# 2@keystone命令与client接口学习 - hustlijian - 博客园

http://www.cnblogs.com/hustlijian/p/3603992.html

**keystone学习**

**------------------**

Keystone（OpenStack Identity Service）是OpenStack框架中，负责身份验证、服务规则和服务令牌的功能， 它实现了OpenStack的IdentityAPI。Keystone类似一个服务总线， 或者说是整个Openstack框架的注册表， 其他服务通过keystone来注册其服务的Endpoint（服务访问的URL），任何服务之间相互的调用， 需要经过Keystone的身份验证， 来获得目标服务的Endpoint来找到目标服务。

Keystone基本概念介绍

        1. User

　　User即用户，他们代表可以通过keystone进行访问的人或程序。Users通过认证信息（credentials，如密码、API Keys等）进行验证。

　　2. Tenant

　　Tenant即租户，它是各个服务中的一些可以访问的资源集合。例如，在Nova中一个tenant可以是一些机器，在Swift和Glance中一个tenant可以是一些镜

像存储，在Quantum中一个tenant可以是一些网络资源。Users默认的总是绑定到某些tenant上。

　　3. Role

　　Role即角色，Roles代表一组用户可以访问的资源权限，例如Nova中的虚拟机、Glance中的镜像。Users可以被添加到任意一个全局的 或 租户内的角色中。

在全局的role中，用户的role权限作用于所有的租户，即可以对所有的租户执行role规定的权限；在租户内的role中，用户仅能在当前租户内执行role规定的权限。

　　4. Service

　　Service即服务，如Nova、Glance、Swift。根据前三个概念（User，Tenant和Role）一个服务可以确认当前用户是否具有访问其资源的权限。但是当一个user尝试着访问其租户内的service时，他必须知道这个service是否存在以及如何访问这个service，这里通常使用一些不同的名称表示不同的服务。在上文中谈到的Role，实际上也是可以绑定到某个service的。例如，当swift需要一个管理员权限的访问进行对象创建时，对于相同的role我们并不一定也需要对nova进行管理员权限的访问。为了实现这个目标，我们应该创建两个独立的管理员role，一个绑定到swift，另一个绑定到nova，从而实现对swift进行管理员权限访问不会影响到Nova或其他服务。

　　5. Endpoint

　　Endpoint，翻译为“端点”，我们可以理解它是一个服务暴露出来的访问点，如果需要访问一个服务，则必须知道他的endpoint。因此，在keystone中包含一个endpoint模板（endpoint template，在安装keystone的时候我们可以在conf文件夹下看到这个文件），这个模板提供了所有存在的服务endpoints信息。一个endpoint template包含一个URLs列表，列表中的每个URL都对应一个服务实例的访问地址，并且具有public、private和admin这三种权限。public url可以被全局访问（如http://compute.example.com），private url只能被局域网访问（如http://compute.example.local），admin url被从常规的访问中分离。

WSGI API（包头）

当成功验证后中间件经为下行的WSGI应用发送如下包头：

X-Identity-Status

提供请求是否被验证的信息。

X-Tenant

提供了租户ID（在Keystone中以URL的形式出现）。在Keysotne转为采用ID/Name模式之前，它为租户提供了对任意遗留实现的支持。

X-Tenant-Id

唯一不变的租户ID。

X-Tenant-Name

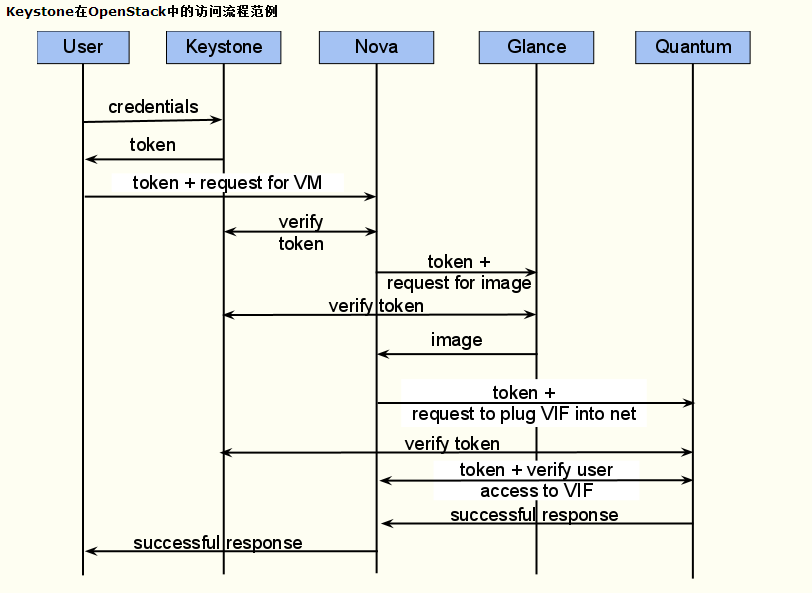
唯一但可变的租户名字。

X-User

用于登录的用户名。

X-Roles

分配给用户的角色。



**命令流程**

**--------------------**

1. 获取token（一个unscoped token，没有 和tenant绑定）

 首先，需要确认你将访问那个tenant，必须使用keystone来获取一个unscoped token（意味着这个token没有和特定的tenant绑定），这个unscoped token能够用来深入查询keystone service，确定你能访问哪些tenants。获取一个unscoped token，使用典型的REST API，在request的body中不指定tenantName。

2. 获取tenants

接下来的一步是，使用unscoped token来获取能访问的tenants，其中租期已经由你分配的角色决定了，对于每个tenant，都有一个确定的角色。所有在service endpoint上执行的操作都需要一个scoped token。获取能访问的tenants，使用 GET /tenants keystone API，其中将unscoped token写入X-Auth-Token。得到一个tenants数组，包含了能够访问的tenants。

3. 获取scoped tokens

获取了能够访问的tenants之后，决定访问某个tenants，就开始需要获取一个scoped token，这个scoped token与某个特定的tenant绑定，能够提供这个tenant的metadata和在tenant中的角色。获取scoped token需要使用POST /tokens keystone API，像第一步一样，这有两种形式的API。

更重要的是，其中包含了一组service endpoints。这些endpoints 确定了获取的token能够访问的服务，Keystone service manage都是基于service/endpoing catalog的.通过这些endpoings，决定访问其中的service。

4. 使用scoped tokens

现在已经获取了scoped tokens，并且知道了endpoint API的url，下一步就是调用这些service endpoint。在这一步，使用keystone来确证token的有效性。存在两种类型的token，一种基于UUID的，一种基于PKI（Public Key Infrastructure）。

5. 验证role metadata

Endpoint service 使用token的metadata来验证用户能够访问请求的服务。这一般都涉及到Role Base Access Control(RBAC)。基于服务的policy.json文件，使用rule engine来决定用户的token包含适当的角色访问。

6. 请求服务

到此，user就能够去通过api访问有权限访问的资源。

一个unscoped tokens例子：



一个scoped tokens例子：







## client SDK使用

**------------------**

1. 使用

>>> from keystoneclient.v2\_0 import client

>>> keystone = client.Client(...)

>>> keystone.tenants.list() # List tenants

2. 获得权限

>>> from keystoneclient.v2\_0 import client

>>> username='adminUser'

>>> password='secreetword'

>>> tenant\_name='openstackDemo'

>>> auth\_url='http://192.168.206.130:5000/v2.0'

>>> keystone = client.Client(username=username, password=password,

...                          tenant\_name=tenant\_name, auth\_url=auth\_url)

3. 建立租户

>>> from keystoneclient.v2\_0 import client

>>> keystone = client.Client(...)

>>> keystone.tenants.create(tenant\_name="openstackDemo",

...                         description="Default Tenant", enabled=True)

4. 建立用户

>>> from keystoneclient.v2\_0 import client

>>> keystone = client.Client(...)

>>> tenants = keystone.tenants.list()

>>> my\_tenant = [x for x in tenants if x.name=='openstackDemo'][0]

>>> my\_user = keystone.users.create(name="adminUser",

...                                 password="secretword",

...                                 tenant\_id=my\_tenant.id)

5. 建立角色和增加用户

>>> from keystoneclient.v2\_0 import client

>>> keystone = client.Client(...)

>>> role = keystone.roles.create('admin')

>>> my\_tenant = ...

>>> my\_user = ...

>>> keystone.roles.add\_user\_role(my\_user, role, my\_tenant)

6. 建立服务和endpoints

>>> from keystoneclient.v2\_0 import client

>>> keystone = client.Client(...)

>>> service = keystone.services.create(name="nova", service\_type="compute",

...                                    description="Nova Compute Service")

>>> keystone.endpoints.create(

...     region="RegionOne", service\_id=service.id,

...     publicurl="http://192.168.206.130:8774/v2/%(tenant\_id)s",

...     adminurl="http://192.168.206.130:8774/v2/%(tenant\_id)s",

...     internalurl="http://192.168.206.130:8774/v2/%(tenant\_id)s")

**参考**

**----------------**

1. keystone命令汇总：<http://wiki.l-cloud.org/index.php/Keystone_%E5%91%BD%E4%BB%A4%E6%B1%87%E6%80%BB>

2. openstack keystone流程分析：<http://www.cnblogs.com/liuan/p/3194499.html>

3. using the client api: <http://docs.openstack.org/developer/python-keystoneclient/using-api.html>

4. the client v2 api: <http://docs.openstack.org/developer/python-keystoneclient/using-api-v2.html>

## python

from keystoneauth1.identity import v2

from keystoneauth1 import session

from keystoneclient.v2\_0 import client

token = 'OR\_KhVn9QLC2OYm5u2W5qQ'

endpoint = 'https://10.0.1.11:5000/v2.0'

auth = v2.Token(auth\_url=endpoint, token=token)

sess = session.Session(auth=auth)

keystone = client.Client(session=sess)

from keystoneclient.v2\_0 import client

keystone = client.Client(username='liyang', password='123456',auth\_url='https://10.0.1.11:5000/v2.0')

keystone = client.Client(username='liyang', password='123456',tenant\_name='paas', auth\_url='https://10.0.1.11:5000/v2.0',insecure='true')

## python-keystoneclient pypi

pypi

Index of Packages : Python Package Index

https://pypi.python.org/pypi/python-keystoneclient

# 2@openstack获取用户token（及endpoints）

- JYao - 博客频道 - CSDN.NET

http://blog.csdn.net/jiajiastudy/article/details/9011407

对于[**OpenStack**](http://lib.csdn.net/base/openstack)的api操作来说，大量的命令都依赖相关用户的token来完成，尤其对自动化[**测试**](http://lib.csdn.net/base/softwaretest)来说，可以说拿到了用户的token就相当于取得了进入openstack这个大工厂大门的钥匙，有了这个钥匙，才能进入这个工厂大显身手。

        要想拿到token, 必须知道用户的相关信息，其中用户名和密码是必须的，如果还想取得更多的信息，例如用户对各种服务包括glance, keystond的访问endpoint, 还需要提供用户的tenant信息。实际上，对于终端用户来说，因为用户名，密码以及tenant名更为直观，所以很少会直接用token进行操作，但对于自动化测试来说，因为要直接和相关api打交道，取得token就相当有必要了。

        命令行取得用户token的命令为：

        # curl -X POST http://localhost:5000/v2.0/tokens -d '{"auth":{"passwordCredentials":{"username": "username", "password":"password"}}}' -H "Content-type: application/json"

        其中localhost:5000是openstack keystone服务的endpoint, 如果没有特殊的设置，5000就是keystone服务进程的端口号。

        /v2.0/token 是openstack api里定义的取得token的URI， 请求方式为POST，这个可以从openstack.org里查到。

       后面json结构的数据‘auth’是提供给keystone服务用户信息，包括用户名和密码。下面看一下输出：

       {"access": {"token": {"expires": "2013-06-04T03:06:23Z", "id": "5fcf748e0d5d4a02ae3465e0dd301f40"}, "serviceCatalog": {}, "user": {"username": "username", "roles\_links": [], "id": "ce205b61760c463cb46e41909de8495f", "roles": [], "name": "username"}}}

       这是openstack/essex版本下的token输出，其中['token']['id']就是我们得到的用户token。对于openstack/grizzly版本, 用户的token比这个要长得多，但基本结构是一样的。

       下面看一下使用tenant的情况：

       curl -X POST http://localhost:5000/v2.0/tokens -d '{"auth":{"passwordCredentials":{"username": "admin", "password":"crowbar"}, "tenantName":"tenantname"}}' -H "Content-type: application/json"

       输出：

       {"access": {"token": {"expires": "2013-06-04T03:14:12Z", "id": "fc3e38a93e95462da5028b1fb3a688c0", "tenant": {"description": "description", "enabled": true, "id": "4e14ab2a2df045f1a6f02081a46deb2c", "name": "tenantname"}}, "serviceCatalog": [{"endpoints": [{"adminURL": "http://localhost:8776/v1/4e14ab2a2df045f1a6f02081a46deb2c", "region": "RegionOne", "internalURL": "http://localhost:8776/v1/4e14ab2a2df045f1a6f02081a46deb2c", "publicURL": "http://localhost:8776/v1/4e14ab2a2df045f1a6f02081a46deb2c"}], "endpoints\_links": [], "type": "volume", "name": "nova-volume"}, {"endpoints": [{"adminURL": "http://localhost:9292/v1", "region": "RegionOne", "internalURL": "http://localhost:9292/v1", "publicURL": "http://localhost:9292/v1"}], "endpoints\_links": [], "type": "image", "name": "glance"}, {"endpoints": [{"adminURL": "http://localhost:8774/v2/4e14ab2a2df045f1a6f02081a46deb2c", "region": "RegionOne", "internalURL": "http://localhost:8774/v2/4e14ab2a2df045f1a6f02081a46deb2c", "publicURL": "http://localhost:8774/v2/4e14ab2a2df045f1a6f02081a46deb2c"}], "endpoints\_links": [], "type": "compute", "name": "nova"}, {"endpoints": [{"adminURL": "http://localhost:8773/services/Admin", "region": "RegionOne", "internalURL": "http://localhost:8773/services/Cloud", "publicURL": "http://localhost:8773/services/Cloud"}], "endpoints\_links": [], "type": "ec2", "name": "ec2"}, {"endpoints": [{"adminURL": "http://localhost:35357/v2.0", "region": "RegionOne", "internalURL": "http://localhost:5000/v2.0", "publicURL": "http://localhost:5000/v2.0"}], "endpoints\_links": [], "type": "identity", "name": "keystone"}], "user": {"username": "admin", "roles\_links": [], "id": "ce205b61760c463cb46e41909de8495f", "roles": [{"id": "454cb6cbddaf41f2af6f87e68ce58d64", "name": "KeystoneAdmin"}, {"id": "5a80a5b5d4244f48ac7d3079d56555c6", "name": "KeystoneServiceAdmin"}, {"id": "c5a190185ea7434eb2c35bbd1bb52051", "name": "username"}], "name": "tenentname"}}}

        可以看到，如果在请求token的时候同时提供了tenant信息，则可以额外获取用户相关的endpoints信息。这样，有了token和相关endpoints, 就能够对openstack的api进行相关的访问和操作了。顺便说明，上述提供的tenant的信息也可以是tenant的id， 格式为"tenantId":"<tenantID>“。

        用户每次发出一次请求，就会生成一个token, 同时会在glance[**数据库**](http://lib.csdn.net/base/mysql)的token表内生成一个记录。每个token的有效时间缺省为24个小时。

        编程实现取得用户token也极为简单，代码如下：

import httplib2  
import json  
  
http\_obj = httplib2.Http()  
headers = {}  
body = {  
    "auth": {  
            "passwordCredentials":{  
                "username": 'username',  
                "password": 'password',  
            },  
            "tenantName": 'tenantname',  
        },  
    }  
  
req\_url = "http://localhost:5000/v2.0/tokens"  
method = "POST"  
  
headers['Content-Type'] = 'application/json'  
headers['Accept'] = 'application/json'  
  
resp, token = http\_obj.request(req\_url, method,  
                               headers=headers, body=json.dumps(body))  
  
print resp  
print token  
  
如果在body里不提供tenantName或tenantId的数据，则返回的是上述command line命令不包括endpoints的输出。

## 登陆

curl -vk -XPOST -H "Content-Type: application/json" https://10.0.1.11:5000/v2.0/tokens -d'{"auth":{"passwordCredentials":{"username": "liyang", "password":"123456"}}}'|python -m json.tool



# Monitoring an Openstack deployment with Prometheus and Grafana

| Service Engineering (ICCLab & SPLab)

https://blog.zhaw.ch/icclab/monitoring-an-openstack-deployment-with-prometheus-and-grafana/

Monitoring an Openstack deployment with Prometheus and Grafana

Posted on [**24. November 2016**](https://blog.zhaw.ch/icclab/monitoring-an-openstack-deployment-with-prometheus-and-grafana/) by [**Bruno Grazioli**](https://blog.zhaw.ch/icclab/author/gaea/)

Following [our previous blog post](https://blog.zhaw.ch/icclab/installing-monasca-a-happy-ending-after-much-sadness-and-woe/), we are still looking at tools for collecting metrics from an Openstack deployment in order to understand its resource utilization. Although Monasca has a comprehensive set of metrics and alarm definitions, the complex installation process combined with a lack of documentation makes it a frustrating experience to get it up and running. Further, although it is complex, with many moving parts, it was difficult to configure it to obtain the analysis we wanted from the raw data, viz how many of our servers are overloaded over different timescales in different respects (cpu, memory, disk io, network io). For these reasons we decided to try Prometheus with Grafana which turned out to be much easier to install and configure (taking less than an hour to set up!). This blog post covers the installation process and configuration of [Prometheus](https://prometheus.io/) and[Grafana](http://grafana.org/) in a Docker container and how to install and configure [Canonical’s Prometheus Openstack exporter](https://github.com/CanonicalLtd/prometheus-openstack-exporter) to collect a small set of metrics related to an Openstack deployment.

*Note that minor changes to this HOWTO are required to install theses services in a VM or in a host machine when using containers is not an option. As preparation, take note of your Openstack deployment’s locations for Keystone and the Docker host. Remember that all downloads should be verified by signature comparison for production use.*

**Installing and configuring Prometheus**

First of all pull the Ubuntu image into you docker machine. Let’s call it ***docker-host.***

*Note that in this blog post we describe Prometheus installation process step-by-step – we chose to install it from scratch to get a better understanding of the system, but using the pre-canned*[*Docker Hub*](https://hub.docker.com/r/prom/prometheus/)*image is also possible.*

docker pull ubuntu:14.04

Then create the docker container opening the port 9090 which will be used to get/push metrics into Prometheus.

docker run -it -p 9090:9090 --name prometheus ubuntu:14.04

Inside the container [download](https://prometheus.io/download/) the latest version of Prometheus and uncompress it (version 1.3.1 is used in this HOWTO; the download size is ca. 16 MB).

wget https://github.com/prometheus/prometheus/releases/download/v1.3.1/prometheus-1.3.1.linux-amd64.tar.gz

tar xvf prometheus-1.3.1.linux-amd64.tar.gz

cd prometheus-1.3.1.linux-amd64

Configure *prometheus.yml* adding the targets from which prometheus should scrape metrics. See the example below for the Openstack exporter (assuming it is installed in the same ***docker-host***):

scrape\_configs:

 - job\_name: 'openstack-deployment-1'

   scrape\_interval: 5m

   Static\_configs:

     - targets: ['***docker-host***:9183']

Start the Prometheus service:

./prometheus -config.file=prometheus.yml

Similarly, install and configure the [Prometheus Openstack exporter](https://github.com/CanonicalLtd/prometheus-openstack-exporter) [i](https://github.com/CanonicalLtd/prometheus-openstack-exporter)n another container. Note that this container needs to be set up manually as there are configuration files to be changed and Openstack libraries to be installed.

docker run -it -p 9183:9183 --name prometheus-openstack-exporter ubuntu:14.04

sudo apt-get install python-neutronclient python-novaclient python-keystoneclient python-netaddr unzip wget python-pip python-dev python-yaml

pip install prometheus\_client

wget https://github.com/CanonicalLtd/prometheus-openstack-exporter/archive/master.zip

unzip master.zip

cd prometheus-openstack-exporter-master/

Next, configure *prometheus-openstack-exporter.yaml* create the */var/cache/prometheus-openstack-exporter/*directory and the novarc file containing credentials for Nova user.

mkdir /var/cache/prometheus-openstack-exporter/

echo  export OS\_USERNAME=**nova-username** \

     export OS\_PASSWORD=**nova-password** \

export OS\_AUTH\_URL=**http://keystone-url:5000/v2.0** \

export OS\_REGION\_NAME=**RegionOne** \

export OS\_TENANT\_NAME=services > novarc

source novarc

./prometheus-openstack-exporter prometheus-openstack-exporter.yaml

Then you’ve got a fully functional Prometheus system with some Openstack metrics on it! Visit **http://docker-host:9090** to graph and see which metrics are available.

Here is the list of the 18 metrics currently collected by [Prometheus Openstack exporter](https://github.com/CanonicalLtd/prometheus-openstack-exporter):

|  |  |
| --- | --- |
| neutron\_public\_ip\_usage | hypervisor\_memory\_mbs\_total |
| neutron\_net\_size | hypervisor\_running\_vms |
| hypervisor\_memory\_mbs\_used | hypervisor\_disk\_gbs\_total |
| hypervisor\_vcpus\_total | hypervisor\_disk\_gbs\_used |
| openstack\_allocation\_ratio | hypervisor\_vcpus\_used |
| nova\_instances | nova\_resources\_ram\_mbs |
| nova\_resources\_disk\_gbs | swift\_replication\_duration\_seconds |
| openstack\_exporter\_cache\_age\_seconds | swift\_disk\_usage\_bytes |
| swift\_replication\_stats | swift\_quarantined\_objects |

Alternatively you could use [Prometheus’s Node exporter](https://github.com/prometheus/node_exporter) for more detailed metrics on node usage – this needs to be installed in the controller/compute nodes and theprometheus.yml configuration file also needs to be changed. A docker container is also available at [Docker Hub](https://hub.docker.com/r/prom/node-exporter/).

Although Prometheus provides some rudimentary graph support, combining it with a more powerful graphing solution makes it much easier to see what’s going on in your system. For this reason, we set up Grafana.

**Installing Grafana**

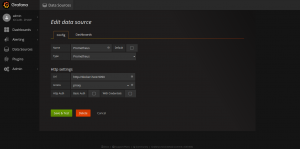
The latest version of Grafana (currently 4.0.0-beta2) had a lot of improvements in its user interface, it also supports now alerting and notifications for every panel available – refer to the [documentation](http://docs.grafana.org/guides/whats-new-in-v4/) for more information. Its integration with Prometheus is very straightforward, as described below.

First of all, pull the grafana image into your **docker-host** and create the docker container opening the port 3000 used to access it.

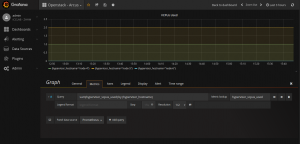
docker pull grafana/grafana

docker run -d -p 3000:3000 grafana/grafana:4.0.0-beta2

Visit **http://docker-host:3000** and use the credentials admin/admin to log into the dashboard. In the Data Sources tab, add a new corresponding data source.

[](https://blog.zhaw.ch/icclab/files/2016/11/Selection_005.png)

Create a new dashboard and add panels containing graphs using the Prometheus datasource.

[](https://blog.zhaw.ch/icclab/files/2016/11/Selection_006.png)

Play around with metrics available and create your own dashboard! See a simple example below.

[](https://blog.zhaw.ch/icclab/files/2016/11/Selection_004.png)

**Conclusion**

Although not many metrics are available yet to monitor an Openstack deployment the combination of Prometheus and Grafana is quite powerful for visualising data; also it was much easier to set up in comparison with Monasca. Further, from a cursory glance, Prometheus seems to be more flexible than Monasca and for these reasons it appears more promising. That said, we are still looking into Prometheus and how it can be used to properly understand resource consumption in an Openstack context, but that will come in another blog post!

# gvauter/prometheus\_openstack: Prometheus exporter for OpenStack

https://github.com/gvauter/prometheus\_openstack

prometheus\_openstack

Prometheus exporter for OpenStack metrics

See [https://prometheus.io](https://prometheus.io/)

Install

pip install git+git://github.com/gvauter/prometheus\_openstack

Usage

openstack\_exporter --config <file> --port <port> --interval <seconds>

Sample config file

openstack:

username: <usename>

password: <password>

tenant: <tenant name>

auth\_url: <http://<ip>:5000/v2.0>

Metrics

Currently only collecting hypervisor metrics from the Nova API

# CanonicalLtd/prometheus-openstack-exporter:

OpenStack exporter for the prometheus monitoring system

https://github.com/CanonicalLtd/prometheus-openstack-exporter

Prometheus OpenStack exporter

Exposes high level [OpenStack](http://www.openstack.org/) metrics to [Prometheus](https://prometheus.io/).

Deployment

Requirements

sudo apt-get install python-neutronclient python-novaclient python-keystoneclient python-netaddr

Install prometheus\_client. On Ubuntu 16.04:

apt-get install python-prometheus-client

On Ubuntu 14.04:

pip install prometheus\_client

Installation

# Copy example config in place, edit to your needs

sudo cp prometheus-openstack-exporter.yaml /etc/prometheus/

## Upstart

# Install job

sudo cp prometheus-openstack-exporter.conf /etc/init

# Configure novarc location:

sudo sh -c 'echo "NOVARC=/path/to/admin-novarc">/etc/default/prometheus-openstack-exporter'

## Systemd

# Install job

sudo cp prometheus-openstack-exporter.service /etc/systemd/system/

# create novarc

sudo cat <<EOF > /etc/prometheus-openstack-exporter/admin.novarc

export OS\_USERNAME=Admin

export OS\_TENANT\_NAME=admin

export OS\_PASSWORD=XXXX

export OS\_REGION\_NAME=cloudname

export OS\_AUTH\_URL=http://XX.XX.XX.XX:35357/v2.0

EOF

# create default config location

sudo sh -c 'echo "CONFIG\_FILE=/etc/prometheus-openstack-exporter/prometheus-openstack-exporter.yaml">/etc/default/prometheus-openstack-exporter'

# Start

sudo start prometheus-openstack-exporter

Or to run interactively:

. /path/to/admin-novarc

./prometheus-openstack-exporter prometheus-openstack-exporter.yaml

Configuration

Configuration options are documented in prometheus-openstack-exporter.yaml shipped with this project

FAQ

Why are openstack\_allocation\_ratio values hardcoded?

There is no way to retrieve them using OpenStack API.

Alternative approach could be to hardcode those values in queries but this approach breaks when allocation ratios change.

Why hardcode swift host list?

Same as above, there is no way to retrieve swift hosts using API.

Why not write dedicated swift exporter?

Swift stats are included mainly because they are trivial to retrieve. If and when standalone swift exporter appears we can revisit this approach

Why cache data?

We are aware that Prometheus best practise is to avoid caching. Unfortunately queries we need to run are very heavy and in bigger clouds can take minutes to execute. This is problematic not only because of delays but also because multiple servers scraping the exporter could have negative impact on the cloud performance

# pip - installing python packages without internet and using source code as .tar.gz and .whl - Stack Overflow

http://stackoverflow.com/questions/36725843/installing-python-packages-without-internet-and-using-source-code-as-tar-gz-and

This is how I handle this case:

On the machine where I have access to Internet:

mkdir keystone-deps

pip install python-keystoneclient --download="/home/aviuser/keystone-deps"

tar cvfz keystone-deps.tgz keystone-deps

Then move the tar file to the destination machine that does not have Internet access and perform the following:

tar xvfz keystone-deps.tgz

cd keystone-deps

pip install python\_keystoneclient-2.3.1-py2.py3-none-any.whl -f ./ --no-index