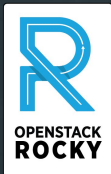




MIRANTIS

Module 5: Orchestration and Telemetry

Orchestration service (Heat)
Telemetry services (Ceilometer / Aodh)



training.mirantis.com



Copyright © 2019 Mirantis, Inc. All rights reserved

1

Heat is the OpenStack Orchestration project. It implements an orchestration engine to launch *composite cloud applications* (multiple types of VMs, optionally with autoscaling policies and load balancing) based on templates in the form of text (YAML) files that can be treated like code. The Heat template format is constantly evolving as additional functionality is added or newer OpenStack resources are created. Heat can be used from the CLI and Dashboard UI, as well as, an OpenStack-native REST API.

OpenStack Telemetry consists of multiple services: Ceilometer, Aodh, CloudKitty, Panko, and Monasca.

In this lecture, you learn about Heat and how to create Heat templates. Near the end of the lecture, autoscaling (Heat, Ceilometer, Aodh) will be discussed.

Objectives

At the end of this presentation, you should be able to:

- Explain the 4 primary sections of a Heat template
- Understand how to define input parameters, including constraints
- Explain how to check the syntax of your Heat template
- Understand where to look for documents that help
- Briefly describe how to implement autoscaling

Orchestration service (Heat)

Overview and architecture

The OpenStack Orchestration service, called Heat, provides template-based orchestration for describing a cloud application, by running OpenStack API calls to generate running cloud applications. The software integrates functionality from other core components of OpenStack (such as, Nova, Glance, and Neutron) into a single-file template. The templates allow you to create most OpenStack resource types, such as instances, floating IPs, volumes, security groups, and users. Templates also provide advanced functionality, such as instance high availability, instance auto-scaling, and nested stacks. This enables OpenStack core projects to satisfy the requirements for a larger user base as compared with the core components.

Orchestration vs. configuration management

- Orchestration is automation, concerned with coordination of multiple components for multilayer applications:
 - For example, instances, networks, volumes, security groups, load balancers, software, and more!
- Configuration Management is automation of server configuration:
 - Typically a declarative model, based on *fact* discovery of the server
 - Abstracts the underlying implementation detail of service deployment
- Both are needed to fully automate cloud application deployment

Heat is used to deploy VMs and OpenStack resources (networks, volumes, load balancers, floating IPs, security groups, and so on).

Configuration management tools are used to configure the infrastructure.

Configuration management encompasses the practices and tooling to automate the delivery and operation of infrastructure. Configuration management solutions model infrastructure, continually monitor and enforce desired configurations, and automatically remediate any unexpected changes or *configuration drift*.

Using **cloud-init** on the VMs and **user_data** in a Heat template, you can invoke configuration management tools from Heat.

Chef, Puppet, Ansible, Vagrant, and SaltStack are examples of configuration management products.

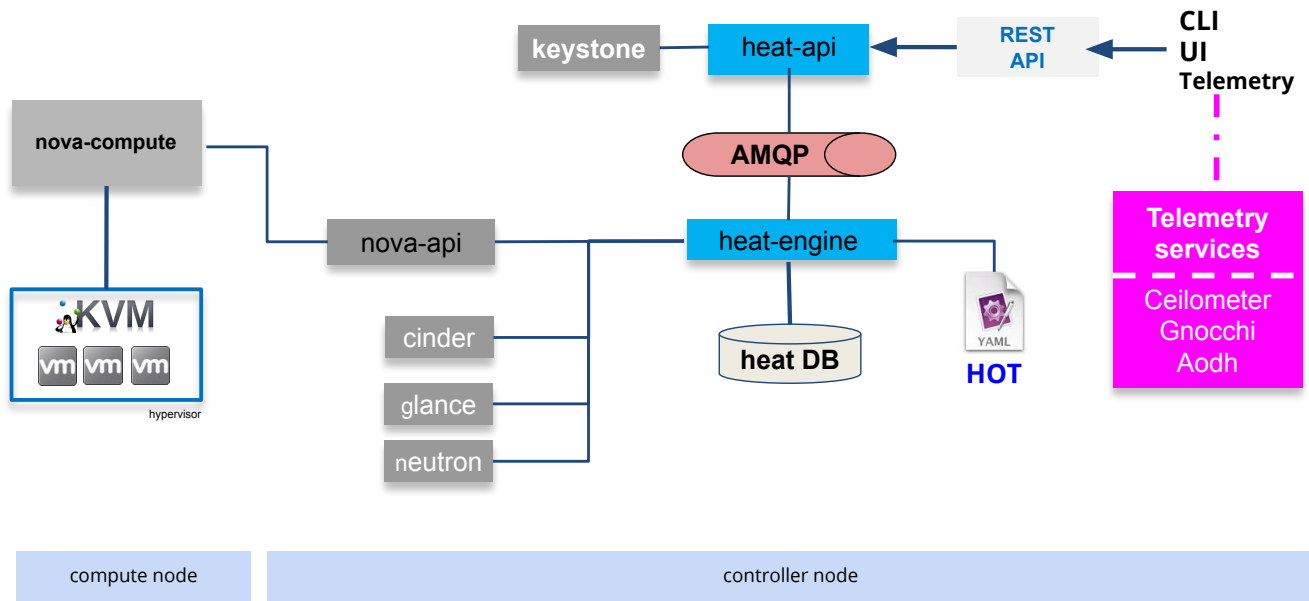
Heat capabilities

- Describes the infrastructure for a cloud application – stack (deployment):
 - OpenStack resources: for example, instances, networks, volumes, security groups, load balancers, software, and more!
 - Relationships between resources: for example, this volume is connected to this server
 - In a YAML text file – **Heat Orchestration Template (HOT)**
- Manages that infrastructure:
 - Changes the infrastructure when the template is modified and re-applied (stack update)
 - Deletes infrastructure when the stack is deleted
- Integrates with software configuration management tools such as Puppet and Chef:
 - For example: create VM with puppet server and install puppet clients on VMs
 - Can also pass parameters
- Provides an autoscaling service that integrates with Ceilometer/Aodh

Companies typically deploy OpenStack to **run applications** in the cloud. Some of those applications have a complex architecture that might require multiple servers, SAN storage, load balancer, multiple networks, etc. For example, when deploying web servers, the typical infrastructure contains several web-servers behind a load balancer, multiple DB servers with replication, etc.

To replicate a similar infrastructure in the cloud, the admin would need to create multiple OpenStack resources including relationships between them. For example, separate VMs for each of the Web and DB servers, Octavia LB pool with VIP and Health Monitor. Heat provides the ability to describe that environment as a template and simplify and automate its creation.

Heat architecture



The Orchestration service consists of the following components:

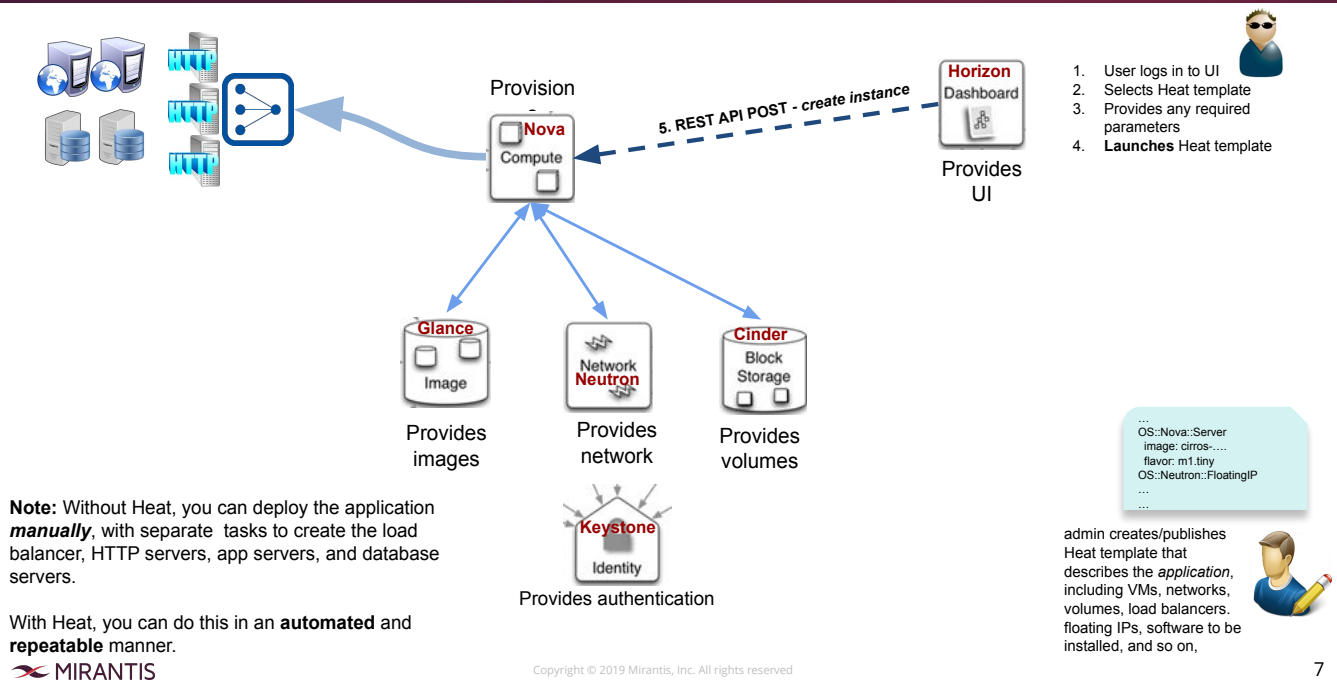
- **heat-api component**: An OpenStack-native REST API that processes API requests by sending them to the heat-engine over Remote Procedure Call (RPC).
- **heat-api-cfn component**: An AWS Query API that is compatible with AWS CloudFormation. It processes API requests by sending them to the heat-engine over RPC.
- **heat-engine**: The heat-engine's main responsibility is to orchestrate the launching of templates (including parsing and validation) and provide events back to the API consumer. Also, responsible for database updates.

The heat engine parses the Heat template (YAML file) and interfaces with all other OpenStack components. For example, Nova (nova-api) to deploy a virtual machine instance or Cinder (cinder-api) to create and attach a volume.

Communication between components uses a message queue product that has implemented AMQP, such as RabbitMQ.

Using the integration with OpenStack Telemetry services, Heat can dynamically adjust resource usage based on demands. This is known as autoscaling. A *scaling group* and *scaling policy* are defined; capable of being triggered through a Ceilometer monitoring alert. Autoscaling allows Heat to provision additional resources to meet demand and remove them later when no longer required.

Orchestration example - overview



Heat templates are used to deploy **composite cloud applications**, consisting of more than a single virtual machine, along with other resources, such as a load balancer, security groups (firewall rules), software installed on the virtual machines at deploy time (for example, Apache or a database), and so on

Typically, an administrator creates the Heat templates. Users can deploy the Heat templates from the CLI, Horizon (Dashboard) UI, or REST API. In this example, there are several Heat resources used by the Heat template (not shown on the slide):

- OS::Nova::Server - 3 instances for HTTP server(s)
- OS::Nova::Server - 2 instances for app server(s)
- OS::Nova::Server - 2 instances for database
- OS::Neutron::FloatingIP
- OS::Octavia::LoadBalancer
- Plus several other resources related to load balancing

When creating VM instances, Heat interacts with Nova (Compute service) to deploy the VMs, software, and other resources. Nova interacts with Glance, Neutron, and so on, as if an **openstack server create** command was issued.

Heat templates can be used to create more than instances. Heat templates can also create OpenStack users, networks, subnetworks, cinder volumes, glance images, load balancers, security groups and rules, and more!

Heat Orchestration Template (HOT)

Syntax and a few simple examples

YAML: “Yet another markup language” or “YAML ain’t markup language.”

This part of the lecture introduces the syntax of Heat orchestration templates (HOTs) and provides several easy to understand examples.

Each Heat template defines the necessary resources for OpenStack to deploy a cloud application.

You might create Heat templates to on-board customers, creating network resources, users, and so on.

Or, you might create a more traditional Heat template to deploy multiple virtual machine instances, including:

- the use of different images, flavors, or networks
- installation and configuration of software
- additional (boot time) customization to create user IDs, add NICs, and so on

First, let's discuss the basics of a Heat template.

Introducing the HOT syntax

```
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
```

- **heat_template_version:** indicates the version of Heat
- **description:** (optional) Describes the template or the workload that can be deployed using the template.
- **parameter_groups:** (optional) Specifies how the input parameters should be grouped and the order to provide the parameters in.
- **parameters:** (optional) Defined input parameters that must to be provided when instantiating the template.
- **resources:** Contains the declaration of the OpenStack resources of the template, such as, servers, networks, volumes, and more!
- **outputs:** (optional) Defines output parameters available to users once the template has been instantiated, such as, IP address of a VM.
- **conditions:** (optional) Includes statements which can be used to restrict when a resource is created or when a property is defined. They can be associated with resources and resource properties in the resources section, also can be associated with outputs in the outputs sections of a template.

A Heat orchestration template (HOT) is a textual specification used to describe an application for Heat. Templates are written in YAML. If you are not familiar with the YAML language, beware that it is **very sensitive to indentation**.

The version date shown on this slide references the OpenStack Newton release, where the **parameter_groups** and **conditions** sections were introduced. Most templates you find on the Web do not include the newer sections. As a result, this lecture does not discuss those sections.

HOT syntax: version and description

- **heat_template_version:** indicates the version of the HOT template:
 - Required for validation of template
 - 2013-05-23: Icehouse
 - 2014-10-16: Juno
 - 2015-04-30: Kilo
 - 2015-10-15: Liberty
 - 2016-04-08: Mitaka
 - 2016-10-14: Newton
 - 2017-02-24: Ocata
 - 2017-09-01: Pike
 - 2018-03-02: Queens
 - 2018-08-31: Rocky
- **description** (optional): text that describes the template: how many servers, what type of servers, and so on
 - Can span more than 1 line

```
heat_template_version: 2013-05-23

description: >

  Version: 2014.1
  This is a heat template for Mirantis OpenStack Training.
  It will start the VM from cirros image, create the Cinder Volume
  Attach the volume to the created VM with specified block device name
  Create a Ceilometer alarm for CPU utilization by this VM
```

heat_template_version is required.

Most Heat template examples that you find on the internet are at the Icehouse level (2013-05-23). Typically, templates written for Icehouse, for example, will run on later releases, such as, Rocky.

Although **description** is optional, it can provide some very useful information about the template.

HOT syntax: parameters

- **parameters** section (optional): defines input parameters, if any, from the user of the template:
 - name of Heat stack
 - name of virtual server(s)
 - what image to use
 - what flavor to use
 - target network
 - software configuration
 - AND MORE!
- Parameters provide choices for the end user when deploying the template
- Heat template author should provide **default** values as well as constraints
 - Best use case is from the Horizon (Dashboard) UI
 - Description provides hover-over help on the Dashboard UI
 - **label** is also displayed in the Horizon (Dashboard) UI

```
parameters:
  <param name>:
    type: <string | number | json | comma_delimited_list>
    label: <human-readable name of the parameter>
    description: <description of the parameter>
    default: <default value for parameter>
    hidden: <true | false>
    constraints:
      <parameter constraints>
```

You can deploy heat stacks from the CLI with parameters, using the **--parameter** operand, as follows:
openstack stack create --template template.yml --parameter param1=value1 --parameter param2=value2 ... *stack_name*

You can also specify the parameters in an environment file and reference that file in the openstack stack create command, as follows:

openstack stack create --template template.yml -e environment.yml *stack_name*

A word regarding (YAML) indentation:

This is the first slide that shows an example of the required indentation (2 blank spaces each). Heat templates are very sensitive to this requirement. For example, you might have every resource, property, and so on, defined correctly ... but ... if 1 line is off by a single blank with regards to the indentation, the Heat template is flagged with an error.

HOT syntax: resources

- **resources** section: defines the resources to be created or used

```
resources:  
  <resource_label>:  
    type: <resource_type>  
    properties:  
      <property name>: <property value>  
      # more resource specific metadata
```

- References OpenStack *resource types*, such as:
 - OS::Nova::Server - Defines a virtual server
 - OS::Cinder::Volume - Defines a volume for Cinder
 - OS::Cinder::VolumeAttachment - Attaches a Cinder volume to an instance
 - OS::Octavia::LoadBalancer - Defines LBaaS v2 load balancer
 - OS::Neutron::SecurityGroup - Allows you to define security group rules for an instance
 - OS::Heat::SoftwareDeployment - Associates a software configuration that is to be deployed to an instance
 - AND MORE!
- Resource properties depend on the resource type
- All resource types are documented in the Heat Resource Type Guide
 - docs.openstack.org/heat/latest/template_guide/openstack.html

The most common resource used in a Heat template is the **OS::Nova::Server** resource to deploy a virtual machine instance.

Notice the resource type syntax?

OS :: <OpenStack_component_name> :: <resource>

OS - OpenStack

OpenStack_component_name - Nova, Glance, Neutron, Cinder, and so on

resource - depends on the component

There are MANY resource types. All are documented in the **Heat Resource Type Guide** URL shown on the slide.

Use the following command to display the list of resource types:

openstack orchestration resource type list

Use the following command to display details for a resource type, such as Nova servers:

openstack orchestration resource type show OS::Nova::Server

HOT syntax: outputs

- **outputs** section (optional): Allows you to set and store output *attributes*, such as the IP address of an instance

```
outputs:  
  <parameter name>:  
    description: <description>  
    value: <parameter value>
```

- This section is *optional* and can be omitted when no output values are required
- The output is *available* once the Heat stack has been instantiated – must take action to retrieve it
 - Dashboard UI, CLI, or REST API
- Refer to the resource type specification for a list of *attributes*

Consider this - When you deploy a Heat template, how do you know the IP address for its instances? The best approach to having that information is to store it in the **stack output**. That is the purpose of the **outputs** section - store data related to the stack.

After the Heat template is deployed, you can retrieve its outputs from REST API, UI, or CLI.

To retrieve the output from the CLI:

- **openstack stack output show --all <stack_name>**

Example 1: Simple HOT to deploy 1 instance

1 3 5 7

```
heat_template_version: 2013-05-23
```

```
description: Simple template to deploy a single compute instance
```

```
resources:
```

```
  nova_instance:
    type: OS::Nova::Server
    properties:
      image: cirros-0.3.5-x86_64
      flavor: m1.tiny
```

nova_instance
User-defined label for the resource

OpenStack (OS) resource type

List of required properties for the
Nova server resource

(notice the indentation – 2 blank spaces each time)

This slide shows a very simple example to use a Heat template to deploy a single virtual machine instance using a cirros image and the m1.tiny flavor.

From the command line, issue a **openstack stack create --template <template_name>** command to deploy the Heat template.

Example 1: Deploy simple HOT from CLI

- To deploy from the CLI:
 - `openstack stack create --template single_server_basic.yaml BASIC`
- After the stack is created:
 - `openstack stack list`

ID	Stack Name	Stack Status	Creation Time	Updated Time
9bfad083-a53c-47c4-9952-9d2915d31e30	BASIC	CREATE_COMPLETE	2019-04-05T18:11:46Z	None

- `openstack server list`

ID	Name	Status	Networks	Image	Flavor
8230abf6-f766-4b22-afc8-1db568b3cf28	BASIC-nova_instance-1jxuz3taa55d	ACTIVE	private=10.0.0.19	cirros-0.3.5-x86_64-disk	ml.tiny

instance_name:
BASIC (stack name)
nova_instance (resource
label in template)

If needed, you can specify the directory with the template name. For example:
`openstack stack create --template ~/hot_templates/single_server_basic.yaml BASIC`

Example 2: Simple HOT using parameters

```
heat_template_version: 2013-05-23
```

parameters:

```
flavor_chosen:
  type: string
  default: m1.small
  constraints:
    - allowed_values: [m1.tiny, m1.small, m1.large]
```

Flavor_chosen parameter:

User-defined input parameter for flavor.
Default: m1.small
Constraints: User can only select tiny, small, or large

resources:

```
nova_instance:
  type: OS::Nova::Server
  properties:
    image: cirros-0.3.3-x86_64
    flavor: { get_param: flavor_chosen }
    name: heatVM
```

get_param function will substitute **flavor_chosen** value

outputs:

```
instance_name:
  value: { get_attr: [nova_instance, name] }
instance_ip:
  value: { get_attr: [nova_instance, first_address] }
```

get_attr function retrieves the IP address and VM name.
Output is stored in the template output



Copyright © 2019 Mirantis, Inc. All rights reserved

16

In this example, the **flavor_chosen** is an input parameter specified by the user at deploy time. The default flavor is m1.small. The user can select from m1.tiny, m1.small, or m1.large. In other words, they cannot select m1.medium.

Also shown is an example of extracting the IP address and instance name of the deployed instance and storing it in the stack output.

OS::Nova::Server resource - parameters / properties

OpenStack Resource Types document

What parameters can you use?

- Each resource *property* can be specified as a Heat template **parameter**
- Some properties are required:
 - flavor for OS::Nova::Server
 - Must be specified in Heat template

OS::Nova::Server

A resource for managing Nova instances.

A Server resource manages the running virtual machine instance within an OpenStack cloud.

Required Properties

flavor

The ID or name of the flavor to boot onto.
String value expected.
Can be updated without replacement.
Value must be of type nova.flavor

```
the_resource:
  type: OS::Nova::Server
  properties:
    admin_pass: String
    availability_zone: String
    block_device_mapping: [{"volume_size": Integer, "volume_
    block_device_mapping_v2: [{"boot_index": Integer, "ephem
    config_drive: Boolean
    deployment_swift_data: {"object": String, "container": S
    diskConfig: String
    flavor: String
    flavor_update_policy: String
    image: String
    image_update_policy: String
```

When developing a Heat template, how do you know what the input parameters might be for a given resource?

In general, any Heat resource **property** can be specified as a parameter. Some properties are required. If you are developing Heat templates, become familiar with the *Heat Resource Type Guide*. For example, this slide shows the **OS::Nova::Server** resource type definition from the *Heat Resource Type Guide*. Flavor is the only required property.

Each resource type supports different properties and attributes. For more details:

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

The Horizon (Dashboard) UI provides a mechanism to display Heat resource types: **Project > Orchestration > Resource Types**.

From the CLI, issue: **openstack orchestration resource type list**

OS::Nova::Server resource - outputs / attributes

What outputs can you specify?

- Resource *attributes* can be used to store data in the stack *outputs*
- For example:
 - instance_name / name
 - first_address
 - networks

OpenStack Resource Types document

OS::Nova::Server

A resource for managing Nova instances.

A Server resource manages the running virtual machine instance with

Attributes

accessIPv4

The manually assigned alternative public IPv4 address of the server.

accessIPv6

The manually assigned alternative public IPv6 address of the server.

addresses

Available since 11.0.0 (Rocky) - The attribute was extended to include subnets and n

A dict of all network addresses with corresponding port_id and subnets. Each network v name and network id. The port ID may be obtained through the following expression: `` name_or_id>, 0, port]]``. The subnets may be obtained trough the following expression: name_or_id>, 0, subnets]]``. The network may be obtained through the following expres <network name_or_id>, 0, network]]``.

console_url

AWS compatible instance h

name

Name of the server.

networks

A dict of assigned network addresses of the form: {"public": [ip1, ip2...], "private": [ip3, ip [ip3, ip4]]. Each network will have two keys in dict, they are network name and network

poller_conf

When developing a Heat template, how do you know what the possible outputs might be for a given resource?

In general, any Heat resource *attribute* can be specified as a stack output. You will need to become familiar with the *Heat Resource Type Guide*. For example, this slide shows the **OS::Nova::Server** resource type definition from the *Heat Resource Type Guide*. Notice the **addresses** attribute. It is used to retrieve the *first_address*.

In general, any Heat resource *attribute* can be specified in the stack output.
Each type of resource supports different properties and attributes.

The Horizon (Dashboard) UI provides a mechanism to display Heat resource types: **Project > Orchestration > Resource Types**.

From the CLI, issue: **openstack orchestration resource type list**

Example 2: Deploy HOT with parameters from CLI

- To deploy from the CLI:
 - `openstack stack create --template VM_with_parms.yaml PARMS --parameter flavor_chosen=m1.tiny`
- After the stack is created:
 - `openstack stack list`

ID	Stack Name	Stack Status	Creation Time	Updated Time
4b55110b-9012-4a30-a53b-7e8f14677602	PARMS	CREATE_COMPLETE	2019-04-05T18:24:37Z	None

- `openstack stack output show PARMS --all`

Field	Value
instance_name	{ "output_value": "heatVM", "output_key": "instance_name", "description": "Name of the instance." }
instance_ip	{ "output_value": "10.0.0.11", "output_key": "instance_ip", "description": "IP address of the instance." }

Outputs for the PARMS stack:

instance name:
heatVM (default)

instance_ip:
10.0.0.11

This example shows how to deploy a Heat stack from the CLI, specifying a value for the `flavor_chosen` parameter.

Since users might not be aware of the possible input parameters, it is always a good idea to provide default values. For example, the instance name is another possible input parameter, with a default of `heatVM`.

The **`openstack stack output show`** command displays the outputs defined in the Heat template along with their instantiated values for the deployed Heat stack.

References

- Start here
 - docs.openstack.org/heat/latest/#hot-spec
- Heat orchestration template (HOT) Guide
 - docs.openstack.org/heat/latest/template_guide/index.html
- Heat resource types (servers, volumes, networks, and more!)
 - docs.openstack.org/heat/latest/template_guide/openstack.html
- Hello World example
 - docs.openstack.org/heat/latest/templates/hot/hello_world.html
 - git.openstack.org/cgit/openstack/heat-templates/tree/hot/hello_world.yaml
- Many templates written by others in git
 - github.com/openstack/heat-templates/tree/master/hot

Many users have shared their Heat templates in GIT. They work! For example:

- `hello_world.yaml`: 1 VM with *user data*
- `vm_with_cinder.yaml`: 1 VM with a Cinder volume
- `servers_in_existing_neutron_net`: 2 VMs with a floating IP
- `asg_of_servers.yaml`: autoscaling example for 1 VM
- `lb_group.yaml`: Load balancer with 2 VMs

Heat Orchestration Template (HOT)

Parameter constraints

Using parameter constraints

```
parameters:  
  <param name>:  
    type: <string | number | json | comma_delimited_list>  
    label: <human-readable name of the parameter>  
    description: <description of the parameter>  
    default: <default value for parameter>  
    hidden: <true | false>  
    constraints:  
      <parameter constraints>
```

- constraints (optional)
 - A list of constraints to apply against a parameter. The constraints are validated by heat-engine when a user deploys a stack. The stack creation fails if the parameter value does not comply to the constraints.
- 2 types: user defined versus custom

User defined parameter constraints - length and allowed pattern

- Specify restrictions on length and/or a pattern

```
parameters:
  user_name:
    type: string
    label: User Name
    description: User name to be configured for the application
    constraints:
      - length: { min: 6, max: 8 }
        description: User name must be between 6 and 8 characters
      - allowed_pattern: "[A-Z]+[a-zA-Z0-9]*"
        description: User name must start with an uppercase character
```

- For example, criteria for creating a user name
 - Similar example applies to passwords

This is an example of a **user defined** constraint for creating a user name. In this case, 2 constraints are used: **length** and **allowed_pattern**.

User defined parameter constraints - allowed values

- Can specify restrictions on parameter value

```
parameters:
  VM_name:
    type: string
    description: Name of the VM instance
    default: heatVM
  image_name:
    type: string
    description: Name of the image
    default: cirros-0.3.5-x86_64-disk
  flavor_chosen:
    type: string
    description: Size of the instance to be created
    default: ml.tiny
    constraints:
      - allowed_values: [ml.tiny, ml.small, ml.medium]
        description: >
          Value must be one of 'ml.tiny', 'ml.small', or 'ml.medium'.
```

- For example, the instance_type parameter value can only be 1 of the **allowed_values**
 - Dashboard UI: User sees list with 3 choices
 - CLI: Error if incorrect parameter value specified

This is an example of a **user defined** constraint. In this example, the user needs to specify a flavor (instance_type parameter). They are restricted to using one of the **allowed_values** (m1.small, m1.medium, m1.large).

Specifying an unsupported value yields an error message:

ERROR: Parameter 'flavor_chosen' is invalid: Value must be one of 'm1.tiny', 'm1.small', or 'm1.medium'.

The *error message text* comes from the **description** specified on the constraint.

User defined parameter constraints - range of values

- Specify restrictions on parameter value range

```
parameters:  
  db_port:  
    type: number  
    description: Database port number  
    default: 50000  
    constraints:  
      - range: { min: 40000, max: 60000 }  
        description: Port number must be between 40000 and 60000
```

- For example, the db_port must be within the specified range (40000 to 60000)
 - Dashboard UI: User sees an error if invalid port
 - CLI: Error if incorrect parameter value specified

This is an example of a **user defined** constraint.

Custom constraints

```
image_name:
  type: string
  label: image to use for VM instance
  description: Name of the image
  default: cirros-0.3.3-x86 64
  constraints:
    - custom_constraint: glance.image
      description: Choose 1 of the listed images for your project
```

- Custom constraints are implemented by plugins; providing *advanced* constraint validation logic
 - Creates a list of all resources of the specified type, owned by the user's project
 - For example, a list of images available to the *demo* project
- Allows **default** value to be specified also
- Dashboard UI is a lot easier to use - resource list is displayed
- Many plugins are provided, such as:
 - cinder.volume
 - glance.image
 - neutron.network
 - neutron.subnet
 - nova.flavor
 - ip_addr
 - mac_addr
 - And more!!

Custom constraints can be used in Heat templates to easily provide a list of resources of the specified type (for example, images) or to provide more advanced constraint validation logic.

There are many available custom constraints. Some are shown on the slide. Each is implemented as a plug-in that is registered with the Heat orchestration engine.

The complete list of custom constraints and their plug-ins is documented in the HOT specification doc: docs.openstack.org/heat/latest/template_guide/hot_spec.html#parameters-section

The example on the slide shows the use of the *glance.image* custom constraint. When launching a Heat stack from the Horizon (Dashboard) UI, the requestor sees a drop-down list of all possible images for their project.

One advantage of this approach – the Heat template does not need to be updated for each new image that is created, in this example.

Example 3: parameter constraints

- User defined constraints:
 - flavor_type: uses allowed_values
- Custom constraints:
 - image_name: glance.image
 - network_name: neutron.network
- **custom_constraints** builds a list of all available resources, such as, images and networks
- UI example shown on next slide
- Notice the use of the **label** property

```
parameters:
  vm_name:
    type: string
    label: Select a name for your instance
    description: The name to use for the VM
    default: test_VM

  image_name:
    type: string
    description: Name of the image
    label: Select an image
    default: cirros-0.3.5-x86_64
    constraints:
      - custom_constraint: glance.image

  flavor_type:
    type: string
    description: Size of the instance to be created
    label: Select a supported instance type
    default: m1.tiny
    constraints:
      - allowed_values: [m1.tiny, m1.small, m1.medium]
        description: >
          Value must be one of 'm1.tiny', 'm1.small', or 'm1.medium'.

  network_name:
    type: string
    description: Name of the network
    label: Select the target network for the instance
    default: private
    constraints:
      - custom_constraint: neutron.network
```

This example shows use of two custom constraints: *glance.image* and *neutron.network*.

- When used from the UI, the image_name and network_name parameters will display all available images and networks, respectively. An example can be seen on the next slide.
- When used from the CLI, the default values specified for image_name and network_name are used if no user-specified value is provided.

The flavor_type parameter uses the constraints defined in the Heat template: m1.tiny, m1.small, or m1.medium

In this case, each parameter has a **label** with text describing the field on the Horizon (Dashboard) UI. If no label is specified, the UI will use the parameter name.

Example 3: parameter constraints seen in the UI

- Provides an easy way to access all resources, such as, images, networks, etc.
 - Without having to maintain the Heat template as new resources are created
- Resources displayed are based on the project
 - Eliminating possible confusion
- HOT *parameter labels* provide descriptive text for each parameter

Launch Stack

Stack Name * ⓘ

Description: Create a new stack with the provided values.

Creation Timeout (minutes) * ⓘ

Rollback On Failure ⓘ

Password for user "demo" * ⓘ

Select a supported instance type ⓘ

Select an image ⓘ

Select the target network for the instance ⓘ

Select a name for your instance ⓘ

m1.tiny

Select Image

amphora-x64-haproxy (574.2 MB)
cirros-0.3.5-x86_64-disk (12.7 MB)

Select Network

private

test_VM

This example shows use of two custom constraints: *glance.image* and *neutron.network*. When used from the UI, the *image_name* and *network_name* parameter fields will display all available images and networks, respectively. When defining custom constraints, the UI does not support a default value. The default can only be used with the **openstack stack create** command from the CLI. The *flavor_type* parameter uses the constraints defined in the Heat template: m1.tiny, m1.small, or m1.medium

Notice the parameter field labels. For example, instead of using the default of *VM_name* (parameter name), the UI uses the parameter label: *name of VM instance*. Using labels for parameters allows you to be more descriptive on the UI.

Heat Orchestration Template (HOT)

Some practical examples

More practical HOT examples

- 4: Deploy multiple instances where 1 depends on the successful deployment of another
- 5: Deploy an instance with security group and rules
- 6: Deploy an instance; associating a floating IP with the instance
- 7: Deploy an instance; create a Cinder volume and attach to the instance

Example 4: HOT with server dependency

```
parameters:
  db_flavor:
    type: string
    default: m1.small
    constraints:
      - allowed_values: [m1.small, m1.medium, m1.large]
  web_flavor:
    type: string
    default: m1.tiny
    constraints:
      - allowed_values: [m1.tiny]
resources:
  DB_server:
    type: OS::Nova::Server
    properties:
      image: cirros-0.3.3-x86_64
      flavor: { get_param: db_flavor }
  Web_server:
    type: OS::Nova::Server
    depends_on: [ DB_server ]
    properties:
      image: cirros-0.3.3-x86_64
      flavor: { get_param: web_flavor }
```

- Heat template to deploy 2 instances
 - OS::Nova::Server
- Both instances use the same image (not required)
- Instance 1 is a *database* server
 - Several flavors are allowed, with **m1.small** the default
- Instance 2 is a *web* server
 - Only **m1.tiny** is allowed for the flavor
- The web server **depends on** the DB server
 - It's deployment is queued until the DB server deployment is complete



Copyright © 2019 Mirantis, Inc. All rights reserved

31

This example shows how to deploy 2 instances where one, `web_server`, instance depends on the second, `DB_server`, instance. The `web_server` instance will not be deployed until the `DB_server` instance has been deployed. The request to provision the `web_server` is created and queued until the `DB_server` instance deployment completes.

Note: There are also **WaitCondition** and **WaitConditionHandle** specifications that provide more complex resource synchronization.

- OS::Heat::WaitCondition
 - A resource to create a synchronization wait point, to be triggered by CM script
- OS::Heat::WaitConditionHandle
 - A resource to signal condition completion. You can signal success by adding `-data-binary '{"status": "SUCCESS"}'`, or signal failure by adding `-data-binary '{"status": "FAILURE"}'`

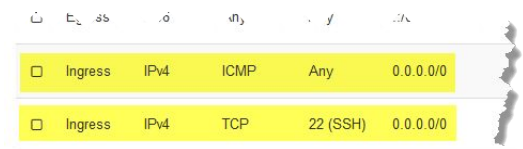
Example 5: Create VM and assign security group rules

resources:

```
my_server:
  type: OS::Nova::Server
  properties:
    image: {get_param: image}
    flavor: {get_param: flavor}
    security_groups:
      - [ default, our_stdSG_rules ]
```

- Deploy an instance
 - OS::Nova::Server
 - Apply security group rules from:
 - default
 - our_stdSG_rules

our_std_SGrules (6843fe0c-fe0a012-b8ceaf754b58)



<input type="checkbox"/>	Ingress	IPv4	ICMP	Any	0.0.0.0/0
<input type="checkbox"/>	Ingress	IPv4	TCP	22 (SSH)	0.0.0.0/0

Security groups define sets of **IP filter table (firewall) rules** that are applied to a virtual machine instance. Security groups can block or allow traffic in to (ingress) or out of (egress) a VM instance. Security groups are project-specific; project members can edit the rules for their **default** group as well as add new rule sets.

All projects have a **default** security group, which is applied to instances that have no other security group defined. Unless changed, the default security group:

- Allows all outbound (egress) traffic
- Allows all traffic between VMs (of the same security group)
- Blocks all inbound (ingress) traffic

This example shows how to apply rules from an *existing security group* to a VM instance as it is deployed: allow ping (all ICMP) and SSH (TCP requests on port 22) requests.

Example 5: Create and assign security group to VM

resources:

```
my_server:
  type: OS::Nova::Server
  properties:
    image: {get_param: image}
    flavor: {get_param: flavor}
    security_groups:
      - {get_resource: the_sg}
```

- Deploy an instance
 - OS::Nova::Server
 - Apply security group, **the_sg**, rules

the_sg:

```
type: OS::Neutron::SecurityGroup
properties:
  name: AllowPingSSH
  description: Allow ping (icmp) and SSH (tcp, port 22)
  rules:
    - protocol: icmp
    - protocol: tcp
      port_range_min: 22
      port_range_max: 22
```

- Create security group
 - OS::Neutron::SecurityGroup
 - Allow
 - ICMP (ping)
 - SSH (TCP port 22)



Copyright © 2019 Mirantis, Inc. All rights reserved

33

Security groups define sets of **IP filter table (firewall) rules** that are applied to a virtual machine instance. Security groups can block or allow traffic in to (ingress) or out of (egress) a VM instance. Security groups are project-specific; project members can edit the rules for their **default** group as well as add new rule sets.

All projects have a **default** security group, which is applied to instances that have no other security group defined. Unless changed, the default security group:

- Allows all outbound (egress) traffic
- Allows all traffic between VMs (of the same security group)
- Blocks all inbound (ingress) traffic

Quite often, users will require HTTP (default TCP port 80) or SSH (default TCP port 22) access to their instances. There are 2 steps to this process. First, you must either update the default security group (not recommended) or create a new security group and apply it against the instance. Second, you need a floating IP address. The floating IP address is shown in the next example.

This example shows how to use a Heat template to create a security group, called **AllowPingSSH**, that defines the necessary security group rules to allow ping (all ICMP) and SSH (TCP requests on port 22) requests for virtual machine instances deployed with this template.

Notice the security group, **the_sg**, is defined as a Heat resource (type: OS::Neutron::SecurityGroup).

Note: When you delete the Heat stack, the VM instance is deleted, as well as the security group and its rules.

Example 6: Create and associate FIP to VM

```
resources:
  nova_instance:
    type: OS::Nova::Server
    properties:
      name: { get_param: vm_name }
      image: { get_param: image_name }
      flavor: { get_param: flavor_type }
      availability_zone: nova
      networks:
        - network: { get_param: network_name }
```

```
floating_ip:
  type: OS::Neutron::FloatingIP
  properties:
    floating_network_id: public
```

```
floating_ip_assoc:
  type: OS::Neutron::FloatingIPAssociation
  properties:
    floatingip_id: { get_resource: floating_ip }
    port_id: { get_attr: [nova_instance, addresses, { get_param: network_name }, 0, port]}
```

- Deploy instance
 - OS::Nova::Server
- Create new floating IP
 - OS::Neutron::FloatingIP
 - On *public* network

- Associate floating IP to instance
 - OS::Neutron::FloatingIPAssociation
 - Uses ***get_resource*** for ID of floating IP
 - Uses ***get_attr*** for ID of port on instance

Quite frequently, when you deploy an instance, you want to SSH into it. The previous example showed how to allow SSH (TCP port 22) ingress connections. You also need a floating IP (IP address on the external/public network) that is associated with the VM instance.

The OS::Neutron::FloatingIPAssociation resource needs the ID of the port on the instance as well as the ID of the floating IP address..

The ***get_resource*** function references another resource within the same template. At runtime, it is resolved to the ID of the referenced resource. In this case, the ID of the floating IP address.

Example 7: Create and attach volume to VM

- Deploy instance
 - OS::Nova::Server
- Create volume
 - OS::Cinder::Volume
 - Default size of 1 GB
 - Can be overridden by **vol_size** input parameter
- Attach volume to instance
 - OS::Cinder::VolumeAttachment
 - Uses **get_resource** for
 - instance ID
 - volume ID

```
parameters:
  key_name:
    type: string
    description: The name of the key to use for the instance
    default: private

  vol_size:
    type: number
    description: The size of the Cinder volume
    default: 1

resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: { get_param: key_name }
      image: { get_param: image }
      flavor: { get_param: flavor }
      networks: [{ network: { get_param: network } }]

  my_vol:
    type: OS::Cinder::Volume
    properties:
      size: { get_param: vol_size }

  vol_att:
    type: OS::Cinder::VolumeAttachment
    properties:
      instance_uuid: { get_resource: my_instance }
      volume_id: { get_resource: my_vol }
      mountpoint: /dev/vdb
```

This example shows how you can use Heat to automate the deployment of a VM, creating a new Cinder volume, and dynamically attach the volume to the instance. The OS::Cinder::VolumeAttachment resource needs the ID of the instance as well as the ID of the volume. The **get_resource** function retrieves each ID.

Customizing VM instances

Install software and configure VMs



The ability to customize VM instances is another common use case for Heat templates. For example, you might need to install and configure software on the VM. Perhaps you might need to create a non-root user ID on the VM. Using Heat, you can automate these tasks as well as many others.

Customizing VM instances

- Three methods for server configuration:
 - Build custom image with all software/configuration updates
 - Install software/update configuration at boot time
 - Run user-data scripts at **initial boot**
 - Requires **cloud-init**
 - Can run scripts/commands natively
 - Or, use Heat **OS::Heat::CloudConfig** resource instead
 - Use Heat software deployment resources at **initial boot**
 - Requires several Linux packages

Deploying VMs typically requires the operating system plus additional software/customization to the VM.

There are several choices available:

- (1) Build a custom image with all of the software/customization pre-installed and configured.
- (2) Take advantage of cloud-init to run scripts, for example, at initial boot of the VM
- (3) Use Heat software deployment resources to install and configure software
 - Note: Requires cloud-init in the base image, as well as several other Linux packages

(2) and (3) require **cloud-init** package in Linux image or **Cloudbase-Init** in Windows image.

OS::Heat::CloudConfig can be used instead of coding scripts/commands to run.

Custom images

- Custom image building
 - Advantages:
 - Boots faster - No need to download and install anything at boot time
 - Boot reliability - Eliminates download failures due to transient network failures or inconsistent software repositories
 - Tested - Image updates verified in test environments before deployment in production
 - No configuration dependencies - post-boot configuration may require agents installed and enabled on guest VM
 - Disadvantages:
 - Creates many images to maintain; known as *image sprawl*
 - Less flexible - Cannot override any configuration

For custom images, you might hear the term that software is ***baked into the image***. In other words, the software is installed and customized in the image.

If you choose to build custom images, consider the pros and cons. For example, the VMs will boot more quickly. However, suppose you need to update database software in the image. You must rebuild the image and retest it. More than likely you still need the old image. Now, you have multiple images. Consider an update to, for example, the HTTP server in the same image. You have to create another image. Creating new images that are very similar to existing images creates an unmanageable scenario, known as *image sprawl*.

Let's look at OpenStack Heat and exploiting cloud-init.

cloud-init

- An *open-source package* from Ubuntu that is the industry standard for bootstrapping cloud servers (*initialization on **first boot***)
- Available on various Linux distributions such as Ubuntu Cloud images and the official Ubuntu images available on EC2
- Some of the things it configures are:
 - setting a default locale
 - setting hostname
 - resizing boot disk to that specified in boot flavor
 - adding ssh keys to user's .ssh/authorized_keys so they can log in
 - setting up ephemeral mount point
 - And more - see the notes

Cloud-init supports several different input formats for user data. The basic syntax is:

- Shell scripts (starts with **#!**)
- Cloud config files (starts with **#cloud-config**)

Examples of what you can do with cloud-init can be found at:

<http://cloudinit.readthedocs.org/en/latest/topics/examples.html>

For example:

- Adding a yum repository
- Adding an apt repository
- Manage disks, partitions, and filesystems
- Creating users
 - For example, most customers do not want to provide the user with root access. Instead, they create a new user during the provisioning process.
- Managing packages and package repositories
- Writing content files
- Bootstrapping Chef or Puppet
- Defining SSH keys

Use **cloudbase-init** for Windows VMs.

Example 8: Deploy VM and run simple user script

```
heat_template_version: 2013-05-23
parameters:
  my_flavor:
    type: string
    default: ml.small
    constraints:
      - allowed_values: [ml.small, ml.medium, ml.large]
resources:
  my_server:
    type: OS::Nova::Server
    properties:
      image: cirros-0.3.5-x86_64-disk
      flavor: { get_param: my_flavor }
      availability_zone: nova
      networks:
        - network: private
      user_data: |
        #!/bin/bash
        cd /tmp
        wget http://my.fake.site.com/myscript.sh
        chmod +x ./myscript.sh
        /tmp/myscript.sh
        ...
```

- Heat template to deploy a VM
- User_data; at initial boot:
 - Get copy of myscript.sh
 - Make it executable
 - Run it

This slide shows a simple example of how you can use the **user_data** property in a Heat template to issue several commands at *initial boot* of the virtual machine instance.

user_data allows you to enter commands or scripts. For example, to define a second NIC, define a user and password, install software, and more.

In this example, the *myscript.sh* script is downloaded to the virtual machine and executed.

Example 9: Deploy VM with Apache install/configuration

```
user_data:
  str_replace:
    params:
      $app_lbaaS_vip: { get_param: app_lbaaS_vip }
    template: |
      #!/bin/bash -v
      #centos has this "security" feature in sudoers to keep scripts from sudo, comment it out
      sed -i '/Defaults \+requiretty/s/~/#/' /etc/sudoers
      #use apt-get for Debian/ubuntu, and yum for centos/fedora
      if apt-get -v &> /dev/null
      then
        apt-get update -y
        apt-get upgrade -y
        #Install Apache
        apt-get -y --force-yes install apache2
        apt-get install -y libapache2-mod-proxy-html libxml2-dev
        a2enmod proxy
        a2enmod proxy_http
        a2enmod deflate
        a2enmod proxy_html
        cat > /etc/apache2/sites-enabled/000-default.conf << EOL
        <VirtualHost *:80>
          ProxyPreserveHost On
          ProxyPass / http://$app_lbaaS_vip/
          ProxyPassReverse / http://$app_lbaaS_vip/
          ServerName localhost
        </VirtualHost>
      EOL
      /etc/init.d/apache2 restart
      elif which yum &> /dev/null
      then
        #yum update -y
        #Install Apache
        yum install -y httpd
        yum install -y wget
```

Debian/Ubuntu

CentOS/Fedora

- Automate the installation and configuration of Apache for:
 - Debian/Ubuntu (apt-get)
 - CentOS/Fedora (yum)

This example is a bit more realistic. It uses **user_data** to install the Apache2 http server on a VM.

This template works for both Debian/Ubuntu (highlighted on the slide) using **apt-get** versus CentOS/Fedora using **yum** to install Apache.

This is an example of a 3-tier Web Application - WordPress/LAMP. This Heat example is a subset to install the Apache2 part of the LAMP (Linux, Apache, MySQL, and PHP/Python/Perl) stack. For complete details:

https://github.com/openstack/osops-tools-contrib/blob/master/heat/lamp/lib/heat_web_tier.yaml

Example 10: str_replace (1)

```
resources:
  nova_instance:
    type: OS::Nova::Server
    properties:
      name : {get_param: vm_name}
      image: { get_param: image_name }
      flavor: { get_param: flavor_type }
      availability_zone: nova
      networks:
        - network: { get_param: network_name }
```

```
user data:
  str_replace:
    template: |
      #!/bin/bash
      echo "Hi ${instance_name}"
    params:
      ${instance_name} : {get_param : vm_name}
```

- For a VM instance (OS::Nova::Server)
- use **str_replace** to
 - Echo a message at **initial boot**
 - "Hi vm_name"
 - vm_name is a parameter
- **Note:** In this example, the image must include **cloud-init**

The **str_replace** function replaces strings.

- Specifies the mapping of the string for replacement. Other functions such as `get_attr` or `get_param` can be used.
- Specifies the string that is the replacement source (target)

For more details:

https://docs.openstack.org/heat/latest/template_guide/hot_spec.html#str-replace

Example 11: str_replace (2)

```
resources:
  my_instance:
    type: OS::Nova::Server

outputs:
  Login_URL:
    description: The URL to log into the deployed application
    value:
      str_replace:
        template: http://host/MyApplication
        params:
          host: { get_attr: [ my_instance, first_address ] }
```

- For a VM instance (OS::Nova::Server)
- use **str_replace** to
 - Substitute the VM (host) IP address (first_address) in the URL to access the application
 - Stored as a stack output
- For example,
 - http://10.0.0.8/MyApplication

user-data and Nova (reference only)

```
openstack server create --image cirros35 --flavor m1.tiny  
--user-data mydata.file VM_INSTANCE
```

```
#!/bin/bash
```

```
< insert code here to retrieve user ($newuser) and password ($newpassword) from corporate directory >
```

```
# quietly add a user without password
```

```
adduser --quiet --disabled-password --shell /bin/bash --home /home/$newuser --gecos "User" $newuser
```

```
# set password
```

```
echo "$newuser:$newpassword" | chpasswd
```

The ability to customize a VM is not limited to Heat templates. This slide shows an example using the **openstack server create** command with a script provided to Nova in the `mydata.file`. The script will automatically define a new user ID and password, at initial boot time of the VM, using user-data (and **cloud-init**). The user ID and password are retrieved from a corporate directory (for example, Microsoft Active Director) as `$newuser` and `$newuserpassword`.

A special key in the **metadata service** holds the user data provided until boot time when the guest instance can access it.

Heat software deployment resources

- OS::Heat::SoftwareConfig
 - Heat resource for describing and storing software configuration
 - Creates a reference to an immutable configuration management script, optionally parameterized with input values, stored in Heat database
- OS::Heat::SoftwareDeployment
 - Heat resource to associate a VM with a configuration defined by OS::Heat::SoftwareConfig
 - Allows for input values to be defined and passed to the configuration
 - After configuration script completes, its output is available as attributes of this resource
 - Can be triggered on stack create, update, suspend, resume, delete
 - Allows software configurations to be added or removed from a server throughout its life-cycle **without rebuilding the instance**

For more details:

http://docs.openstack.org/developer/heat/template_guide/openstack.html#OS::Heat::SoftwareConfig

http://docs.openstack.org/developer/heat/template_guide/openstack.html#OS::Heat::SoftwareDeployment

The Software Deployment Guide provides many more details:

http://docs.openstack.org/developer/heat/template_guide/software_deployment.html

For example, when creating a custom image that will run Heat SoftwareConfig and SoftwareDeployment resources, you might need

- os-collect-config, os-refresh-config, os-apply-config
 - Collectively responsible for polling of changes in Heat and Nova metadata, and applying the changes to the instance
- heat-config, heat-config-hook, heat-config-notify, heat-config-puppet, etc.
 - These hooks function in relation to the "group" property of SoftwareConfig
 - The group property is used to specify the type of SoftwareConfig hook that will be used to deploy the configuration

Validating Heat templates

When creating Heat templates, you will make errors.
This lesson discusses how you can validate your Heat templates
to find and fix errors.

Heat template validation

- During development of a Heat template, you might want to validate the syntax
- From the CLI, use the **--dry-run** parameter

```
openstack stack create --template template_name.yaml stack_name --dry-run
```

- Sample (common) errors on next slide
- The UI performs some validation
 - Error messages might be very generic and not helpful

OpenStack provides the ability to validate Heat templates with the addition of the `--dry-run` parameter. When you add `--dry-run`, the yaml file will be checked against the expected resource properties, attributes, syntax, etc. documented in the [Heat template guide](#).

You might also want to try other online tools, such as:

<http://yaml-online-parser.appspot.com/>

<http://www.yamllint.com/>

The Dashboard UI provides some validation, but the error messages might be too generic; therefore, not very useful.

The next several slides provide examples of some common errors.

Common template errors (1)

- Invalid template version

ERROR: The template version is invalid: "heat_template_version: 20103-05-23". "heat_template_version" should be one of: 2013-05-23, 2014-10-16, 2015-04-30, 2015-10-15, 2016-04-08, 2016-10-14, 2017-02-24, 2017-09-01, 2018-03-02, 2018-08-31, newton, ocata, pike, queens, rocky

- Invalid resource type

ERROR: The Resource Type (OS::Nova:Server) could not be found.

- Invalid resource property

ERROR: Property error: : resources.nova_instance.properties: : Unknown Property **flavvor**

- Block syntax/parsing error (indentation)

Error parsing template file:///opt/stack/hot_templates/MyBadTemplate.yaml mapping values are not allowed here in "<unicode string>", **line 11, column 24:**

availability_zone: nova

- Invalid template version: The year is incorrect
- Invalid resource type: Notice the missing ‘:’
- Invalid resource property: The **flavvor** property is not valid for the OS::Nova::Server resource type. It should be one of the supported resource properties.
- Syntax/parsing error: Look for the line specified, (11 in this case) it is displayed on the following line (availability_zone: nova). **The error is most likely on the preceding line - line 10.**

Common template errors (2)

- Missing required property

ERROR: Property error: : resources.nova_instance.properties: : **Property flavor not assigned**

- Invalid resource property value

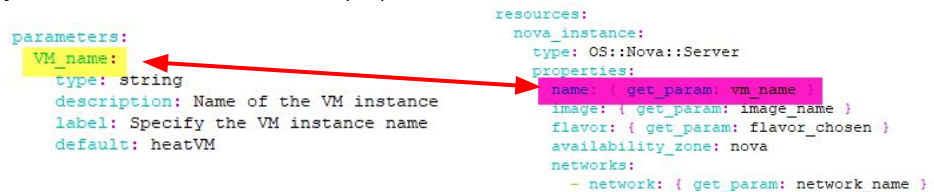
ERROR: Property error: : resources.nova_instance.properties.image: : Error validating value 'cirros-0.3.3-x86_64-disk': **No images matching {'name': u'cirros-0.3.3-x86_64-disk'}.**

- Invalid parameter name

ERROR: Property error: : resources.nova_instance.properties.name: : The Parameter (**vm_name**) was not provided.

```
parameters:
  VM_name:
    type: string
    description: Name of the VM instance
    label: Specify the VM instance name
    default: heatVM

resources:
  nova_instance:
    type: OS::Nova::Server
    properties:
      name: { get_param: vm_name }
      image: { get_param: image_name }
      flavor: { get_param: flavor_chosen }
      availability_zone: nova
      networks:
        - network: { get_param: network_name }
```



- Missing required property: The flavor property must be specified for OS::Nova::Server. It is missing.
- Invalid resource property value: The value for each resource property is validated. In this example, the cirros-0.3.3-x86_64-disk image does not exist.
- Invalid parameter name: The parameter name is specified as VM_name, but it is referenced as vm_name in the OS::Nova::Server resource.

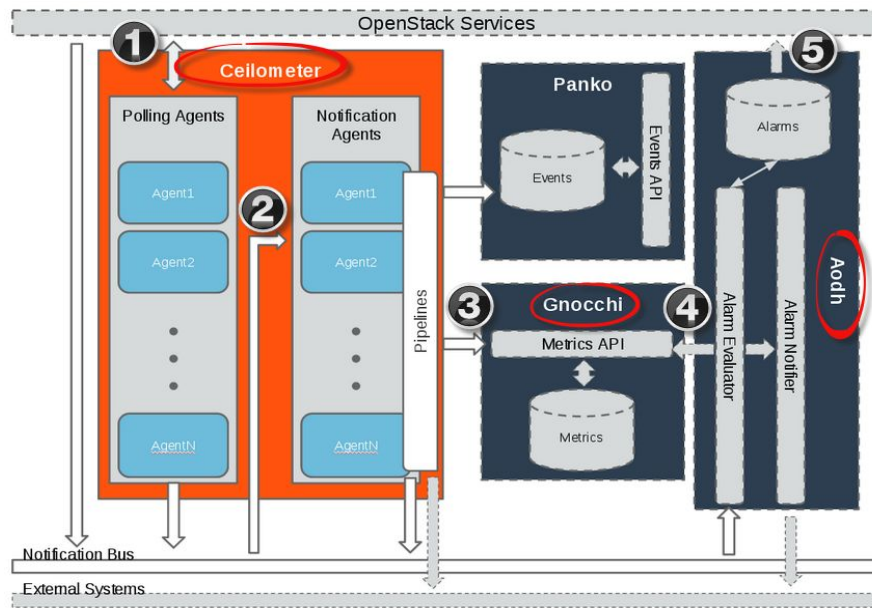
Autoscaling Cloud applications

Heat / Ceilometer / Gnocchi / Aodh

This lesson introduces the topic of autoscaling cloud applications with:

- Heat templates to orchestrate the deployment the cloud applications, including Heat resources that define the metrics to collect and the alarms to trigger actions.
- Ceilometer and Gnocchi to collect the metrics (data) to support autoscaling of Heat applications.
- Aodh to provide the alarming service to drive the autoscaling actions.

Autoscaling: Heat / Ceilometer / Gnocchi / Aodh



Telemetry = metrics, alarms, and events

- Ceilometer collects **many** metrics
 - Collects metrics/measurements about the core OpenStack components, such as how much RAM has been consumed per hour for a VM. These measurements might be used for billing purposes and analytics.
 - Forwarded to Gnocchi
- Gnocchi is *metrics data as a service*
 - Gnocchi is a multi-tenant time series, metrics and resources database
 - Stores large amount of metrics, while being scalable, performant, and fault-tolerant
- Aodh is an alarming service
 - You can define a threshold that triggers an alarm
 - Supported thresholds (alarms):
 - Gnocchi_resources_threshold
 - Gnocchi_aggregation_by_metrics_threshold
 - Gnocchi_aggregation_by_resources_threshold

For more details:

<https://docs.openstack.org/ceilometer/rocky/contributor/architecture.html>

Autoscaling: Orchestration + Telemetry services

- Heat integrates with Ceilometer/Gnocchi/Aodh to provide autoscaling (up and down) of virtual machine instances for cloud composite applications (stacks)
- Defined with Heat template, using the following **resources**:
 - OS::Heat::AutoScalingGroup - defines **what** to scale
 - Includes OS::Nova::Server
 - OS::Heat::ScalingPolicy - defines **how many** to scale
 - OS::Aodh::Gnocchi...Alarm - defines alarms for **when** to scale
 - Possibly others for load balancing
- Manual scaling (up or down) supported through REST API calls
 - Uses *web hook* (URL) in **outputs** of Heat stack

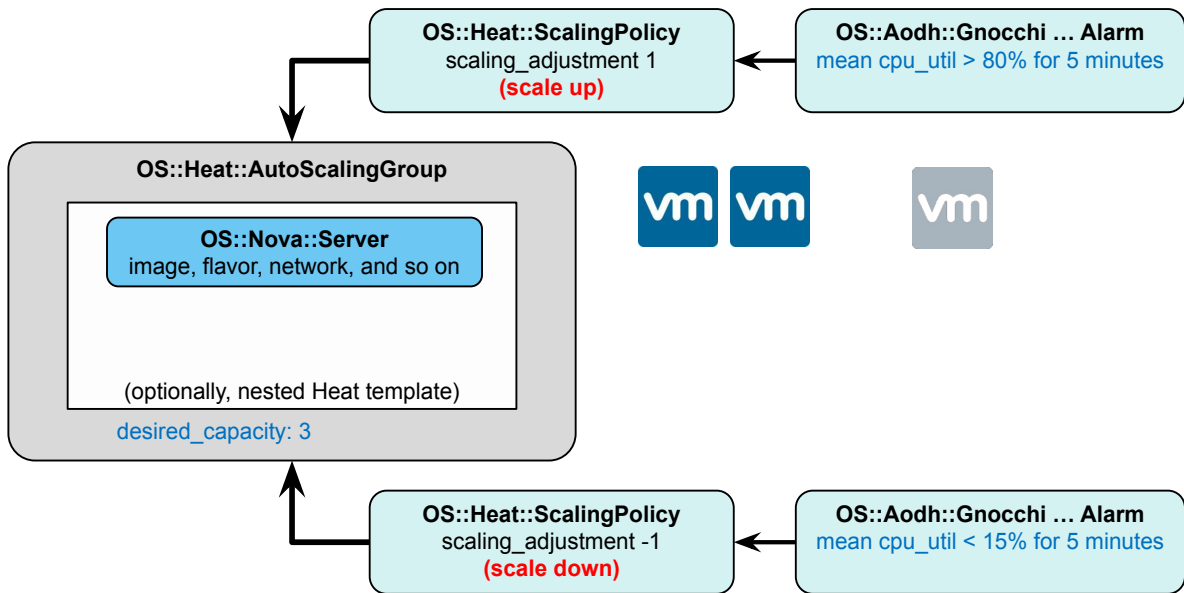
This slide shows the minimum Heat template resources that are required to implement auto-scaling:

- **OS::Heat::AutoScalingGroup** defines **what** resource to scale (typically a VM instance)
- **OS::Heat::ScalingPolicy** defines a count of **how** many resources (VM instances) to scale up or down
- **OS::Aodh::Gnocchi ... Alarm** defines **when** to generate an alarm; the alarm drives the scaling process
 - Note: There are several types of Gnocchi alarms
 - OS::Aodh::GnocchiAggregationByMetricsAlarm
 - OS::Aodh::GnocchiAggregationByResourcesAlarm
 - OS::Aodh::GnocchiResourcesAlarm

There might be other resources, such as a load balancer, included as part of the overall Heat template.

You can also manually scale the cloud application through REST API calls. There should be 2 *web hooks* (URLs) defined as *outputs* in your Heat template. One web hook is used to scale-up, the other to scale-down.

Autoscaling overview



This slide shows the relationship of the Heat resources for autoscaling of a virtual machine instance:

- **OS::Heat::AutoScalingGroup:** Defines the resource to auto-scale.
 - In this case, the **OS::Nova::Server** resource. Hint: This could be a nested Heat template.
- **OS::Heat::ScalingPolicy:** Defines the scaling (up or down) and how many resources to add or delete.
- **OS::Aodh::Gnocchi ... Alarm:** Defines the conditions (events) when autoscaling should be triggered.

Auto-scaling requires at least 1 AutoScalingGroup resource, 1 or more ScalingPolicy resources to scale up or down, and 1 or more Aodh Alarm resources.

In this case, the AutoScalingGroup defines a *desired capacity* of 3 VMs. The application starts with 3 VMs. Suppose an alarm is generated related to a cpu high condition. A fourth VM is deployed and added to the AutoScalingGroup. Next, suppose another alarm is generated for a cpu low condition. 1 of the VMs is destroyed. It might not be the same VM that was added when the scale up was executed.

For more information on the metrics collected:

<https://docs.openstack.org/ceilometer/latest/admin/telemetry-measurements.html>

Heat AutoScalingGroup

Answers the question:

What resource is
auto-scaled?

```
asg:
  type: OS::Heat::AutoScalingGroup
  properties:
    resource:
      type: OS::Nova::Server
      properties:
        image: { get_param: image_name }
        flavor: { get_param: flavor_type }
        availability_zone: nova
        networks:
          - network: { get_param: network_name }

  min_size: 1
  desired_capacity: 3
  max_size: 5
```

The **AutoScalingGroup** is a Heat resource that defines a minimum, maximum, and initial desired number of virtual machine instances for the composite application represented by the Heat template. In this case, the resource type is a Nova server (virtual machine instance). The **OS::Nova::Server** resource is defined within the AutoScalingGroup resource.

min_size: defines the minimum number of instances that can be provisioned. Auto-scaling cannot exceed this limit.

desired_capacity: defines the desired number of instances for the application. This matches the number of servers provisioned when the application is deployed.

max_size: defines the maximum number of instances that can be provisioned. Auto-scaling cannot exceed this limit.

Note: When coded as shown on the slide, the AutoScalingGroup can only scale 1 resource. In this case, a VM instance. To scale multiple resources, such as an instance and a LB pool member, you must define the resources in a separate template and nest it within the AutoScalingGroup.

Heat ScalingPolicy: Scale up and down

Answers the question:
How many resources are auto-scaled?

ServerScaleUpPolicy:

```
type: OS::Heat::ScalingPolicy
properties:
  adjustment_type: change_in_capacity
  auto_scaling_group_id: {get_resource: asg}
  cooldown: 300
  scaling_adjustment: 1
```

ServerScaleDownPolicy:

```
type: OS::Heat::ScalingPolicy
properties:
  adjustment_type: change_in_capacity
  auto_scaling_group_id: {get_resource: asg}
  cooldown: 300
  scaling_adjustment: -1
```

Each **OS::Heat::ScalingPolicy** resource sets a **signal_url** attribute with the web hook (URL) to use as action (scaling).

The action (signal_url) is defined in the **Aodh Alarm** resource. (see next 2 slides)



Copyright © 2019 Mirantis, Inc. All rights reserved

55

The **ScalingPolicy** is a Heat resource used to manage the scaling of an **AutoScalingGroup** resource. The **ScalingPolicy** resource defines the scaling to perform (adjust up or down and how many) for the referenced resource (in this case, the **asg** AutoScalingGroup resource from the previous slide).

adjustment_type defines when the autoscaling occurs:

- Change in capacity
- Exact match
- Percent change in capacity

cooldown defines a period between auto-scaling calls where no further auto-scaling can take place.

Aodh Alarm: Scale up

Answers the question:
When is auto-scaling
driven?

```
cpu_alarm_high:
  type: OS::Aodh::GnocchiAggregationByResourcesAlarm
  properties:
    description: scale up if the mean CPU > 80% for 5 minutes
    metric: cpu_util
    aggregation_method: mean
    granularity: 300
    evaluation_periods: 1
    threshold: 80
    resource_type: instance
    comparison_operator: gt
    alarm_actions:
      - str_replace:
          template: trust+url
          params:
            url: {get_attr: [ServerScaleUpPolicy, signal url]}
  query:
    str_replace:
      template: '{"=": {"server_group": "stack_id"}}'
      params:
        stack_id: {get_param: "OS::stack_id"}
```

The **OS::Aodh::GnocchiAggregationByResourcesAlarm** is a metric service (gnocchi) resource that defines the metering metric (for example, **cpu_util**) and condition (for example, if mean **cpu_util** > 80%) to be monitored.

An alarm is triggered when the threshold is exceeded (greater than). The action taken is determined by the **alarm_actions** property. In this example, the web hook (from the **signal_url** attribute of the **ServerScaleUpPolicy**) can be invoked to scale up.

Aodh Alarm: Scale down

Answers the question:
When is auto-scaling
driven?

cpu_alarm_low:

type: OS::Aodh::GnocchiAggregationByResourcesAlarm

properties:

description: scale down if the mean CPU < 5% for 5 minutes

metric: cpu_util

aggregation_method: mean

granularity: 300

evaluation_periods: 1

threshold: 5

resource_type: instance

comparison_operator: lt

alarm_actions:

- str_replace:

template: trust+url

params:

url: {get_attr: [ServerScaleDownPolicy, signal_url]}

query:

str_replace:

template: '{"=": {"server_group": "stack_id"}}'

params:

stack_id: {get_param: "OS::stack_id"}

Autoscaling outputs

```
outputs:
  server_list:
    description: >
      This is a list of server names that are part of the asg group.
    value: {get_attr: [asg, outputs_list, name]}

  server_ips:
    description: >
      This is a list of first IP addresses of the servers in the asg group
      for a specified network.
    value: {get_attr: [asg, outputs_list, networks, {get_param: network_name},
0]}
```

This slide shows example outputs to store the list of server (VM instance) names and IP addresses in the stack output.

Autoscaling outputs: Web hooks

`scale_up_url:`

description: >

This URL is the webhook to scale up the group. You can invoke the scale-up operation by doing an HTTP POST to this URL; no body nor extra headers are needed.

value: {get_attr: [`ServerScaleUpPolicy`, `signal_url`]}

`scale_dn_url:`

description: >

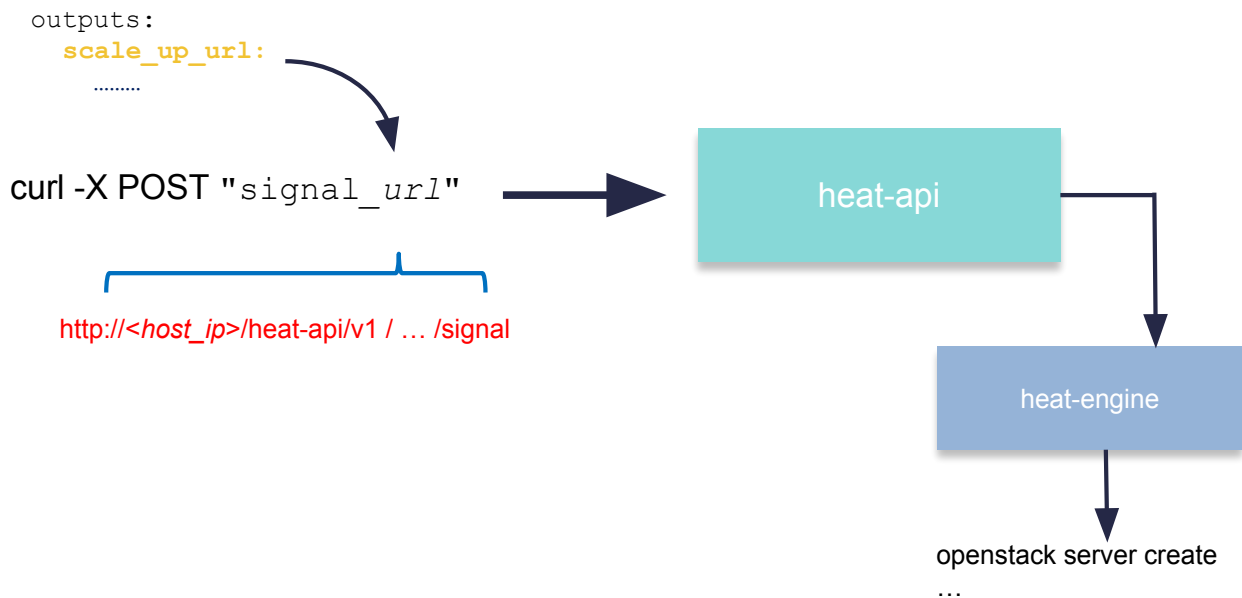
This URL is the webhook to scale down the group. You can invoke the scale-down operation by doing an HTTP POST to this URL; no body nor extra headers are needed.

value: {get_attr: [`ServerScaleDownPolicy`, `signal_url`]}

This slide shows example outputs for the autoscaling Heat template; to create the *Web hooks* for scaling up and scaling down.

To scale the Heat stack manually, simply create a REST API POST request using the **scale_up_url** or **scale_down_url** web hook (URL) stored in the stack outputs.

Overview of process to manually scale up



To manually scale a Heat stack, you must use the REST API to POST to a Web hook (for example, `scale_up_url` or `scale_down_url` outputs).

The request is routed to the **heat-api** process, defined by the `heat-api` endpoint in the service catalog..

In the case of a **scale-up** request, the REST API equivalent of an **openstack server create** request is issued to create another virtual machine instance.

Note: The `signal_url` must be enclosed within double quotes.

An example of a manual scale-up REST API request is on the next slide.

Manual scale up example

```
curl -X POST
"http://<host_ip>/heat-api/v1/c26147194d1243c5870f13b3a382002e/
stacks/LB-ASG/7b7b6885-388e-4949-afd5-707f555f9fdb/
resources/ServerScaleUpPolicy/signal"
```

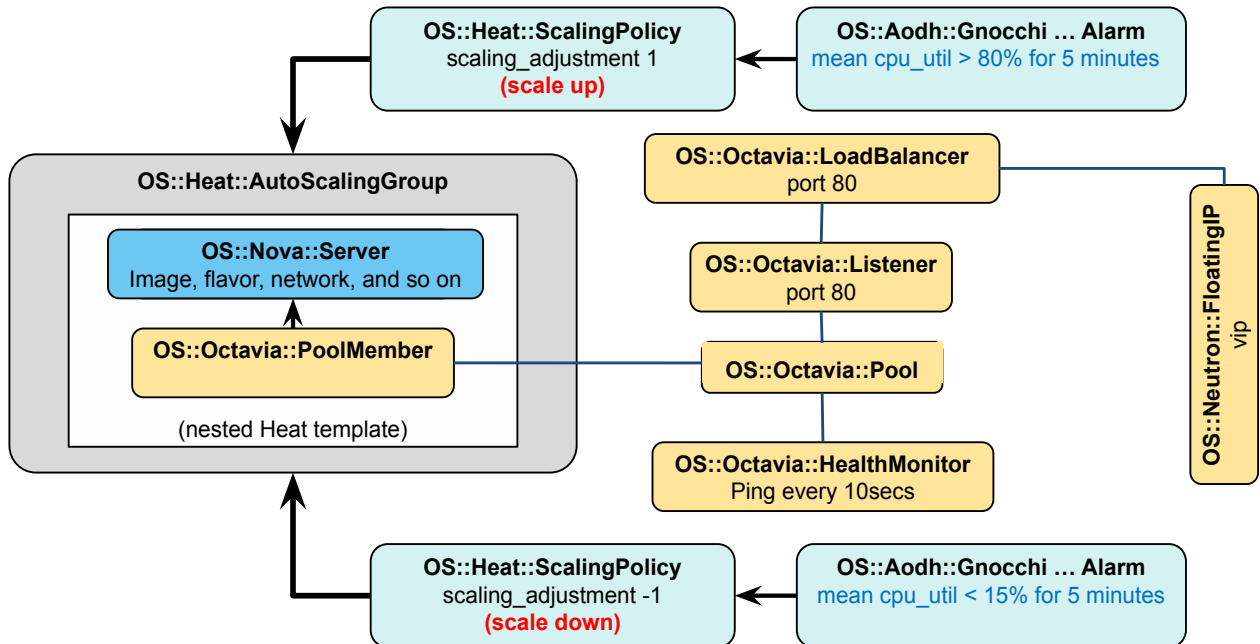
- HTTP POST to heat-api endpoint ...
`<host_ip>/heat-api/v1/c26147194d1243c5870f13b3a382002e/`
- ... to scale up ...
`resources/ServerScaleUpPolicy`
- ... the LB-ASG stack
`stacks/LB-ASG/7b7b6885-388e-4949-afd5-707f555f9fdb/`

This slide shows an example of manually scaling up using the **scale_up_url** from the stack output of the LB-ASG stack. Foreexample:
`http://172.31.29.98/heat-api/v1/c26147194d1243c5870f13b3a382002e/stacks/LB-ASG/7b7b6885-388e-4949-afd5-707f555f9fdb/resources/ServerScaleUpPolicy/signal`

Notes:

- (1) The user issuing the POST must have the **heat_stack_owner** role assigned to them
- (2) The request **MUST** be encapsulated within double quotes
- (3) If you receive the ***User is not authorized to perform action*** message:
 - check your **heat.conf** file and remove the **[ec2authtoken]** stanza and properties
 - check the quotation marks – if you paste the URL, you might have issues with the pasted character string. It is best to re-type them

Autoscaling plus load balancing



Up to this point, the discussion focused on autoscaling for the **OS::Nova::Server** resource. This slide expands beyond the simple autoscaling of a set of VMs. It illustrates a common deployment for an HTTP server application, assuming the autoscaling of the VMs, including a load balancer to front-end the HTTP server.

- The HTTP server application includes a load balancer defined for port 80.
- When a new VM instance is added (to scale up), it is added to the load balancer group (pool) due to the **PoolMember** resource also being defined as part of the autoscaling group (**OS::Heat::AutoScalingGroup**) resource.
- Typically, you also have a floating IP address associated with the load balancer. Users connect to the HTTP servers using the floating IP address. The load balancer spreads the HTTP traffic across the HTTP servers in the group.

For this example, there are **12 Heat template resources** defined. While that might seem daunting, you can find examples on github that can be used to create the Heat template.

Autoscaling plus load balancing: Heat template resources

- OS::Heat::AutoScalingGroup
 - OS::Nova::Server
 - OS::Octavia::PoolMember
- OS::Heat::ScalingPolicy
 - 1 for scale-up
 - 1 for scale-down
- OS::Aodh::GnocchiAggregationByResourcesAlarm
 - 1 for scale-up
 - 1 for scale-down
- OS::Octavia::LoadBalancer
- OS::Octavia::Listener
- OS::Octavia::Pool
- OS::Octavia::HealthMonitor
- OS::Neutron::FloatingIP

This slide summarizes the Heat template resources required for the previous slide. As each VM instance is created/deleted, it must be added to the load balancer pool with the PoolMember resource.

The following slides in this section contain a working example, broken down by resource type. They are provided as reference material. Your instructor might choose to present the slides.

LBaaS: AutoScalingGroup with nested resources

asg:

```
type: OS::Heat::AutoScalingGroup
properties:
  min_size: 1
  max_size: 5
  desired_capacity: 3
  resource:
    type: lb_server.yaml
    properties:
      flavor: {get_param: flavor}
      image: {get_param: image}
      network: {get_param: network}
      subnet: {get_param: subnet}
      pool_id: {get_resource: pool}
      metadata: {"metering.server_group": {get_param: "OS::stack_id"}}
```

Multiple resources requires use of a nested Heat template

Data required for resources in nested Heat template

Heat templates can be nested, as shown with the `lb_server.yaml` template. In this case, it is required because `lb_server` defines multiple resources to as members of the `AutoScalingGroup`.

There are 2 ways to reference the nested template:

- (1) Template file name: as shown on the slide. Using this method, you can not deploy the Heat stack from the Dashboard UI (Horizon). You can only use the CLI.
- (2) Template URL: `http://127.0.0.1:8091/lb_server.yaml`. Using this method, you can deploy the Heat stack from the CLI and the Dashboard.

The beginning number of resources (instances) is determined by the **desired_capacity** (3) property of the `AutoScalingGroup`.

The **metadata** property is used to set the **metering.server_group = ID of the Heat stack**. It is required and referenced in the query property in the Aodh alarms.

LBaaS: ScalingPolicy resources

ServerScaleUpPolicy:

```
type: OS::Heat::ScalingPolicy
properties:
  adjustment_type: change_in_capacity
  auto_scaling_group_id: {get_resource: asg}
  cooldown: 300
  scaling_adjustment: 1
```

Scale-up 1 instance each time

ServerScaleDownPolicy:

```
type: OS::Heat::ScalingPolicy
properties:
  adjustment_type: change_in_capacity
  auto_scaling_group_id: {get_resource: asg}
  cooldown: 300
  scaling_adjustment: -1
```

Scale-down 1 instance each time

The ScalingPolicy resources define how many resources (instances) to add or remove from the Heat stack when an alarm is exceeded. Example alarms are discussed in the next 2 slides.

LBaaS: Aodh alarm to scale-up

```
cpu_alarm_high:
  type: OS::Aodh::GnocchiAggregationByResourcesAlarm
  properties:
    description: scale up if the mean CPU > 80% for 1 minutes
    metric: cpu_util
    aggregation_method: mean
    granularity: 60
    evaluation_periods: 1
    threshold: 80
    resource_type: instance
    comparison_operator: gt
  alarm_actions:
    - str_replace:
        template: trust+url
        params:
          url: {get_attr: [ServerScaleUpPolicy, signal_url]}
  query:
    str_replace:
      template: '{"=": {"server_group": "stack_id"}}'
      params:
        stack_id: {get_param: "OS::stack_id"}
```

Use **signal_url** from
ScaleUpPolicy

This alarm example will scale up based on the **ServerScaleUpPolicy** (add an instance) when the (mean) aggregation of CPU across all instances for the stack is GT 80% over the last minute. The **signal_url** is used to drive the heat engine for the scaling.

LBaaS: Aodh alarm to scale-down

```
cpu_alarm_low:
  type: OS::Aodh::GnocchiAggregationByResourcesAlarm
  properties:
    description: scale down if the mean CPU < 5% for 5 minutes
    metric: cpu_util
    aggregation_method: mean
    granularity: 300
    evaluation_periods: 1
    threshold: 5
    resource_type: instance
    comparison_operator: lt
  alarm_actions:
    - str_replace:
        template: trust+url
        params:
          url: {get_attr: [ServerScaleDownPolicy, signal_url]}
  query:
    str_replace:
      template: '{"=": {"server_group": "stack_id"}}'
      params:
        stack_id: {get_param: "OS::stack_id"}
```

Use **signal_url** from
ScaleDownPolicy

This alarm example will scale down based on the **ServerScaleDownPolicy** (remove an instance) when the (mean) aggregation of CPU across all instances for the stack is LT 5% over the last 5 minutes. The **signal_url** is used to drive the heat engine for the scaling.

LBaaS: Octavia LB resources (1)

```
lb:
  type: OS::Octavia::LoadBalancer
  properties:
    vip_subnet: {get_param: subnet}

listener:
  type: OS::Octavia::Listener
  properties:
    loadbalancer: {get_resource: lb}
    protocol: HTTP
    protocol_port: 80

pool:
  type: OS::Octavia::Pool
  properties:
    listener: {get_resource: listener}
    lb_algorithm: ROUND_ROBIN
    protocol: HTTP
    session_persistence:
      type: SOURCE_IP
```

Still part of the Heat template ... this slide shows example Octavia resources to implement a v2 load balancer. The load balancer uses port 80 (HTTP). Requests are routed based on the ROUND_ROBIN method.

LBaaS: Octavia LB resources (2)

```
lb_monitor:
  type: OS::Octavia::HealthMonitor
  properties:
    pool: { get_resource: pool }
    type: TCP
    delay: 5
    max_retries: 5
    timeout: 5

# assign a floating ip address to the load balancer pool.
lb_floating:
  type: OS::Neutron::FloatingIP
  properties:
    floating_network_id: {get_param: external_network_id}
    port_id: {get_attr: [lb, vip_port_id]}
```

Still part of the Heat template ... the load balancer is monitored every 5 seconds. The last resource that must be defined is a floating IP address. The floating IP is associated with the VIP port of the load balancer.

lb_server template: resources

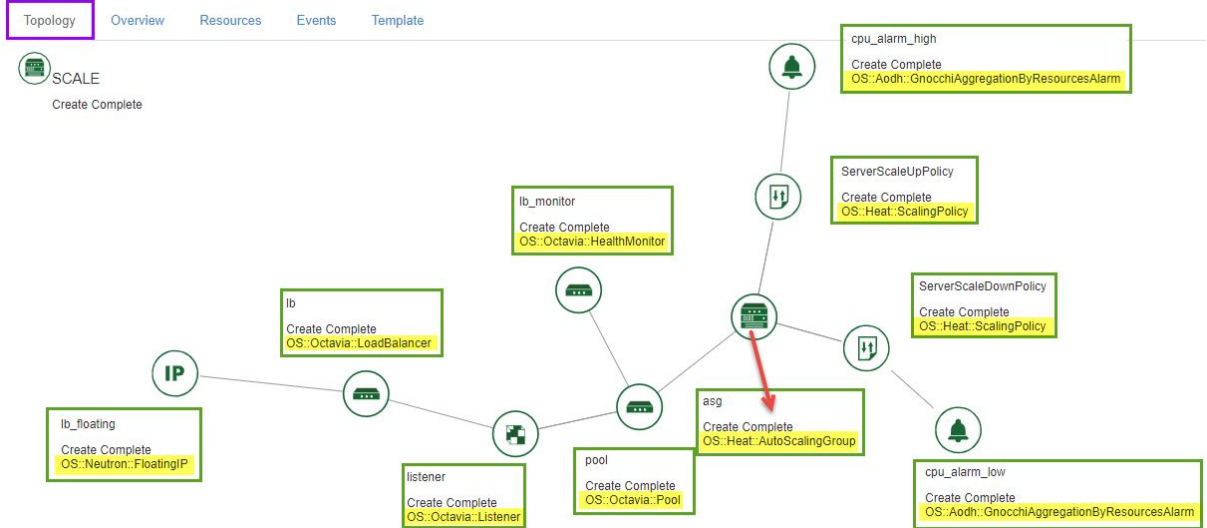
```
server:
  type: OS::Nova::Server
  properties:
    flavor: {get_param: flavor}
    image: {get_param: image}
    networks: [{network: {get_param: network} }]

member:
  type: OS::Octavia::PoolMember
  properties:
    pool: {get_param: pool_id}
    address: {get_attr: [server, first_address]}
    protocol_port: 80
    subnet: {get_param: subnet}
```

This slide shows the resources from the **lb_server.yaml** template, nested in the AutoScalingGroup resource. As each instance is added/removed, it must also be added/removed from the load balancer pool. Since the AutoScalingGroup resource does not explicitly support multiple resources, you can nest this template to implicitly support multiple resources.

UI: Stack topology

SCALE



71

This slide shows the Heat stack topology for the load balanced / autoscaled stack discussed over the previous slides. Compare the stack topology with the Heat template.

The nested template, `lb_server.yaml`, defines the resources for the AutoScalingGroup. Those resources are not shown in the stack topology. To view the autoscaled resources, you must *drill-down* on the AutoScalingGroup. At that point, you see the VM instances.

UI: Stack events

SCALE

Check Stack ▾

Topology Overview Resources **Events** Template

Displaying 20 items | Next »

VMs over time

Stack Resource	Resource	Time Since Event	Status	Status Reason
ServerScaleUpPolicy	77bb55e5dd92454295828d96ed10d3a6	0 minutes	Signal Com	alarm state changed from ok to alarm (Transition to alarm due to 1 samples outside threshold, most recent: 91.6147527992)
ServerScaleUpPolicy	77bb55e5dd92454295828d96ed10d3a6	14 minutes	Signal Com	alarm state changed from ok to alarm (Transition to alarm due to 1 samples outside threshold, most recent: 99.0865271946)
ServerScaleUpPolicy	77bb55e5dd92454295828d96ed10d3a6	31 minutes	Signal Com	alarm state changed from ok to alarm (Transition to alarm due to 1 samples outside threshold, most recent: 100.0)
ServerScaleDownPolicy	6db9e330fc2440dfa3ccdefefa7c0bb6	1 hour, 15 minutes	Signal Com	alarm state changed from alarm to alarm (Remaining as alarm due to 5 samples outside threshold, most recent: 0.1536429361)
ServerScaleDownPolicy	6db9e330fc2440dfa3ccdefefa7c0bb6	1 hour, 21 minutes	Signal Com	alarm state changed from alarm to alarm (Remaining as alarm due to 5 samples outside threshold, most recent: 0.154425630683)
SCALE	c9a9900b-e6d1-4744-8682-9226c1fb232e	1 hour, 26 minutes	Create Complete	Stack CREATE completed successfully
SCALE	c9a9900b-e6d1-4744-8682-9226c1fb232e	1 hour	Create	

This slide shows an example of the UI where several events have been generated based on the **OS::Aodh::GnocchiAggregationByResourcesAlarm** resources (cpu_alarm_high and cpu_alarm_low).

The stack started with 3 instances. Then:

- A **cpu_alarm_low** event scaled the stack down by an instance, leaving 2 instances running.
- Another **cpu_alarm_low** event scaled the stack down by another instance, leaving a single instance running.
 - However, due to the **min_size** property value (1) on the AutoScalingGroup, no more instances can be removed.
- A **cpu_alarm_high** event scaled the stack up by an instance, leaving 2 instances running.
- Another **cpu_alarm_high** event scaled the stack up by an instance, leaving 3 instances running.
- Another **cpu_alarm_high** event scaled the stack up by an instance, leaving 4 instances running.

Summary

Summary

You should now be able to:

- Explain the 4 primary sections of a Heat template
- Understand how to define input parameters, including constraints
- Explain how to check the syntax of your Heat template
- Understand where to look for documents that help
- Briefly describe how to implement autoscaling

Lab exercises

Lab 10: Orchestration service (Heat)