the exact commands I ran, and links to more tutorials for doing this procedure.

# **Hardening the Rest of Sensu**

If you are running Sensu in production, you need to be at least aware of what Sensu's attack surface is, from a security prospective. Obviously different environments warrant different levels of security precautions. Even if you don't do anything about all of pieces, you should at least be informed.

In this lecture I'll discuss some potential places where Sensu's security footprint can be improved, although I won't actually be demonstrating how to do so.

## RabbitMQ

We talked before about using SSL with RabbitMQ to encrypt communication between RabbitMQ and the other components. You should know that by default RabbitMQ does come with a guest account, even though it can only be used locally. In a production environment you should probably remove the guest account, and ensure that the Sensu credentials are strong.

If you can, I recommend firewalling off everything except the SSL port. The SSL port 5671, is the only port the Sensu components need to use.

Configuring RabbitMQ and Sensu to talk over SSL actually is demonstrated in a separate lecture.

## Redis

The only thing that needs to talk to Redis is the Sensu-server and Sensu-API components. If possible, try to lock down the access to the redis port to only those things that need it. Do not expose Redis to the internet.

If you absolutely must use untrusted networks to communicate with Redis, it is advised to use stunnel or a vpn to encrypt your traffic.

Redis does store things like silences and previous check output, which could potentially store secrets.

## Sensu API

The Sensu API is an http interface, and should not be exposed to the outside world unprotected. The Sensu-api configuration has settings for enabling http-basic auth, and you can a SSL terminating webserver like nginx in front, to encrypt any traffic that goes to the api.

Don't forget that if you do enable authentication and SSL on the Sensu API, any tool that utilizes the API will need to be updated to use SSL and authentication. This means at least the dashboard configuration will need to be updated.

## Sensu Client

The Sensu client by default exposes port 3030 on localhost only, for pushing external event data. If an attacker got on the local host, they could potentially send arbitrary check results, potentially DOS'ing your infrastructure. I'm not currently aware of a way to disable the client socket, but it is a potential attack vector. Again the attacker would have to have access to localhost.

## Sensu Server

The Sensu server has no externally facing endpoint.

## Dashboards

The Sensu dashboard is another http endpoint that can use hardening. Just like most web frontends, it is recommend to put SSL termination in front, and of course use some sort of authentication.

The stock Uchiwa dashboard has very basic authentication, and the enterprise dashboard has more fancy authentication. Either way, an SSL-terminating webserver should be placed in front of the dashboard. Alternatively you can make the SSL-terminating webserver, say, apache or nginx, do authentication for you.

## Conclusion

The general philosophy here is defense in depth: use firewalls, authentication, and encryption where possible.

Luckily Sensu uses existing components and traditional HTTP endpoints, which have existing, well-known best-practices for securing them.

# **Installing and Using the Sensu-cli**

## Intro

The sensu-cli is a nice command line tool that complements Sensu. It puts all of the power of the Sensu-API on the command line, for easy interactive use and for integrating Sensu with command line tools.

## Installation

The sensu-cli is not an official Sensu component. However, it is a normal rubygem, and can be install however you like to install rubygems. Of course, I like to install rubygems that are related to Sensu using the omnibus Ruby that Sensu comes with:

/opt/sensu/embedded/bin/gem install sensu-cli

Now the bin stubs for these Sensu-install ruby gems end up in the embedded bin folder here:

ls /opt/sensu/embedded/bin/

But I would like to have this just in my normal path, so I'm going to put this in my path by adding a symlink

ln -s /opt/sensu/embedded/bin/sensu-cli /usr/local/bin/

which sensu-cli

## Background Prep

sed -i "s/localhost/`hostname -f`/" /etc/sensu/conf.d/client.json

/etc/init.d/sensu-client restart

sensu-cli client delete localhost

sensu-cli socket create -n check\_http -o "CRITICAL: 400 bad" -s 2

sensu-cli socket create -n check\_mem -o "WARNING: Out of memory" -s 1

sensu-cli socket raw '{"name": "check\_ssh", "output": "CRIT: ssh is down", "status": 2, "source": "web02"}'

sensu-cli socket raw '{"name": "check\_disk", "output": "CRIT: Disk is full", "status": 2, "source": "sadserver"}'

## Usage

Now let's see what it can do. Out of the box the tool is very human-friendly, outputting color where possible and has some nice ways to visualize the output.

For example, let's see a list of clients. The cli is organized very well:

sensu-cli client list

So sure, you can see the list of clients. This is cool. Now with each subcommand there are some output and filtering options:

sensu-cli client list --help

The table format is a bit interesting:

sensu-cli client list --format table

But I think JSON is the most interesting:

sensu-cli client list --format json

I'll show you a bit more about what you can do with JSON output in a bit. Let's take a look at some more of the features:

sensu-cli --help

There is pretty much a 1 to 1 correspondence with what the sensu-cli can do and what the Sensu-API can do.

* aggregate is the advanced feature I've hinted at before that allows you to execute a subscription check across a set of subscribers
* check allows you to see and issue check requests
* client is what I demonstrated right at first. You can see and delete clients.
* event allows you to list and resolve events, just like on the Sensu dashboard.
* info and health are commands to inspect the healthiness of the Sensu infrastructure.
* silence allows you to silence hosts or checks, which is handy to do from the command line
* stash allows you to add arbitrary stashes in Sensu's key-value store. silence is just a specific type of stash.
* resolve does the same thing as it does on the dashboard: it makes the failing event go away.
* socket is an interesting one. I have a later lecture demonstrating how you can push your own events to the local socket. The sensu-cli provides a convenient way to do this.

## Doing more interesting things

Lets say we wanted to take advantage of this command line tool and make it so we put the list of currently failing checks in the message of the day of the server, so when you log in you get some immediate situational awareness of what is wrong with it.

Let's see if we can do it. We'll start with:

sensu-cli event list

This is a good start, but we want to filter only the events for this local host? Luckily we can do that:

sensu-cli event list --filter name,`hostname -f `

This is good, but it is a little verbose for being in the message of the day. The table format is a little better:

sensu-cli event list --filter name,`hostname -f ` --format=table

But if we are going to do anything really fancy, we are going to have to pull out exactly the fields we want. Specifically I'm kinda only interested in "what" is failing and what the output is. This is where the JSON output can come in hand:

sensu-cli event list --filter name,`hostname -f ` --format=json

But what are we going to use to extract the fields we need? Well, you could certainly write a program to do it, but I'm going to use one of my favorite unixy tools: jq. You have seen me use jq a few times before in previous lectures. Here I'm going to use it to print out just the check name and output:

apt-get -y install jq

sensu-cli event list --filter name,`hostname -f ` --format=json | jq -r '.[].check | .name + ":|" + .output'

And then one more filter I'll apply is to pipe it through the column tool to align it:

sensu-cli event list --filter name,`hostname -f ` --format=json | jq -r '.[].check | .name + ":|" + .output' | column -t -s "|"

Any more complicated than this then I would want to put it into a script or something. But it is nice how much we can do just the command line with pipes and unixy tools.

You could stick this in your bash\_profile or motd and see right away what alerts are failing for the host before you start investigating. Pretty cool.

## Having a Server Silence Itself

Let's say you would like to use the sensu-cli not just for reporting purposes, but also to actually make your infrastructure interact with your monitoring system.

Specifically, let's say that you want your servers to silence themselves for a few minutes after they do a reboot. Or maybe you want your provisioning system to silence newly provisioned servers for a bit. How could you use the sensu-cli tool to do this? Well, it it can be as simple as a single command:

sensu-cli silence -h

The first argument is the hostname itself

sensu-cli silence `hostname -f`

sensu-cli stash list

If you are going to silence the whole machine, then we won't provide a check name.

The owner might be the user, like root. Or if you know who owns the box from other metadata, you could insert it here. For the reason argument you could say "for a reboot" or "freshly provisioned". I like adding the expire argument so that if something goes wrong, it will automatically be un-silenced eventually. I don't think anything should be silenced indefinitely.

sensu-cli silence `hostname -f` --owner root --reason "This server was just created" --expire 3600

sensu-cli stash list

Likewise the server could "un-silence" itself, maybe after a chef run or something like that.

## Silencing Clients

Let's do something else. Let's start by just getting a list of clients that Sensu knows about.

sensu-cli client list

This is a good start, but I really want just the raw hostnames. To do that I'm going to use jq again:

sensu-cli client list -f json | jq -r .[].name

Now that we have the raw names, we can pass them onto another tool. Let's say it was an emergency and you needed to silence them all. You could use on of my other favorite tools, xargs:

sensu-cli client list -f json | jq -r .[].name | xargs --verbose -n1 --no-run-if-empty sensu-cli silence

So here we are taking every sensu client, and xargs will turn that and execute the sensu-cli silence command. The n1 indicates that we want xargs to execute one sensu-cli command per argument. I like the --verbose flag so it will print out exactly what xargs is running. Let's see what happens...

Of course with this you could easily just use grep and filter only the clients you are interested.

## Emitting Alerts

Another interesting things you can do with the Sensu-cli is emitting your own events.

sensu-cli socket create -h

This function allows you to create a Sensu event, without having to define the check in the first place. This is an advanced feature. I have a dedicated lecture just for explaining what this is and why you would want to use it.

## Resolving Alerts

The sensu-cli tool can also help with manually resolving alerts.

sensu-cli resolve --help

Why would you want to do this? This can be handy if you are in a dynamic environment, and checks and disappear as well as appear at will. The resolve command can help "clean up" any residual checks. For example, let's say you were running a hosting company and had a check for every customer that you had, and when a customer leaves, you would want to resolve any lingering events that might have been open, so they don't clutter up the dashboard.

sensu-cli socket create -n customer1 --output "Customer1 is DOWN" -s 2

sensu-cli event list -f table

And then lets say customer1 left or was terminated, you could use the sensu-cli tool to resolve that check manually:

sensu-cli resolve `hostname -f` customer1

sensu-cli event list -f table

Certainly your customer provisioning tool could interact with the Sensu api directly, but not everything has to be that fancy, if you just have some script to provision new customers, webservers, clusters, whatever, you can easily integrate the sensu-cli with your command line tools.

## Conclusion

The sensu-cli is a powerful tool, with a 1 to 1 mapping against the Sensu API. You can quickly integrate your existing scripts and tools with it. And if you get really fancy, you can combine it with things like jq and do some pretty crazy things. Check out the show notes for some even more complicated examples of using this cli tool that I'm too embarrassed to admit that I've used them in production.

## All Examples

### Have a host silence itself

sensu-cli silence `hostname -f` --owner root --reason "This server was just created" --expire 3600

### Silence any client that has the word "test" in the name

sensu-cli client list -f json |

jq -r .[].name |

grep "test" |

xargs --verbose --no-run-if-empty -n1 sensu-cli silence

### Delete sliences older than 3 days

THRESHOLD=$(date +%s --date="3 days ago")

sensu-cli stash list --format json |

jq -r "map(select( .[\"content\"][\"timestamp\"] < $THRESHOLD )) | .[].path " |

xargs --verbose --no-run-if-empty -n1 sensu-cli stash delete

### Purge any checks that haven't checked in in a month

THRESHOLD=$(date +%s --date="1 month ago")

sensu-cli event list --format json |

jq --raw-output "map(select( .[\"check\"][\"issued\"] < $THRESHOLD )) | .[] | .client.name + \" \" + .check.name " |

xargs --verbose --no-run-if-empty -n2 sensu-cli resolve

# **Pushing Your Own External Check Results**

Pushing your own external check results to the Sensu client is and advanced topic, that I was going to reserve for a future course, but I figured I would at least give a mini-lecture on how to use this feature.

The official Sensu documentation does a pretty good job at explaining what this does, and gives and example:

https://sensuapp.org/docs/latest/clients#client-socket-input

But I think it could use some elaboration.

## What are these checks and where do they come from?

Let's review how normal checks actually end up processed by the Sensu infrastructure. Remember that Sensu checks can be either subscription based, which means the Sensu-server schedules them, or they can be standalone, where the Sensu-client schedules them. Regardless, the actual check is executed by the Sensu-client, and then the Sensu-client takes the result of the check, call it "event data", and then puts it on the queue, and then the Sensu server picks up on it for processeing. From there the Sensu server might invoke some handlers, send you an email, whatever.

Now, imagine if there was some way to just "inject" event data like that, but without needing the sensu-server or sensu-client to actually execute the check. That is what this external result input is. It is where something else does the actual check execution, and just relays the resulting event data. Think of it as a kind of bypass, where the Sensu client gives you away to deposit your own custom result data onto the queue, even if that event data isn't even "from" the local Sensu client.

## Where would this be useful?

Where would this be useful? It is useful in situations where you have something you want to check that you won't want to have to pre-configure the Sensu-client to be aware of it. Let's look at one of my favorite use cases for this: Cron jobs.

Let's say we have a cron job in place that we want to monitor, like this

crontab -l

0 0 \* \* \* send-nightly-customer-report

So we have this nightly customer report, and I would like to know if it doesn't work. Sure, you could check your inbox for cron emails. Or maybe you could edit your script to touch a file or something, and then configure Sensu to watch the modification date of that file, but I have a better idea.

We can use the fact that Sensu allows us to push arbitary event data onto the queue, by pushing our own event data for this report. And in that event data, we'll push a status code of 0, meaning "OK", when it works, and a "2", for CRITICAL, when it doesn't work. But that means we need to adjust are script or something to make it talk to RabbitMQ right? Wrong.

First, we don't have to talk to RabbitMQ or anything like that. We can talk directly to the local socket, which is listening on localhost port 3030. It isn't even HTTP, it is just a tcp socket that accepts plain text JSON. This interface is so simple, you could write a bash script to use it.

So I did:

https://github.com/solarkennedy/sensu-shell-helper/

This is a bash script that I use, to help make it even easier to send these events to the local socket. All you have to do is prepend your command with sensu-shell-helper, and you will get an event after your script runs.

Let's try it out and see how it works behind the scense. First I'll download it:

wget -O /usr/local/bin/sensu-shell-helper https://github.com/solarkennedy/sensu-shell-helper/raw/master/sensu-shell-helper

chmod +x /usr/local/bin/sensu-shell-helper

And now let's make up our fake report

vim /usr/local/bin/nightly-customer-report

#!/bin/bash

echo "Sales are great! Everything is fine!"

exit 0

chmod +x /usr/local/bin/nightly-customer-report

Now we can prepend our command with sensu-shell-helper:

sensu-shell-helper nightly-customer-report

That's it! If the nightly-customer-report script starts failing, we will get a sensu alert. What would that alert look like? We'll let's simulate a failure with our script

vim /usr/local/bin/nightly-customer-report

echo "But the script didn't work for some reason!!!"

exit 1

Now let's run the command, with the -d option for a dry-run:

sensu-shell-helper -d nightly-customer-report

Now you can see the actual JSON event data that would have been sent to Sensu. It has the name of this check: nightly-customer-report, and it even gives us the output, and the status code is 2, meaning it is CRITICAL. By default the sensu-shell-helper interprets any non-zero return code from the script as a critical failure.

Let's run this thing and see what it looks like using the sensu-cli tool:

sensu-shell-helper nightly-customer-report

sensu-cli event list

Now you can see that the event is there. Now let's "fix the script" and do it again, simulating what it would be like when cron runs it the next time:

vim /usr/local/bin/nightly-customer-report

dd

exit 0

And now let's run it again...

sensu-shell-helper nightly-customer-report

sensu-cli event list

Now you can see the event is gone, and we would have gotten a resolve email, saying that everything worked ok this time around.

So the sensu-shell-helper is just one example of a quick tool that takes advantage of this local-socket feature, and allows you to make Sensu alerts for any command line invocation of anything, not just cron jobs of course.

But you see, I didn't define any checks on the Sensu client, or on the sensu-server. No daemons were reloaded, no check config files were put on disk. The check for the nightly-customer-report came and went freely, the Sensu server didn't care that it came from the localhost socket and not from the Sensu-client itself. It processes the event just the same.

## Another Example: External Devices

Once you can send arbitrary events, you can do even more interesting things. This is extra useful when the things that you are checking are very dynamic.

Let's build off our customer example, but this time let's say we want to send an event on a per-client basis. And this time, I don't actually want to use the shell-helper, I'm going to use the standard sensu-cli command.

First let's setup the customer report script:

#!/bin/bash

#

# Iterates through each client and will alert or resolve based on

# whether action is required.

#

function check\_customer {

customer=$1

standing=$2

# If a customer is in bad standing, send an alert so someone

# can look at it and try to fix it

if [[ $standing == "good" ]]; then

sensu-cli socket create --name "$customer" --output "Ok: Customer $customer is fine and is in in $standing standing" --status 0

else

sensu-cli socket create --name "$customer" --output "Critical: Customer $customer is in $standing standing. Take action!" --status 2

fi

}

# Iterate over all the customers

check\_customer "customerA" "good"

check\_customer "customerB" "bad"

check\_customer "customerC" "good"

Obviously this silly little bash script just serves as example. Hopefully real people are not writing their customer reports in bash.

But you can see that this script iterates over customers and our check\_customer function will send an event for that customer. It is important that we send and event when something is wrong, but we also want to send a "good" event when everything is ok, so we get that "resolve" email when things are fixed.

Let's run this thing and see what happens:

nightly-customer-report

If that worked, then it would have send an "ok" event for customers A and C, and a "critical" event for customer B. Let's see what events are now out there:

sensu-cli event list

And now let's put customerB in good sanding and see if the event goes away:

vim /usr/local/bin/nightly-customer-report

/bad/good/

nightly-customer-report

And now do we have any events?

sensu-cli event list

And it resolved.

## Conclusion

Obviously this is just an example, hopefully no one is writing customer reports in bash. But it just serves as an example of how you can use this external event data feature of sensu to push your own events, for things that come and go like cron jobs and customers, without having to "let sensu know" beforehand that it exists.

Before we conclude I would like to give a couple more examples of how this feature can be used.

https://github.com/solarkennedy/sensu-shell-helper/

We saw how my sensu-shell-helper can be used to monitor the output of any command line invocation, like in a cron job.

In this check-serverspec.rb example:

https://github.com/sensu/sensu-community-plugins/blob/master/plugins/serverspec/check-serverspec.rb

The script iterates over a bunch of serverspec tests, and then emits and event for each test, whether it was a pass or a fail.

In this example:

http://gist.leavesongs.com/countryHick/26a3dd2824b86dd5f994

Someone has written a script that loops over some special SNMP traps and emits custom events based on teh name of the trap.

In this example:

https://gist.github.com/joemiller/5806570

The script inspects the Pantheon API, enumerates over all the endpoints, and pings each one, and emits and event for eachendpoint.

I hope that better illuminates what this feature can do for you. It is certainly an advanced feature. You can get by with normal Sensu checks for a long time before you encounter a case where you need this kind of custom event creation.

But when you do have this tool in your toolbox, it means that when you encounter such a situation, where you don't know beforehand what it is that you need to monitor exactly, and you need to have a custom event based on what is out there, this feature can be a very powerful tool, and can help you monitor things that are inherently dynamic.