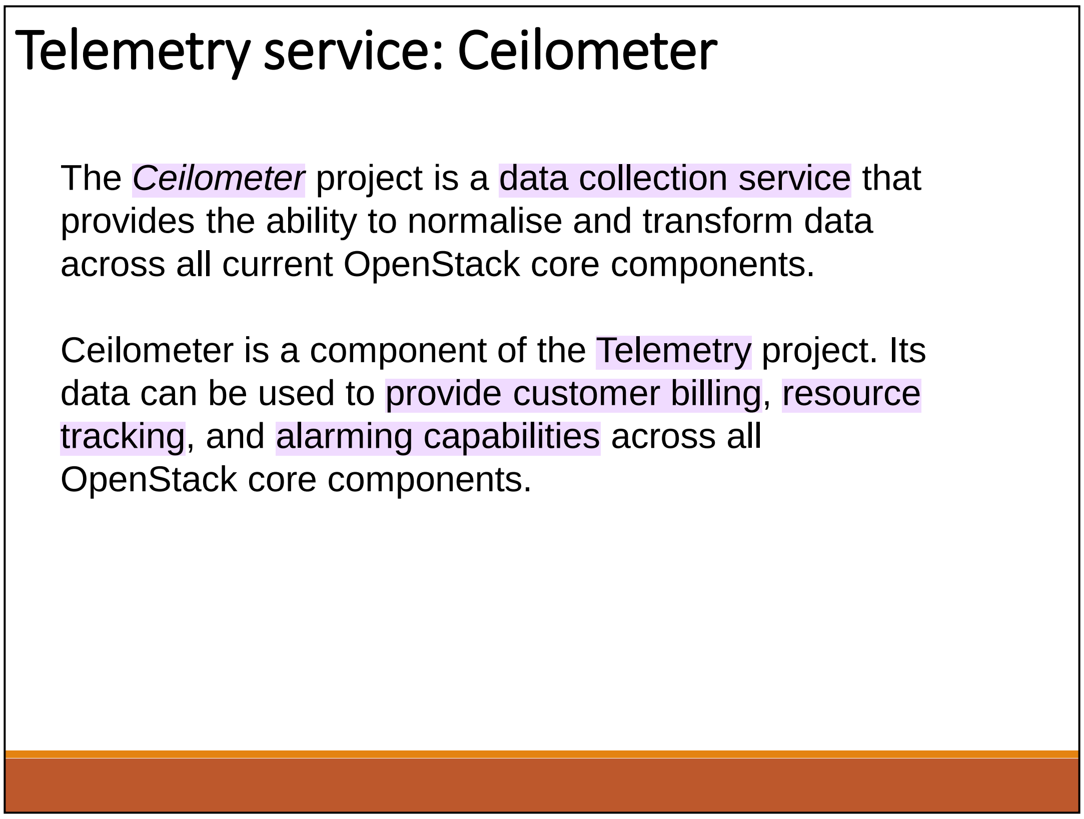
Part 3 – OpenStack Cloud Management and Orchestration

Gianluca Reali

Università degli Studi di Perugia

Ref. www.opestack.org



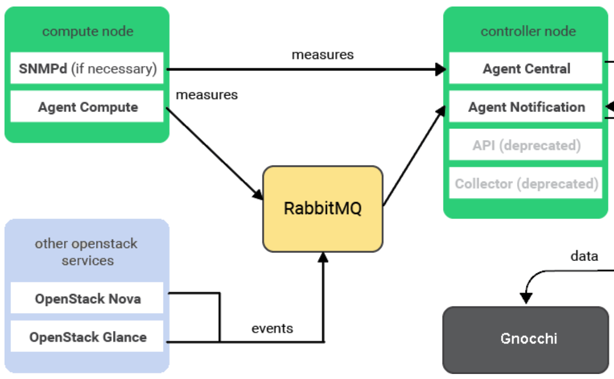
Telemetry service: Ceilometer

The *Ceilometer* project is a data collection service that provides the ability to normalise and transform data across all current OpenStack core components.

Ceilometer is a component of the Telemetry project. Its data can be used to provide customer billing, resource tracking, and alarming capabilities across all OpenStack core components.

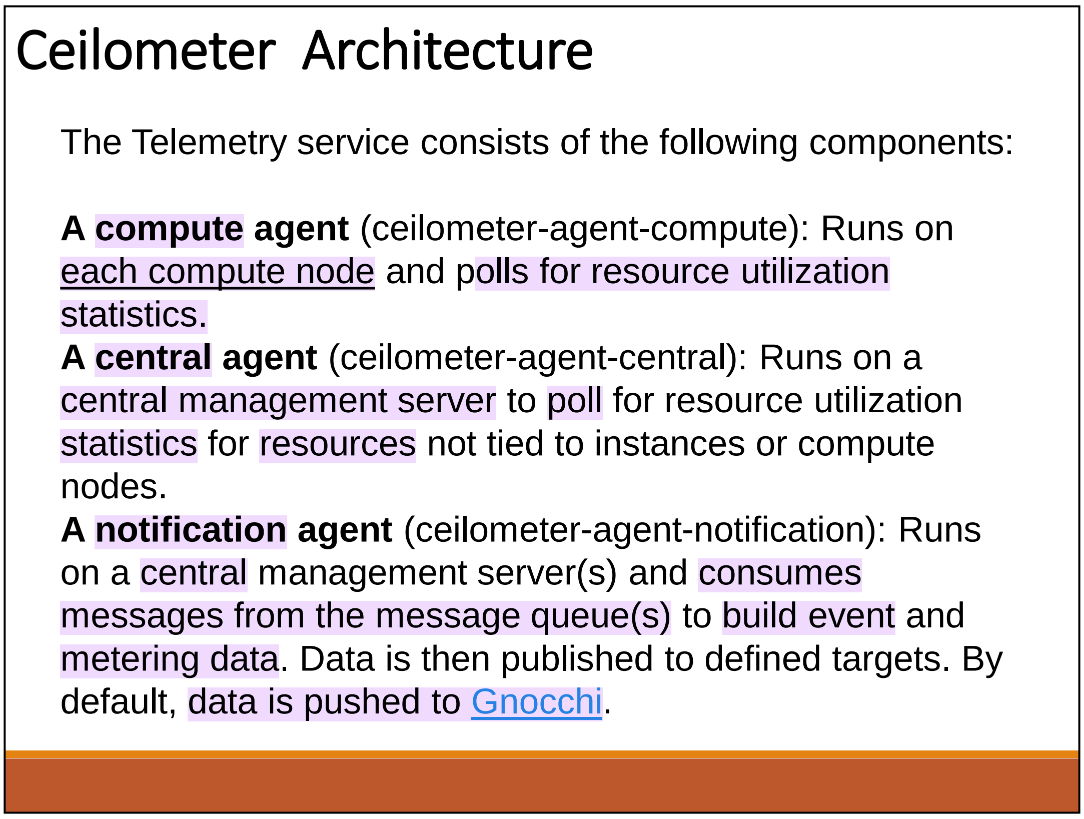
https://docs.openstack.org/ceilometer/latest/

Ceilometer Architecture



These services communicate by using the OpenStack **messaging bus.** Ceilometer data is designed to be published to various endpoints for storage and analysis.
Gnocchi is an open-source time series database, useful for storing and indexing of time series data and resources at a large scale. This is useful in modern cloud platforms which are not only huge but also are dynamic and potentially multi-tenant.

The peculiarity of Gnocchi is that, instead of storing the raw point data as is usually done with time series, it aggregates them before storing them. By doing so, retrieving the data is extremely fast, since no aggregation needs to be done at the time of the queries but the data is ready.



Ceilometer Architecture

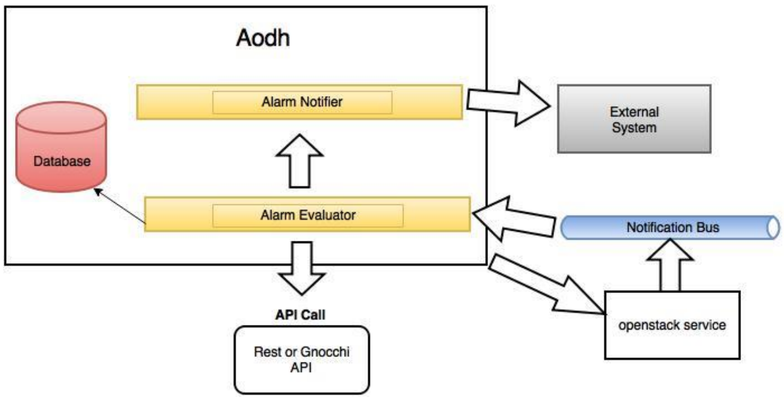
The Telemetry service consists of the following components:

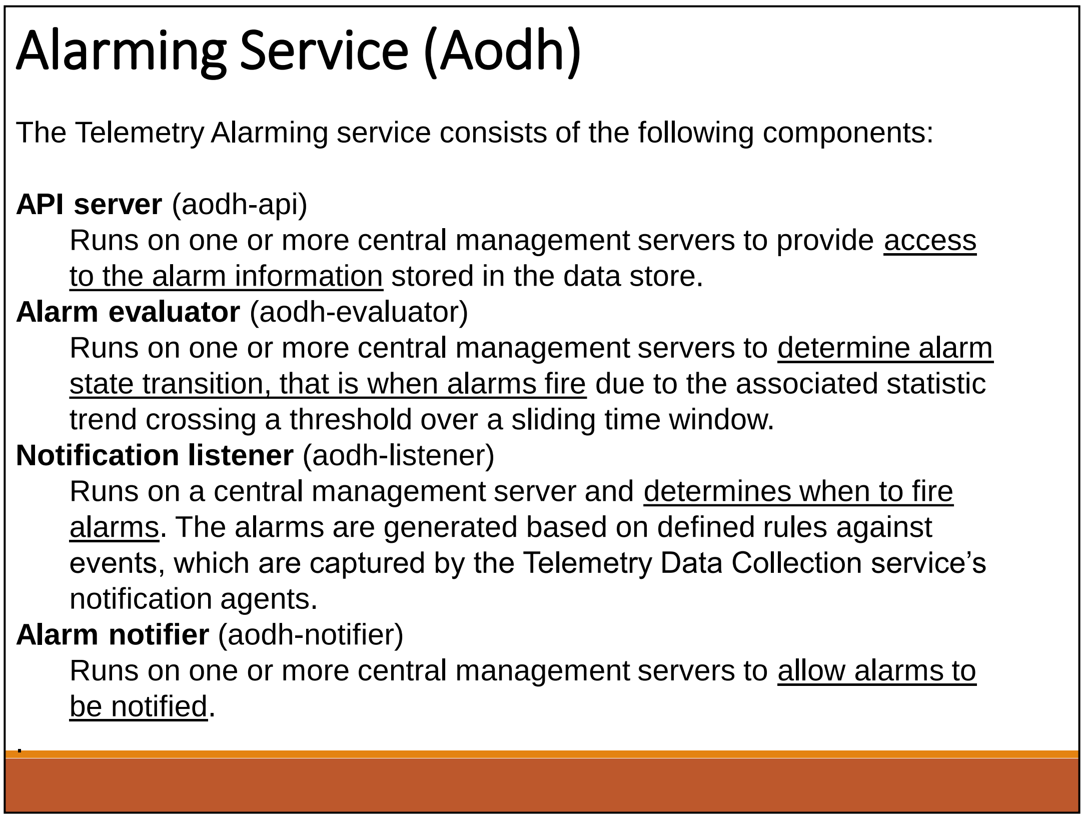
A compute agent (ceilometer-agent-compute): Runs on each compute node and polls for resource utilization statistics.
A central agent (ceilometer-agent-central): Runs on a central management server to poll for resource utilization statistics for resources not tied to instances or compute nodes.
A notification agent (ceilometer-agent-notification): Runs on a central management server(s) and consumes messages from the message queue(s) to build event and metering data. Data is then published to defined targets. By default, data is pushed to Gnocchi.

https://docs.openstack.org/ceilometer/latest/

Alarming Service (Aodh)

The Alarming service (aodh) project provides a service that enables the ability to trigger actions based on defined rules against metric or event data collected by Ceilometer or Gnocchi.





Alarming Service (Aodh)

The Telemetry Alarming service consists of the following components:

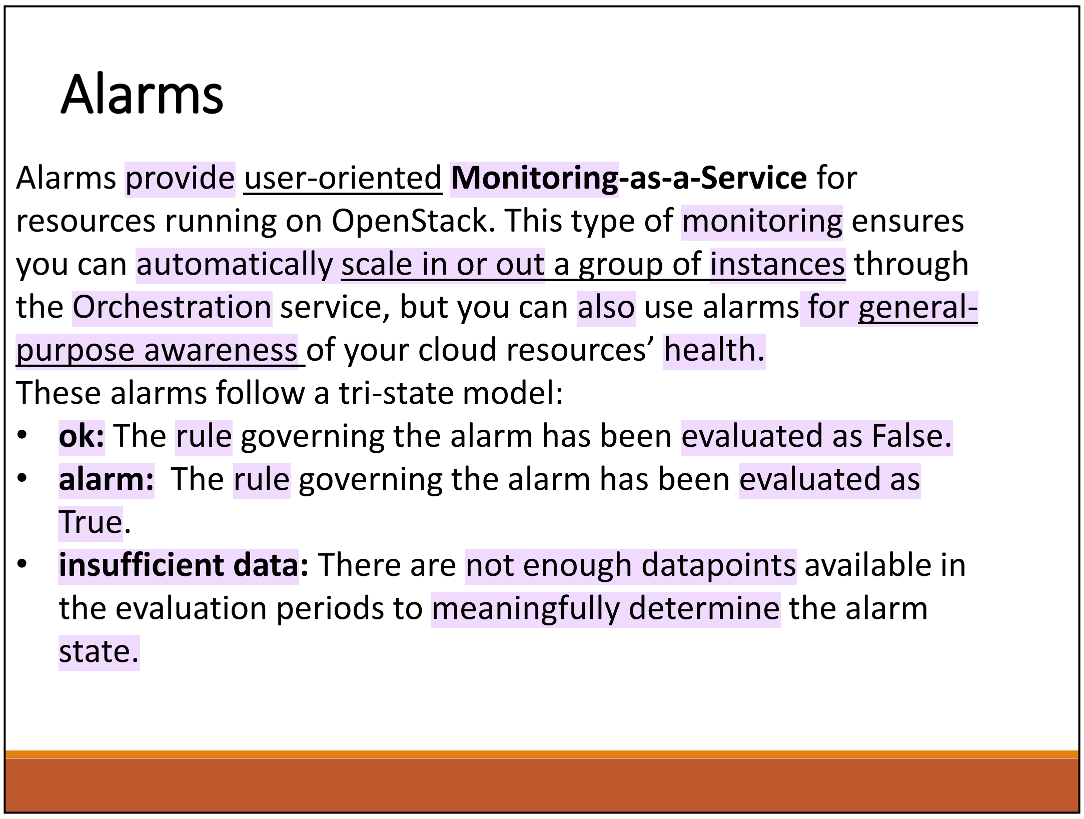
API server (aodh-api)
Runs on one or more central management servers to provide access to the alarm information stored in the data store.

Alarm evaluator (aodh-evaluator)
Runs on one or more central management servers to determine alarm state transition, that is when alarms fire due to the associated statistic trend crossing a threshold over a sliding time window.

Notification listener (aodh-listener)
Runs on a central management server and determines when to fire alarms. The alarms are generated based on defined rules against events, which are captured by the Telemetry Data Collection service's notification agents.

Alarm notifier (aodh-notifier)
Runs on one or more central management servers to allow alarms to be notified.

.

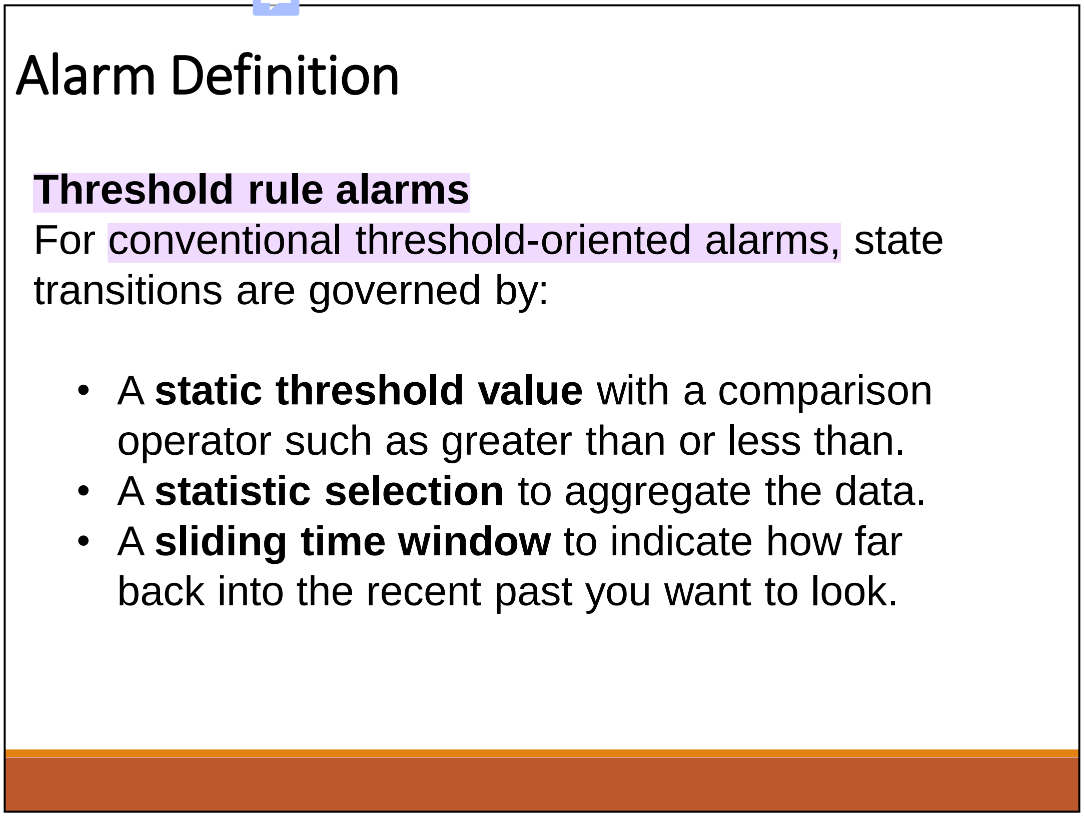


Alarms

Alarms provide user-oriented **Monitoring-as-a-Service** for resources running on OpenStack. This type of monitoring ensures you can automatically scale in or out a group of instances through the Orchestration service, but you can also use alarms for general-purpose awareness of your cloud resources' health.
These alarms follow a tri-state model:

- **ok:** The rule governing the alarm has been evaluated as False.
- **alarm:** The rule governing the alarm has been evaluated as True.
- **insufficient data:** There are not enough datapoints available in the evaluation periods to meaningfully determine the alarm state.

https://docs.openstack.org/aodh/latest/admin/telemetry-alarms.html



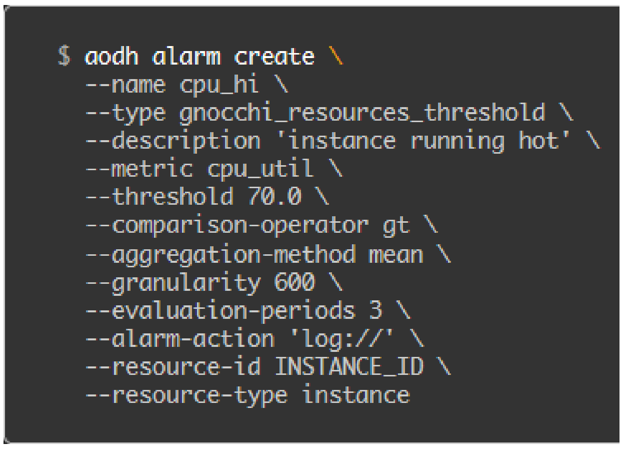
Alarm Definition

Threshold rule alarms
For conventional threshold-oriented alarms, state transitions are governed by:

- A **static threshold value** with a comparison operator such as greater than or less than.
- A **statistic selection** to aggregate the data.
- A **sliding time window** to indicate how far back into the recent past you want to look.

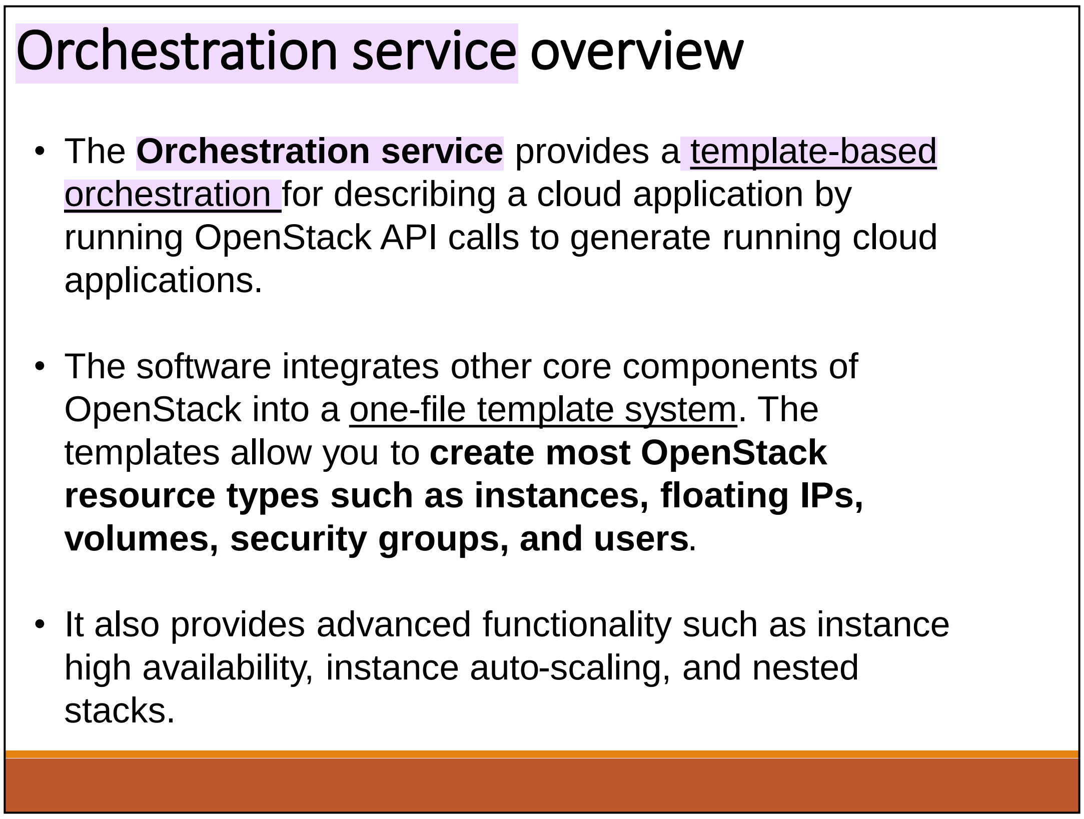
Alarm Creation

An example of creating a Gnocchi threshold-oriented alarm, based on an upper bound on the CPU utilization for a particular instance:



```
$ aodh alarm create \
  --name cpu_hi \
  --type gnocchi_resources_threshold \
  --description 'instance running hot' \
  --metric cpu_util \
  --threshold 70.0 \
  --comparison-operator gt \
  --aggregation-method mean \
  --granularity 600 \
  --evaluation-periods 3 \
  --alarm-action 'log://' \
  --resource-id INSTANCE_ID \
  --resource-type instance
```

This creates an alarm that will fire when the average CPU utilization for an individual instance exceeds 70% for three consecutive 10 minute periods. The notification in this case is simply a log message, though it could alternatively be a webhook URL

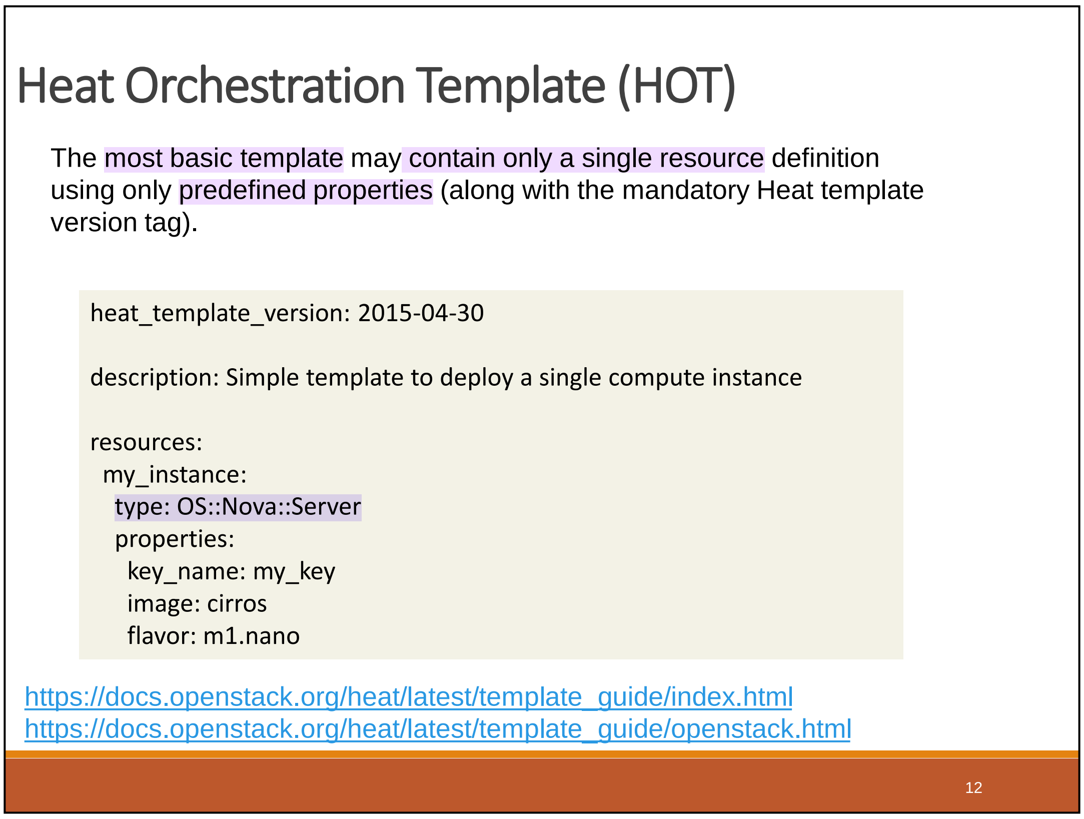


Orchestration service overview

- The **Orchestration service** provides a template-based orchestration for describing a cloud application by running OpenStack API calls to generate running cloud applications.
- The software integrates other core components of OpenStack into a one-file template system. The templates allow you to **create most OpenStack resource types such as instances, floating IPs, volumes, security groups, and users**.
- It also provides advanced functionality such as instance high availability, instance auto-scaling, and nested stacks.

Orchestration service overview

Stack - A stack stands for all the resources necessary to deploy an application. It can be as simple as a single instance and its resources, or as complex as multiple instances with all the resource dependencies that comprise a multi-tier application.



Heat Orchestration Template (HOT)

The most basic template may contain only a single resource definition using only predefined properties (along with the mandatory Heat template version tag).

```
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: my_key
      image: cirros
      flavor: m1.nano
```

https://docs.openstack.org/heat/latest/template_guide/index.html
https://docs.openstack.org/heat/latest/template_guide/openstack.html

12

Each HOT template has to include the *heat_template_version* key with a valid version of HOT, e.g. 2015-10-15 (see Heat template version for a list of all versions). While the *description* is optional, it is good practice to include some useful text that describes what users can do with the template.

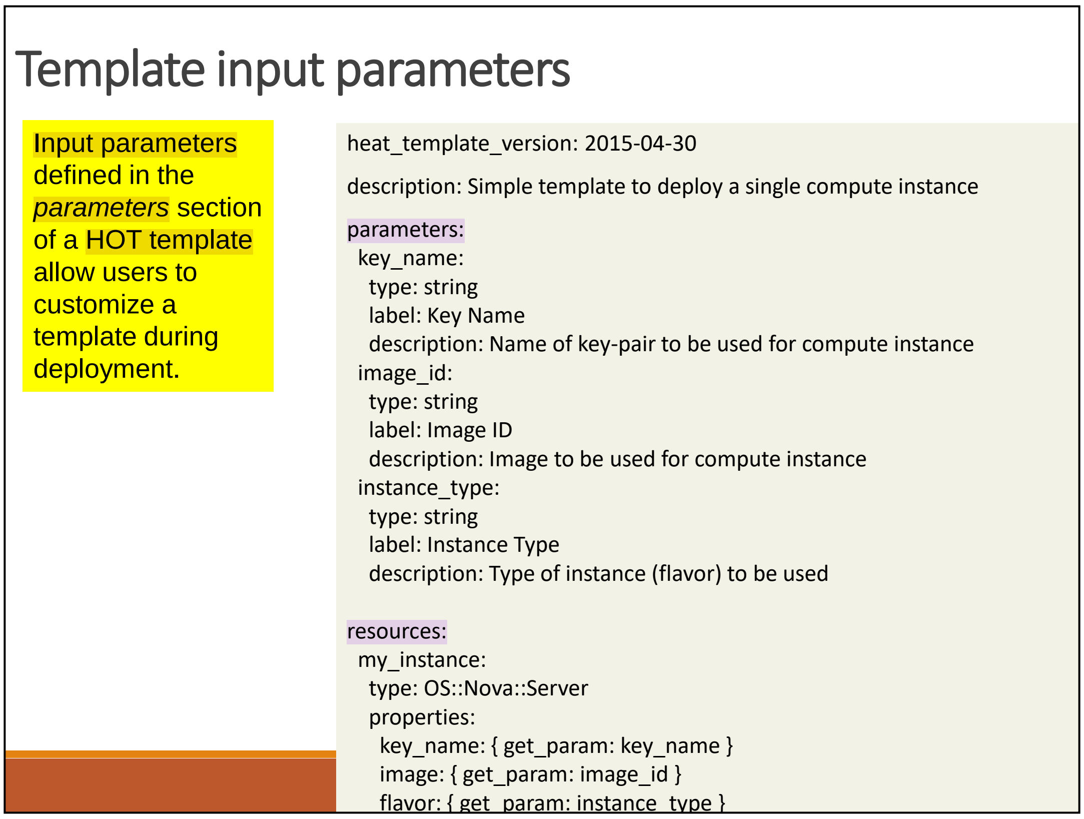
In case you want to provide a longer description that does not fit on a single line, you can provide multi-line text in YAML, for example:

```
description: >
   This is how you can provide a longer description
   of your template that goes over several lines.
```

The *resources* section is required and must contain at least one resource definition. In the example above, a compute instance is defined with fixed values for the ‘key_name’, ‘image’ and ‘flavor’ parameters.

Note that all those elements, i.e. a key-pair with the given name, the image and the flavor have to exist in the OpenStack environment where the template is used. Typically a template is made more easily reusable, though, by defining a set of *input parameters* instead of hard-coding such values.

See OpenStack Resource Types file Heat pag. 239



Template input parameters

Input parameters defined in the *parameters* section of a HOT template allow users to customize a template during deployment.

```
heat_template_version: 2015-04-30

description: Simple template to deploy a single compute instance

parameters:
  key_name:
    type: string
    label: Key Name
    description: Name of key-pair to be used for compute instance
  image_id:
    type: string
    label: Image ID
    description: Image to be used for compute instance
  instance_type:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: { get_param: key_name }
      image: { get_param: image_id }
      flavor: { get_param: instance_type }
```

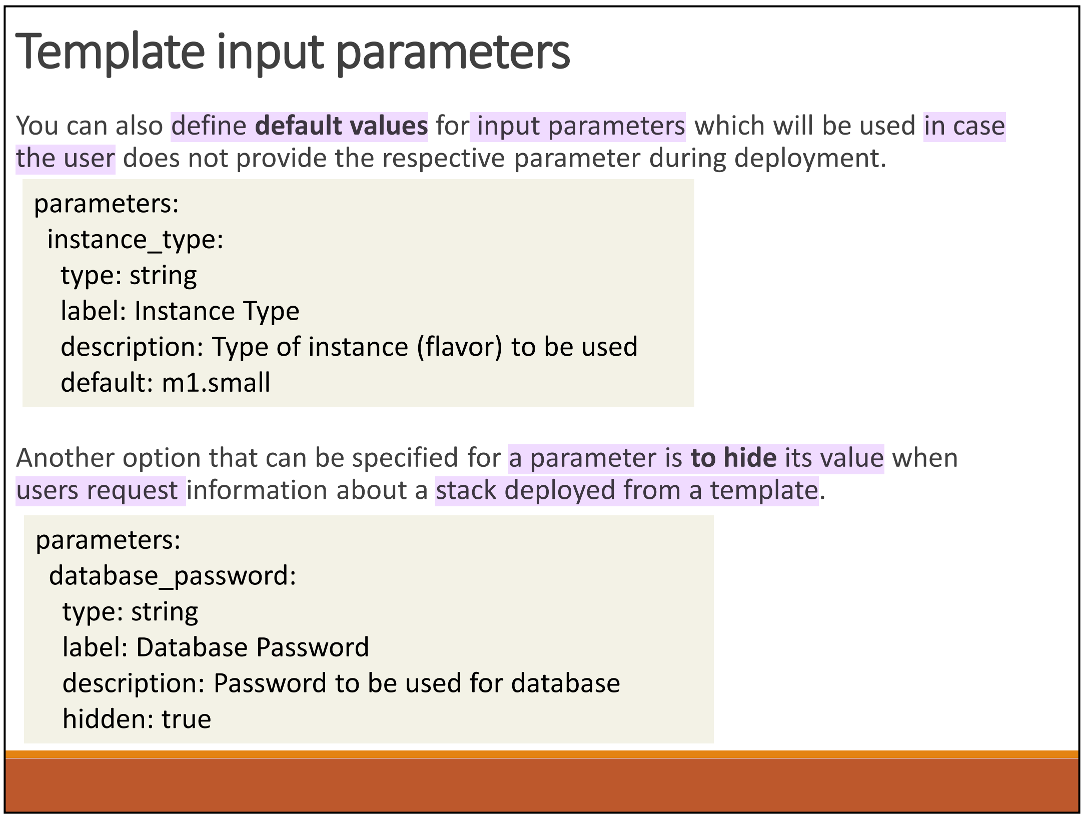
this allows for providing custom key-pair names or image IDs to be used for a deployment. From a template author's perspective, this helps to make a template more easily reusable by avoiding hardcoded assumptions.

Three input parameters have been defined that have to be provided by the user upon deployment. The fixed values for the respective resource properties have been replaced by references to the corresponding input parameters by means of the *get_param* function (see also Intrinsic functions).

The get_param function references an input parameter of a template. It resolves to the value provided for this input parameter at runtime.

OpenStack Resource Types:

https://docs.openstack.org/heat/latest/template_guide/openstack.html



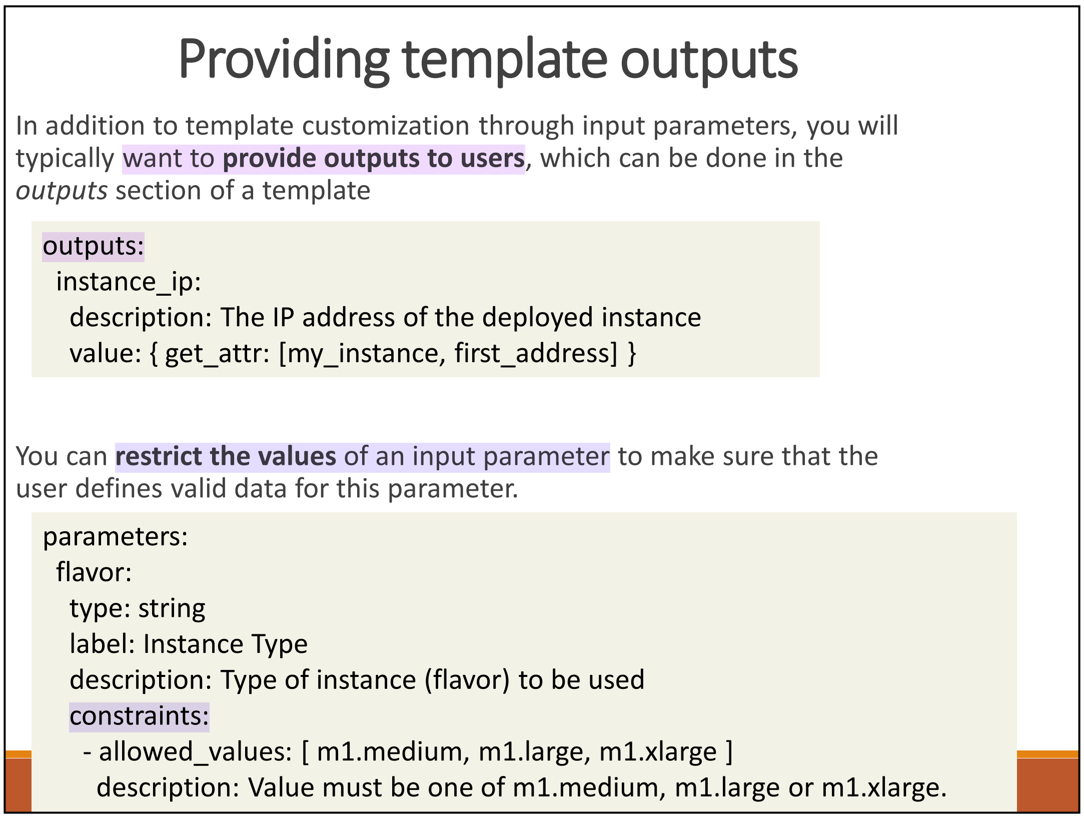
Template input parameters

You can also define **default values** for input parameters which will be used in case the user does not provide the respective parameter during deployment.

```
parameters:
  instance_type:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used
    default: m1.small
```

Another option that can be specified for a parameter is **to hide** its value when users request information about a stack deployed from a template.

```
parameters:
  database_password:
    type: string
    label: Database Password
    description: Password to be used for database
    hidden: true
```



Providing template outputs

In addition to template customization through input parameters, you will typically want to **provide outputs to users**, which can be done in the *outputs* section of a template

```
outputs:
  instance_ip:
    description: The IP address of the deployed instance
    value: { get_attr: [my_instance, first_address] }
```

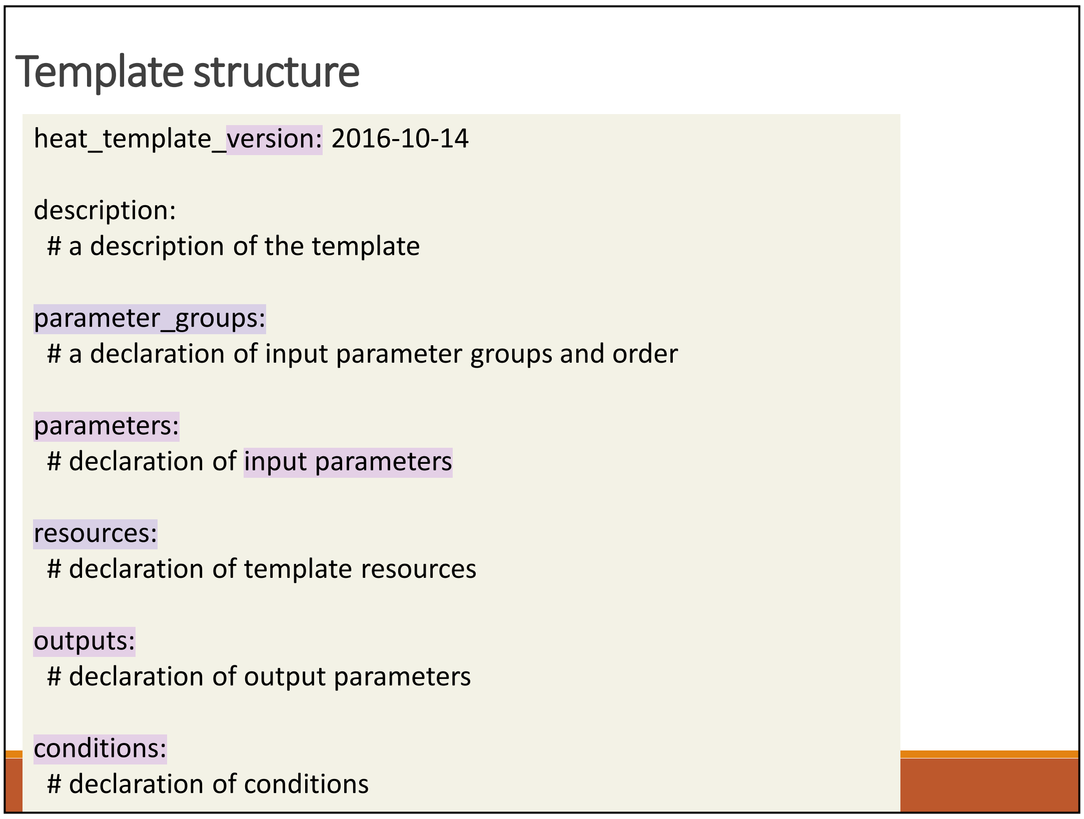
You can **restrict the values** of an input parameter to make sure that the user defines valid data for this parameter.

```
parameters:
  flavor:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used
    constraints:
      - allowed_values: [ m1.medium, m1.large, m1.xlarge ]
        description: Value must be one of m1.medium, m1.large or m1.xlarge.
```

For example, the IP address by which the instance defined in the example above can be accessed should be provided to users. Otherwise, users would have to look it up themselves. The definition for providing the IP address of the compute.

Output values are typically resolved using intrinsic function such as the *get_attr* function in the example above (see also Intrinsic functions). instance as an output is shown.

The get_attr function references an attribute of a resource. The attribute value is resolved at runtime using the resource instance created from the respective resource definition.



Template structure

```
heat_template_version: 2016-10-14

description:
  # a description of the template

parameter_groups:
  # a declaration of input parameter groups and order

parameters:
  # declaration of input parameters

resources:
  # declaration of template resources

outputs:
  # declaration of output parameters

conditions:
  # declaration of conditions
```

The value of heat_template_version tells Heat not only the format of the template but also features that will be validated and supported.

The parameter_groups section allows for specifying how the input parameters should be grouped and the order to provide the parameters in. These groups are typically used to describe expected behavior for downstream user interfaces.
These groups are specified in a list with each group containing a list of associated parameters. The lists are used to denote the expected order of the parameters. Each parameter should be associated with a specific group only once using the parameter name to bind it to a defined parameter in the parameters section.

The parameters section allows for specifying input parameters that have to be provided when instantiating the template. Such parameters are typically used to customize each deployment (e.g. by setting custom user names or passwords) or for binding to environment-specifics like certain images.
Each parameter is specified in a separated nested block with the name of the parameters defined in the first line and additional attributes such as type or default value defined as nested elements.

The resources section defines actual resources that make up a stack deployed from the HOT template (for instance compute instances, networks, storage volumes).
Each resource is defined as a separate block in the resources section.

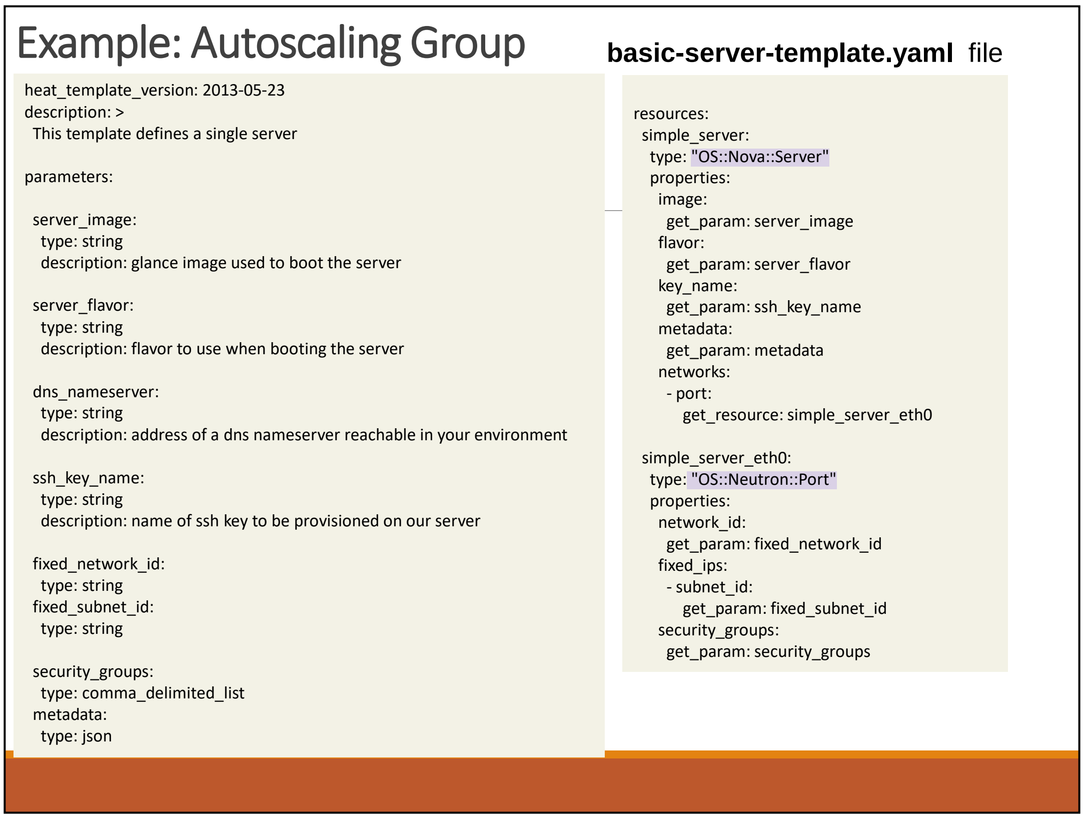
The outputs section defines output parameters that should be available to the user after a stack has been created. This would be, for example, parameters such as IP addresses of deployed instances, or URLs of web applications deployed as part of a stack.
Each output parameter is defined as a separate block within the outputs section

The conditions section defines one or more conditions which are evaluated based on input parameter values provided when a user creates or updates a stack. The condition can be associated with resources, resource properties and outputs. For example, based on the result of a condition, user can conditionally create resources, user can conditionally set different values of properties, and user can conditionally give outputs of a stack.

Example: Autoscaling Group

env.yaml file

```
parameters:
  server_image: cirros
  server_flavor: m1.nano
  dns_nameserver: 8.8.8.8
  ssh_key_name: mykey
  external_network_id: e6e3539a-bf61-4497-af3e-14d59bcc8b72
  cooldown_value: 180
  granularity_value: 300
  threshold_high: 60
  threshold_low: 15
```



Example: Autoscaling Group

basic-server-template.yaml file

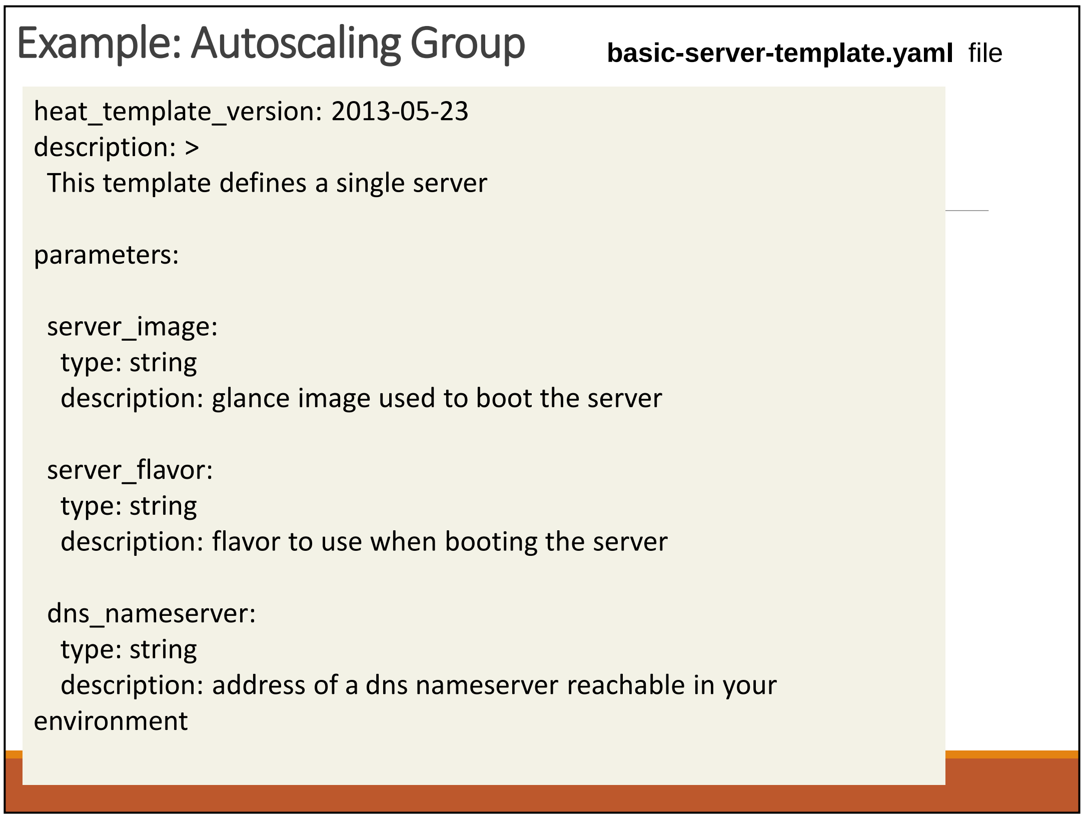
```
heat_template_version: 2013-05-23
description: >
 This template defines a single server

parameters:

 server_image:
  type: string
  description: glance image used to boot the server

 server_flavor:
  type: string
  description: flavor to use when booting the server

 dns_nameserver:
  type: string
  description: address of a dns nameserver reachable in your environment

 ssh_key_name:
  type: string
  description: name of ssh key to be provisioned on our server

 fixed_network_id:
  type: string
 fixed_subnet_id:
  type: string

 security_groups:
  type: comma_delimited_list
 metadata:
  type: json
```

```
resources:
 simple_server:
  type: "OS::Nova::Server"
  properties:
   image:
    get_param: server_image
   flavor:
    get_param: server_flavor
   key_name:
    get_param: ssh_key_name
   metadata:
    get_param: metadata
   networks:
    - port:
      get_resource: simple_server_eth0

 simple_server_eth0:
  type: "OS::Neutron::Port"
  properties:
   network_id:
    get_param: fixed_network_id
   fixed_ips:
    - subnet_id:
      get_param: fixed_subnet_id
   security_groups:
    get_param: security_groups
```

OpenStack resource types:
https://docs.openstack.org/heat/latest/template_guide/openstack.html

https://docs.openstack.org/python-heatclient/latest/cli/stack.html

stack create [-f {json,shell,table,value,yaml}] [-c COLUMN] [--noindent] [--prefix PREFIX] [--max-width <integer>] [--fit-width] [--print-empty] [-e <environment>] [-s <files-container>] [--timeout <timeout>] [--pre-create <resource>] [--enable-rollback] [--parameter <key=value>] [--parameter-file <key=file>] [--wait] [--poll SECONDS] [--tags <tag1,tag2...>] [--dry-run] –t <template> <stack-name>



Example: Autoscaling Group

basic-server-template.yaml file

```
heat_template_version: 2013-05-23
description: >
  This template defines a single server

parameters:

  server_image:
    type: string
    description: glance image used to boot the server

  server_flavor:
    type: string
    description: flavor to use when booting the server

  dns_nameserver:
    type: string
    description: address of a dns nameserver reachable in your
environment
```

Example: Autoscaling Group

basic-server-template.yaml file

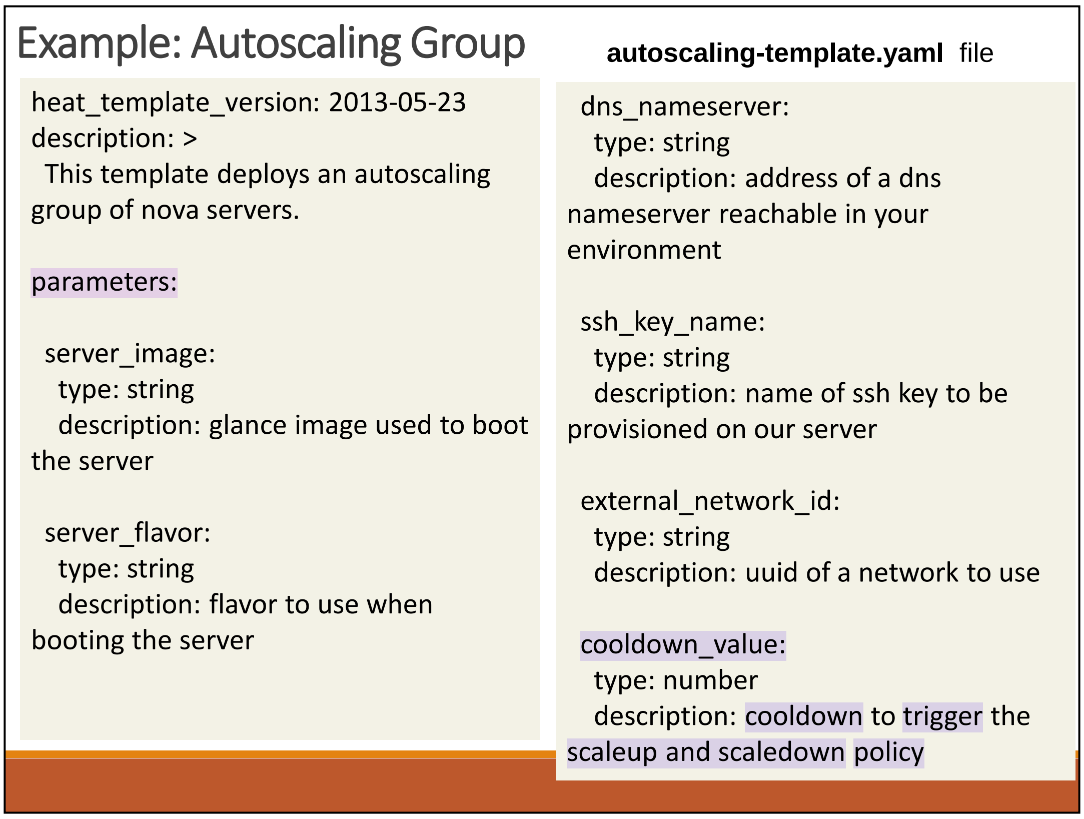
```
ssh_key_name:
  type: string
  description: name of ssh key to be provisioned on our server

fixed_network_id:
  type: string
fixed_subnet_id:
  type: string

security_groups:
  type: comma_delimited_list
metadata:
  type: json
```

Example: Autoscaling Group

basic-server-template.yaml file

```
resources:
  simple_server:
    type: "OS::Nova::Server"
    properties:
      image:
        get_param: server_image
      flavor:
        get_param: server_flavor
      key_name:
        get_param: ssh_key_name
      metadata:
        get_param: metadata
      networks:
        - port:
            get_resource:simple_server_eth0
```

```
simple_server_eth0:
  type: "OS::Neutron::Port"
  properties:
    network_id:
      get_param: fixed_network_id
    fixed_ips:
      - subnet_id:
          get_param: fixed_subnet_id
    security_groups:
      get_param: security_groups
```

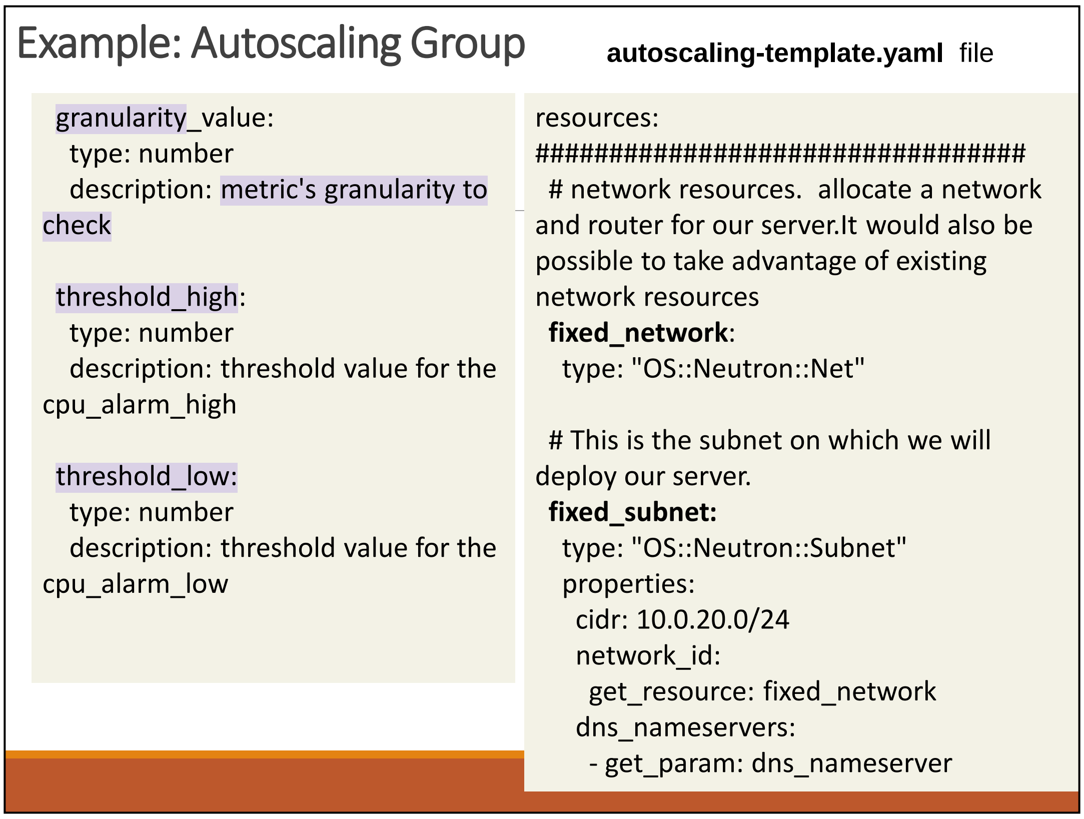


Example: Autoscaling Group

autoscaling-template.yaml file

```
heat_template_version: 2013-05-23
description: >
  This template deploys an autoscaling
group of nova servers.

parameters:

  server_image:
    type: string
    description: glance image used to boot
the server

  server_flavor:
    type: string
    description: flavor to use when
booting the server
```

```
  dns_nameserver:
    type: string
    description: address of a dns
nameserver reachable in your
environment

  ssh_key_name:
    type: string
    description: name of ssh key to be
provisioned on our server

  external_network_id:
    type: string
    description: uuid of a network to use

  cooldown_value:
    type: number
    description: cooldown to trigger the
scaleup and scaledown policy
```

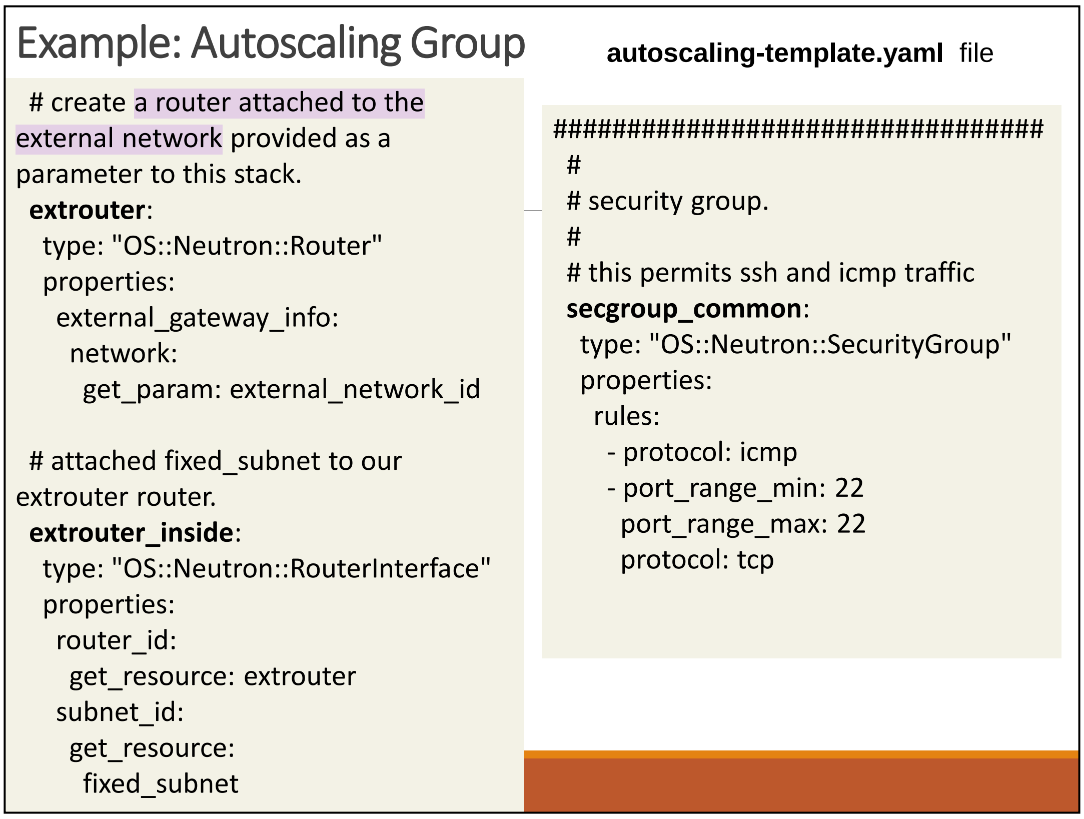


Example: Autoscaling Group

autoscaling-template.yaml file

```
  granularity_value:
    type: number
    description: metric's granularity to
check

  threshold_high:
    type: number
    description: threshold value for the
cpu_alarm_high

  threshold_low:
    type: number
    description: threshold value for the
cpu_alarm_low
```

```
resources:
####################################
  # network resources.  allocate a network
and router for our server.It would also be
possible to take advantage of existing
network resources
  fixed_network:
    type: "OS::Neutron::Net"

  # This is the subnet on which we will
deploy our server.
  fixed_subnet:
    type: "OS::Neutron::Subnet"
    properties:
      cidr: 10.0.20.0/24
      network_id:
        get_resource: fixed_network
      dns_nameservers:
        - get_param: dns_nameserver
```



Example: Autoscaling Group

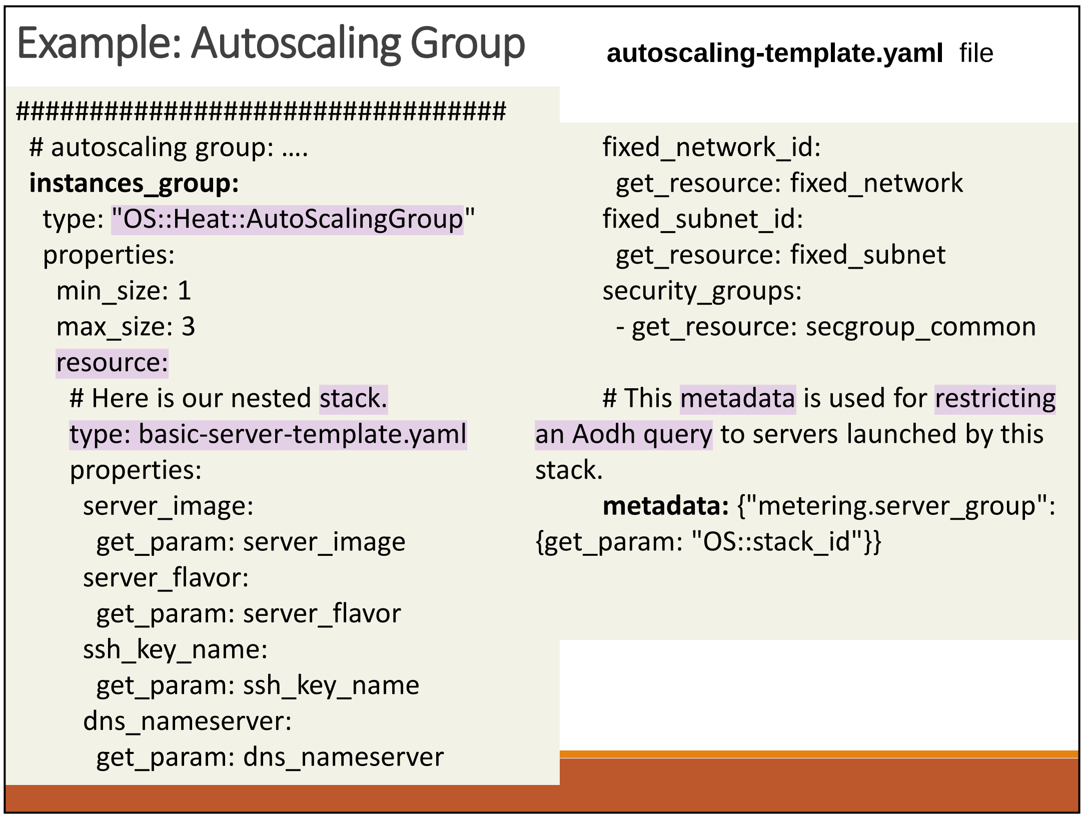
autoscaling-template.yaml file

```
 # create a router attached to the
external network provided as a
parameter to this stack.
 extrouter:
   type: "OS::Neutron::Router"
   properties:
     external_gateway_info:
       network:
         get_param: external_network_id

 # attached fixed_subnet to our
extrouter router.
 extrouter_inside:
   type: "OS::Neutron::RouterInterface"
   properties:
     router_id:
       get_resource: extrouter
     subnet_id:
       get_resource:
         fixed_subnet
```

```
####################################
 #
 # security group.
 #
 # this permits ssh and icmp traffic
 secgroup_common:
   type: "OS::Neutron::SecurityGroup"
   properties:
     rules:
       - protocol: icmp
       - port_range_min: 22
         port_range_max: 22
         protocol: tcp
```

SSH port 22

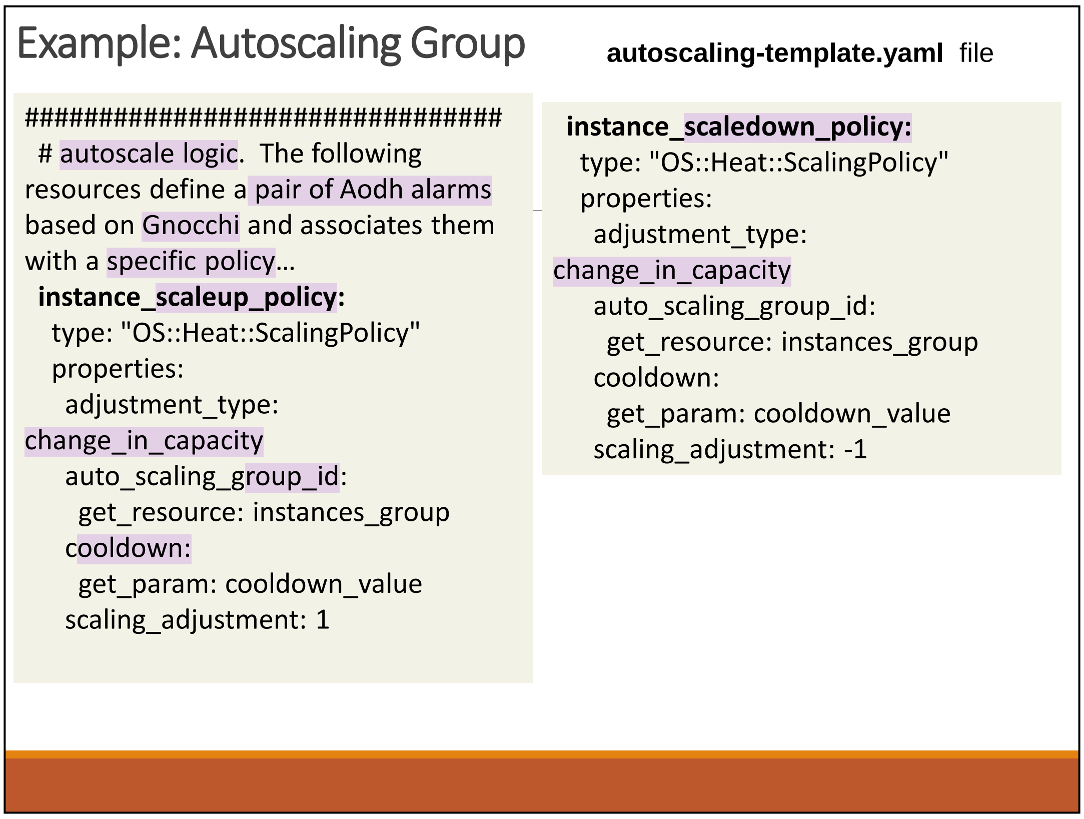
The port is used for Secure Shell (SSH) communication and allows remote administration access to the VM.



Example: Autoscaling Group

autoscaling-template.yaml file

```
####################################
 # autoscaling group: ….
 instances_group:
   type: "OS::Heat::AutoScalingGroup"
   properties:
     min_size: 1
     max_size: 3
     resource:
       # Here is our nested stack.
       type: basic-server-template.yaml
       properties:
         server_image:
           get_param: server_image
         server_flavor:
           get_param: server_flavor
         ssh_key_name:
           get_param: ssh_key_name
         dns_nameserver:
           get_param: dns_nameserver
         fixed_network_id:
           get_resource: fixed_network
         fixed_subnet_id:
           get_resource: fixed_subnet
         security_groups:
           - get_resource: secgroup_common

         # This metadata is used for restricting
an Aodh query to servers launched by this
stack.
         metadata: {"metering.server_group":
{get_param: "OS::stack_id"}}
```



Example: Autoscaling Group

autoscaling-template.yaml file

```
##################################
  # autoscale logic.  The following
resources define a pair of Aodh alarms
based on Gnocchi and associates them
with a specific policy…
  instance_scaleup_policy:
    type: "OS::Heat::ScalingPolicy"
    properties:
      adjustment_type:
change_in_capacity
      auto_scaling_group_id:
        get_resource: instances_group
      cooldown:
        get_param: cooldown_value
      scaling_adjustment: 1
```

```
  instance_scaledown_policy:
    type: "OS::Heat::ScalingPolicy"
    properties:
      adjustment_type:
change_in_capacity
      auto_scaling_group_id:
        get_resource: instances_group
      cooldown:
        get_param: cooldown_value
      scaling_adjustment: -1
```

Details about the resource "OS::Heat::ScalingPolicy“ can be found at

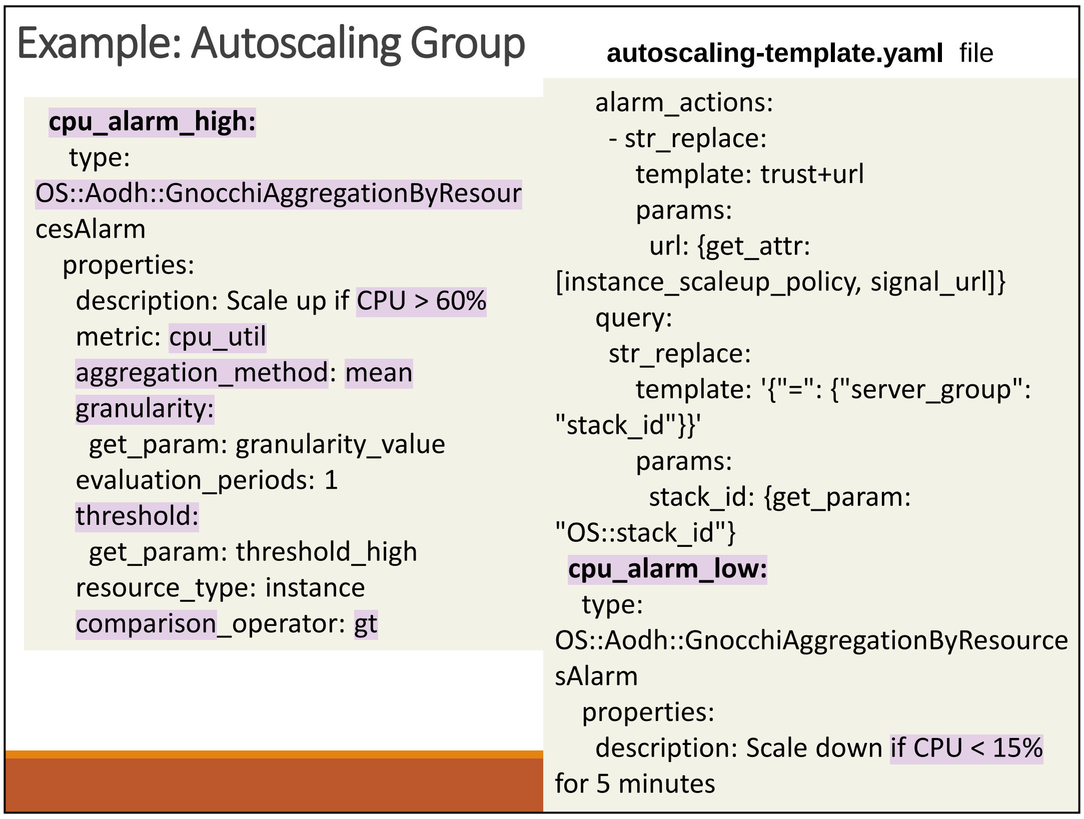
https://docs.openstack.org/heat/pike/template_guide/openstack.html#OS::Heat::ScalingPolicy

In partucular it has the attributes:

alarm_url: A signed url to handle the alarm.

Show: Detailed information about resource.

signal_url: A url to handle the alarm using native API.



Example: Autoscaling Group

autoscaling-template.yaml file

```
  cpu_alarm_high:
    type:
 OS::Aodh::GnocchiAggregationByResour
 cesAlarm
    properties:
      description: Scale up if CPU > 60%
      metric: cpu_util
      aggregation_method: mean
      granularity:
        get_param: granularity_value
      evaluation_periods: 1
      threshold:
        get_param: threshold_high
      resource_type: instance
      comparison_operator: gt
```

```
      alarm_actions:
        - str_replace:
            template: trust+url
            params:
              url: {get_attr:
[instance_scaleup_policy, signal_url]}
      query:
        str_replace:
            template: '{"=": {"server_group":
"stack_id"}}'
            params:
              stack_id: {get_param:
"OS::stack_id"}
  cpu_alarm_low:
    type:
OS::Aodh::GnocchiAggregationByResource
sAlarm
    properties:
      description: Scale down if CPU < 15%
for 5 minutes
```

Specifications of the resource OS::Aodh::GnocchiAggregationByResourcesAlarm can be found in https://docs.openstack.org/heat/pike/template_guide/openstack.html#OS::Heat::ScalingPolicy-attr-signal_url

OpenStack Identity manages authentication and authorization. A trust is an OpenStack Identity extension that enables delegation and, optionally, impersonation through keystone. A trust extension defines a relationship between:

Trustor The user delegating a limited set of their own rights to another user. **Trustee** The user trust is being delegated to, for a limited time.

The trust can eventually allow the trustee to impersonate the trustor. For security reasons, some safeties are added. For example, if a trustor loses a given role, any trusts the user issued with that role, and the related tokens, are automatically revoked.

When adding an alarm action with the scheme `trust+http:`, Aodh allows the user to provide a trust ID to acquire a token with which to make a webhook request (Webhooks are **automated messages sent from apps when something happens**. They have a message—or payload—and are sent to a unique URL). (If no trust ID is provided then Aodh creates a trust internally, in which case the issue is not present.) However, Aodh makes no attempt to verify that the user creating the alarm is the trustor or has the same rights as the trustor - it also does not attempt to check that the trust is for the same project as the alarm.

The nature of the `trust+http:` alarm notifier is that it allows the user to obtain a token given the ID of a trust for which Aodh is the trustee, since the URL is arbitrary and not limited to services in the Keystone catalog.

The str_replace function replaces strings.

signal_url: A url to handle the alarm using native API.

The *get_attr* function allows referencing an attribute of a resource. At runtime, it will be resolved to the value of an attribute of a resource instance created from the respective resource definition of the template.

Aodh calls whatever URL you ask it to. The URL that the Heat autoscaling resources provide (via the signal_url attribute) is for the resource signal API.

Send a signal to a resource
POST
/v1/{tenant_id}/stacks/{stack_name}/{stack_id}/resources/{resource_name}/signal

Example: Autoscaling Group

autoscaling-template.yaml file

```
metric: cpu_util
aggregation_method: mean
granularity:
  get_param: granularity_value
evaluation_periods: 1
threshold:
  get_param: threshold_low
resource_type: instance
comparison_operator: lt
```

```
   alarm_actions:
    - str_replace:
        template: trust+url
        params:
          url: {get_attr:
[instance_scaledown_policy, signal_url]}
   query:
    str_replace:
        template: '{"=": {"server_group":
"stack_id"}}'
        params:
          stack_id: {get_param:
"OS::stack_id"}
```