Occopus

Release v1.10

József Kovács, Zoltán Farkas, Márk Emődi

Sep 05, 2022

USER GUIDE

This document is envisaged to introduce the abilities of Occopus, a cloud orchestrator framework developed at SZTAKI (Hungary) to create and manage flexible computing infrastructures and services in a single or multi cloud system.

How to start?

Get started in minutes. Follow the *Installation* guide!

CHAPTER
ONE

WHAT IS OCCOPUS?

Occopus is an easy-to-use hybrid cloud orchestration tool. It is a framework that provides features for configuring and orchestrating distributed applications (so called virtual infrastructures) on single or multi cloud systems. Occopus can be used by application developers and devops to create and deploy complex virtual infrastructures as well as to manage them at deployment time and at runtime.

If you use Occopus, please cite at least one of the following publications:

- Kovács, J. & Kacsuk, P. Occopus: a Multi-Cloud Orchestrator to Deploy and Manage Complex Scientific Infrastructures J Grid Computing (2018) 16: 19. https://doi.org/10.1007/s10723-017-9421-3
- Lovas, R ; Nagy, E ; Kovacs, J Cloud agnostic Big Data platform focusing on scalability and cost-efficiency ADVANCES IN ENGINEERING SOFTWARE 125 pp. 167-177. , 11 p. (2018) http://dx.doi.org/10.1016%2Fj.advengsoft.2018.05.002
- József Kovács, Péter Kacsuk, Márk Emődi, Deploying Docker Swarm cluster on hybrid clouds using Occopus, Advances in Engineering Software, Volume 125, 2018, Pages 136-145, ISSN 0965-9978, https://doi.org/10.1016/j.advengsoft.2018.08.001.
- Kacsuk, P., Kovács, J. & Farkas, Z. The Flowbster Cloud-Oriented Workflow System to Process Large Scientific Data Sets J Grid Computing (2018) 16: 55. https://doi.org/10.1007/s10723-017-9420-4
- Lovas, R ; Farkas, A ; Marosi, A Cs ; Acs, S ; Kovacs, J ; Szaloki, A ; Kadar, B Orchestrated Platform for Cyber-Physical Systems COMPLEXITY 2018 pp. 1-16. Paper: 8281079 , 16 p. (2018) http://dx.doi.org/10.1155%2F2018%2F8281079

CHAPTER

TWO

ACKNOWLEDGEMENT

This development has been supported by the National Development Agency, Hungary under contract no. KMR12-1-2012-0119 (CLAKK); by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 283481 (SCI-BUS) and 608886 (CloudSME); and by the European Union Horizon 2020 research and innovation programme under grant agreement no. 731574 (COLA) and no. 768892 (CloudiFacturing). We thank for the usage of MTA Cloud that significantly helped us achieve the results published here.

2.1 Concept

Occopus is an easy-to-use hybrid cloud orchestration tool. It is a framework that provides features for configuring and orchestrating distributed applications (so called virtual infrastructures) on single or multi cloud systems. Occopus can be used by application developers and devops to create and deploy complex virtual infrastructures as well as to manage them at deployment time and at runtime.

It has many similarities with HEAT, Juju, etc. Its main advantage that it is cloud technology neutral, open source and easily extendable. Occopus works based on an infrastructure description that describes the services to be deployed in the cloud (or other type of resource) and the order of their deployment. Occopus deploys the services in the cloud according to deployment order specified in the description.

Occopus not only deploys the services but checks their availability, accessibility i.e. their health before deploying the next service. Furthermore, the description can contain contextualisation information for every deployable service and based on that information Occopus carries out contextualisation for the deployed services. As a result after contextualisation the services can call each other, i.e. they can collaborate to realize a higher level service called as virtual infrastructure.

Occopus can be used in three different ways:

1. Desktop software (i.e. as command line util): In this case virtual infrastructure developers can run Occopus on their desktop machine and they give the infrastructure description as input to Occopus together with their credentials to the target cloud where they want to deploy theinfrastructure. Based on the description Occopus deploys and activates the infrastructure in the cloud and then exits. Then any potential user can use the infrastructure that were built by Occopus in the cloud(s).
2. REST API: Occopus can expose its functionalities through a web service with RESTful interface. The functionalities like deployment, management, destroy, etc. can be realized through REST calls.
3. Library providing the Occopus API: In this case infrastructure developers can create a program that deploys the infrastructure in the cloud by calling the Occopus APIs. APIs provide a more fine-tuned controlling of the deployment, and management process while further functionalities can be added to the extendable Occopus architecture.

2.2 Features

Wide range of supported resources

Occopus supports a wide variety of interfaces for allocating cloud or container resources. Some of the most widespread supported interfaces are ec2, nova, docker and Azure. The list of supported interfaces is growing due to the modular and extensible feature of Occopus.

List of supported technologies

Hybrid cloud support

Occopus can deploy your infrastucture by allocating each node on a different cloud. Multiple type of clouds can be used within the same infrastructure, therefore we call it hybrid cloud support. It is also possible to let Occopus dynamically choose between a given list of clouds at deployment time. To use multiple resources, nodes must have multiple implementations and optionally implementations can be filtered in the infrastructure description.

Resource section of node definition

Infrastructure descriptions

Multiple configuration management support

Occopus can utilise Chef, Cloud-init, pre-defined images, or any combination of these in the same infrastructure. Occopus has access to various configuration and contextualization methods:

- You can use pre-defined images in the resource section of the node definition
  - *Resource section of node definition*
- You can use cloud-init as a contextualisation tool
  - *Contextualisation section of node definition*
- Occopus supports configuration management tools. Currently, Chef is supported.
  - *Using Chef in node definition*
  - *Demo infrastructures using chef*

Different usage possibilities

Occopus functionalities can be utilised in different ways. First, it provides command-line tools exposing different functionalities to build, query, update or destroy infrastructures. Second, REST API can be used after Occopus web service has been launched. Finally, orchestration functionalities can be utilised in your application through the API of the Occopus python library.

Overview of usage possibilities

Command-line interface

REST interface

Python API

Simple YAML format

Occopus uses YAML files for node definitions and infrastructure descriptions, making them simple, human-readable and easy-to-learn.

Node definitions

Infrastructure descriptions

Schema checking

Both, infra description and node definition files are analysed and validated by Occopus by searching for missing or invalid sections and attributes. This helps users to avoid creating syntactically wrong descriptor files.

Dynamic reconfiguration

Make changes to your infrastructures on the fly with a single command. Modify the infrastructure description file the way you want - add and remove nodes, set new paramateres (e.g. scaling) and variables for your nodes and your infrastructure and modify the dependency graph as you wish. After updating the infrastructure description in the datastore using *occopus-build command*, Occopus will automatically reconfigure the nodes of running infrastructures to match the new definitions.

Health checking

Occopus supports health checking primitives to check if a node is still operating. These primitives are network availability of the node (ping), checking the connectivity of a certain port (port access), checking the responsiveness of a web service (url checking) or checking the connectivity of a mysql database (mysql access).

Health checking primitives

Auto healing

Occopus monitors the states of the nodes by applying the primitives configured by health-checking for each node. Once, a node does not fails on health-checking, it is considered as fail node, Occopus destroys and rebuilds it.

Manual scaling

Scaling up or down any nodes in the infrastructure is supported. Occopus can launch multiple instances of a certain node, however the infrastructure itself must be built in a way to handle scaling events.

Scaling commands

Scaling limits in node description

Multiple node implementations

Occopus supports defining multiple implementations for a node (type) and utilise different backends, images, tools and variables in them. You can filter the available implementations in the infrastructure description, and occopus will select an implementation from the remaining ones.

Multiple node implementations

Node type filtering in infrastructure description

Multiple authenticators

Occopus can handle multiple authenticators during building an infrastructure on multiple resource. Multiple resources may have different authenticators and authentication procedures. Occopus supports defining authenticators and selecting one of them for a certain resource. The selection can be based on any parameter of a resource handler, including name, type, image-id, etc.

Description of authentication

Extensible architecture

Occopus was created with extensibility and flexibility in mind - New modules for resource-handlers, configuration-managers, additional schema-checker rules or health-checking primitives can easily be implemented and added without modifying other components.

2.3 Supported Resources

Occopus has an extendible, pluginable architecture for interfacing external tools and services. The actual version contains four different resource plugin implementations for handling clouds and one for docker containers.

2.3.1 EC2

Occopus can utilise cloud resources supporting the Amazon Elastic Compute Cloud (EC2) interface.

2.3.2 Nova

Occopus has a resource plugin to interface with NOVA API. With this interface OpenStack type cloud systems can be utilised.

2.3.3 Azure

The Azure and Azure ACI resource plugins of Occopus enables the usage of Azure resources.

2.3.4 CloudBroker

This is a special resource plugin serving resource allocation and program execution on CloudBroker platform operated by CloudBroker Inc..

Using the CloudBroker plugin you can access all the different cloud types that are supported by CloudBroker platform. These are:

- Amazon
- CloudSigma
- OpenStack
- OpenNebula

If you want to use clouds via the CloudBroker platform, please, contact the CloudBroker GmbH:

- Email: info@cloudbroker.com
- Web: http://www.cloudbroker.com

2.3.5 Docker

Occopus has a resource plugin which enables to utilise pure Docker or Swarm resources. With this plugin it is possible to deploy containers and to combine them into an infrastructure.

2.3.6 CloudSigma

The CloudSigma resource plugin of Occopus enables the usage of CloudSigma resources.

2.4 Setup

2.4.1 Installation

Important: We primarily support **Ubuntu** operating system. The following instruction steps were tested on **Ubuntu 20.04** version.

1. Install a few system-wide packages

   Python 3.x, Virtualenv, Redis server for data storage and SSL devel lib for Chef to work

   ```
   sudo apt update && \
   sudo apt install -y python3-pip python3-dev virtualenv redis-server libssl-dev
   ```

2. Prepare the environment (you may skip this part to have a system-wide installation, not recommended)

   ```
   virtualenv -p python3 $HOME/occopus
   source $HOME/occopus/bin/activate
   ```

3. Deploy all Occopus packages

   ```
   pip install --no-index --find-links https://pip3.lpds.sztaki.hu/packages OCCO_API
   ```

   Now, all Occopus packages are deployed under your virtualenv `occopus`.

4. Optionally, copy your certs under Occopus if you plan to use VOMS authentication against Nova resources

   ```
   cat /etc/grid-security/certificates/*.pem >> $(python -m requests.certs)
   ```

Note: Do not forget to activate your virtualenv before usage!

Note: Please, proceed to the next chapter to continue with configuration!

2.4.2 Configuration

Occopus requires one configuration file containing static parameters and objects to be instantiated when Occopus starts. The file is occopus_config.yaml.

This file must be specified for Occopus through command line parameters. Alternatively, we recommend to store this file in $HOME/.occopus directory, so that Occopus will automatically find and use it.

Please, download and save your configuration file:

```
mkdir -p $HOME/.occopus
curl https://raw.githubusercontent.com/occopus/docs/devel/tutorials/.occopus/occopus_
→config.yaml -o $HOME/.occopus/occopus_config.yaml
```

Occopus uses YAML as a configuration language, mainly for its dynamic properties, and its human readability. The parsed configuration is a dictionary, containing both static parameters and objects instantiated by the YAML parser.

Note: Please, do not modify the configuration file unless you know what you are doing!

Note: Please, proceed to the next chapter to continue with setting up authentication information!

2.4.3 Authentication

Authentication file

In order to get access to a resource, Occopus requires your credentials to be defined. For this purpose you have to create a file, auth_data.yaml containing authentication information for each target resource in a structured way.

Once you have your auth_data.yaml file, you must specify it as command line argument for Occopus. A more convenient (recommended) way is to save this file at $HOME/.occopus/auth_data.yaml so that Occopus will automatically find and use it.

You can download and save your initial authentication file:

```
mkdir -p $HOME/.occopus
curl https://raw.githubusercontent.com/occopus/docs/devel/tutorials/.occopus/auth_data.
→yaml -o $HOME/.occopus/auth_data.yaml
```

Once you have your initial authentication file, edit and insert your credentials to the appropriate section.

For each different type of resources, you may specify different authentication information, which must fit to the format required by the resource plugin defined by the type keyword. Here are the formats for the different resource types.

Authentication data formats

For EC2 resources:

```
resource:
    -
        type: ec2
        auth_data:
            accesskey: your_access_key
            secretkey: your_secret_key
```

For nova resources:

In case of username/password authentication:

```
resource:
    -
        type: nova
        auth_data:
            username: your_username
            password: your_password
```

In case of application credential based authentication:

```
resource:
    -
        type: nova
        auth_data:
            type: application_credential
            id: id_of_the_app_cred
            secret: password_of_the_app_cred
```

In case of VOMS proxy authentication:

```
resource:
    -
        type: nova
        auth_data:
            type: voms
            proxy: path_to_your_x509_voms_proxy_file
```

For azure resources:

```
resource:
    -
        type: azure_vm
        auth_data:
            tenant_id: your_tenant_id
            client_id: your_client_id
            client_secret: your_client_secret
            subscription_id: your_subscription_id
```

Please consult the Azure Documentation on how to obtain the necessary tenant_id, client_id, client_secret and subscription_id values, and how to gain proper access for being able to manage Azure virtual machines and associated resources.

For azure_aci resources:

```
resource:
    -
        type: azure_aci
        auth_data:
            tenant_id: your_tenant_id
            client_id: your_client_id
            client_secret: your_client_secret
            subscription_id: your_subscription_id
```

Please consult the Azure Documentation on how to obtain the necessary tenant_id, client_id, client_secret and subscription_id values, and how to gain proper access for being able to manage Azure continer instances and associated resources.

For cloudbroker resources:

```
resource:
    -
        type: cloudbroker
        auth_data:
            email: your@email.com
            password: your_password
```

For cloudsigma resources:

```
resource:
    -
        type: cloudsigma
        auth_data:
            email: your@email.com
            password: your_password
```

For chef config managers:

```
config_management:
    -
        type: chef
        auth_data:
            client_name: name_of_user_on_chef_server
            client_key: !text_import
                url: file://path_to_the_pem_file_of_cert_for_user
```

The values for client_name and client_key attributes must be the name of the **user** that can login to the Chef server and the public key of that Chef user. This user and its key will be used by Occopus to register the infrastructure before deployment of nodes starts. As the example shows above, the key can be imported from a separate file, so the path to the **pem** file is enough to be specified in the last line.

For multiple resource types:

```
resource:
    -
        type: ec2
        auth_data:
            accesskey: your_access_key
            secretkey: your_secret_key
    -
        type: nova
        auth_data:
            type: voms
            proxy: path_to_your_voms_proxy_file
```

For multiple resources with different endpoints:

```
resource:
    -
        type: ec2
        endpoint: my_ec2_endpoint_A
```

(continues on next page)

(continued from previous page)

```
        auth_data:
            accesskey: your_access_key_for_A
            secretkey: your_secret_key_for_A
    -
        type: ec2
        endpoint: my_ec2_endpoint_B
        auth_data:
            accesskey: your_access_key_for_B
            secretkey: your_secret_key_for_B
```

Note: The authentication file has YAML format. Make sure you are using spaces instead of tabulators for indentation!

2.5 Composing an infrastructure

In order to deploy an infrastructure, Occopus requires

1. description of the infrastructure
2. definition of the individual nodes

The following section explains how the various descriptions must be formatted.

2.5.1 Infrastructure Description

Dependency graph on *Node Description*-s.

The graph contains the following information:

`user_id` The identifier of the owner of the infrastructure instance.

`infra_name` The name of the infrastructure.

`nodes` List of node.

`dependencies` List of edge definitions. Each of these can be either

- A pair (2-list) of node references.
- A mapping containing:

  `connection` The pair (2-list) of node references.

  `mappings` List of attribute mappings. Each mapping can be a pair (2-list) of strings (attribute specifications, dotted strings permitted) or a mapping containing:

  `attributes` The pair of attribute specifications.

  `synch` Whether to synchronize on the availability of the source attribute.

  `**` Anything else that is required by mediating services.

  `**` Anything else that is required by mediating services.

`variables`

Arbitrary mapping containing infrastructure-wide information. This information is static (not parsed anywhere). Nodes will inherit these variables, but they may also override them.

The following example describes a two nodes infrastructure where B depends on A, i.e. B uses the service provided by A.

```
user_id: me
infra_name: simple
nodes:
    - &A
        name: A
        type: mysql
    - &B
        name: B
        type: wordpress
dependencies:
    - [ *B, *A ]
```

2.5.2 Node Description

Abstract description of a node, which identifies a type of node a user may include in an infrastructure. It is an abstract, *resource-independent* definition of a class of nodes and can be stored in a repository.

This data structure does *not* contain information on how it can be instantiated. It rather contains *what* needs to be instantiated, and under what *conditions*. It refers to one or more *implementations* that can be used to instantiate the node. These implementations are described with *node definition* data structures.

To instantiate a node, its implementations are gathered first. Then, they are filtered and one is selected by Occopus randomly.

name Name of node which uniquely identifies the node inside the infrastructure.

type The type of the node i.e. the node definition to be used when intantiating the node. If node definition exists for 'XXX' then use "type: XXX" to instantiate the implementation of node 'XXX'.

filter (dict)

```
filter:
   type: ec2
   regionname: ROOT
   instance_type: m1.small
```

Optional. Provides filtering among the available implementations of a node definition specified for 'type'. The dictionary must define key-value pairs where keywords are originated from resource section of the node definitions. If unspecified or filtering results more than one implementations, one will be chosen by Occopus.

scaling (dict)

```
scaling:
   min: 1
   max: 3
```

Optional. Keywords for scaling are ''min" and ''max". They specify how many instances of the node can have minimum (''min") and maximum (''max") in the infrastructure. At startup ''min" number of instances of the node will be created. Default and minimal value for ''min" is 1. Default value for ''max" equals to ''min". Both values are hardlimits, no modification of these limits are possible during infrastructure maintenance.

variables Arbitrary mapping containing static node-level information:

1. Inherited from the infrastructure.
2. Overridden/specified in the node's description in the infrastructure description.

The final list of variables is assembled by Occopus.

2.5.3 Node Definition

Describes an *implementation* of a *node*, a template that is required to instantiate a node.

A node definition consists of 4 different sections:

1. `resource` Contains the definition of the resource and its attributes, like endpoint, image id, etc. The attributes to be defined are resource type dependent. There are 5 different resource plugins as mentioned in the *Supported Resources* section, each one handles its own required and optional attributes. Possible attributes are defined in the *Resource section*.
2. `contextualisation` Optional. Contains contextualisation information for the node to be instantiated. Possible attributes are defined in the *Contextualisation section*.
3. `config_management` Optional. Describes the configuration manager to be used and its required parameters. Currently, chef and puppet are supported. Possible attributes are defined in the *Config management section*.
4. `health_check` Optional. Can be specified if health of the node can be monitored. Default is ping to check network access. Possible attributes are defined in the *Health check section*.

2.5.3.1 Resource

In this section, the attributes (keywords) are listed and explained which can be used for the different resource handlers.

EC2

`type: ec2` Selects the ec2 resource handler.

`endpoint` The endpoint (url) of the ec2 cloud interface.

`regionname` Region name of for the ec2 cloud interface.

`image_id` The identifier of the image behind the ec2 cloud to be instantiated to realize a virtual machine.

`instance_type` The instance type determines the characteristics (CPU, memory, storage, networking) of the VM created (e.g. m1.small).

`key_name` Optional. The name of the keypair to assign to the allocated virtual machine.

`security_group_ids` Optional. The list of security group IDs which should be assigned to the allocated virtual machine.

`subnet_id` Optional. The ID of the subnet which should be assigned to the allocated virtual machine.

`tags` Optional. List of key-value pairs of tags to be registered for the virtual machine.

`name` Optional. A user-defined name for this resource. Used in logging and can be referred to in the *authentication file* to distinguish authentication to be applied among resources having the same type.

Nova

type: nova Selects the nova resource handler.

endpoint The endpoint (URL) of the OpenStack Identity API service. If the URL includes the API version (e.g. `https://foo.bar:5000/v3/`), then the given API version will be used, otherwise API v3 will be assumed as default.

project_id Specifies the ID of the project to connect to.

region_name Optional. Specifies the name of the region within the project.

user_domain_name Optional. Specifies the name of the user domain. The default value of this attribute is "Default".

network_id Optional. Specifies the ID of the network to attach to the virtual machine.

image_id The identifier of the image on the cloud to be instantiated to realize a virtual machine.

flavor_name The type of flavor to be instantiated through nova when realizing this virtual machine. This value refers to a flavour of the nova cloud. It determines the resources (CPU, memory, storage, networking) of the node.

volume_size Optional. When set, can be used to tell the `nova` plugin to create a volume from the image specified, and boot the VM from the volume created. Value `0` makes OpenStack create a volume automatically, other values can be used to specify the desired volume size.

volume_persist Optional. Values `True` or `true` tell Occopus to keep the volume of the VM after it has been terminated. The default value of this attribute is `false`.

server_name Optional. The hostname of the instantiated virtual machine.

key_name Optional. The name of the keypair to be associated to the instance.

security_groups Optional. List of security groups to be associated to the instance.

floating_ip Optional. If defined (with any value), new floating IP address will be allocated and assigned for the instance.

floating_ip_pool Optional. If defined, also implies **floating_ip**, and specifies the name of the floating IP pool that should be used to allocate a new floating IP for the VM.

name Optional. A user-defined name for this resource. Used in logging and can be referred to in the *authentication file* to distinguish authentication to be applied among resources having the same type.

tenant_name Deprecated. A container used to group or isolate resources on the cloud behind the nova interface. If this option is not specified, **project_id** and **user_domain_name** must be set.

Azure

type: azure_vm Selects the Azure resource handler.

endpoint The endpoint (url) of the Azure interface, e.g. https://management.azure.com

resource_group The resource group to allocate Azure resources in. You can use a new resource group, or an existing one. The list of existing resource groups can be queried from the Azure Portal.

location The location where the resources should be allocated, e.g. francecentral. The Azure command line client can be used to query the list of usable location: `az account list-locations -o table` (for usable names, see the "Name" row of the table).

vm_size The size of the VM to allocate, e.g. Standard_DS1_v2. The Azure command line client can use used to query the list of available VM sizes: `az vm list-sizes --location <location>`, enter the value of the "name" key for the desired VM size in the list.

publisher The image publisher's name, e.g. Canonical. One can use the Azure command line client to get the list of images: `az vm image list`. The output of this command contains the values which should be used for `publisher`, `offer`, `sku` and `version`.

offer The published name of the image, e.g. UbuntuServer.

sku The type of the OS, e.g. 18.04.0-LTS.

version The version of the image to use, e.g. latest.

username The name of the admin user to create on the VM. Azure currently has the following restrictions on the username: must only contain letters, numbers, hyphens, and underscores and may not start with a hyphen or number, must not include reserved word, is between 1 and 64 characters long.

password Optional. The password for the admin user. If not set, then `ssh_key_data` must be defined.

ssh_key_data Optional. The public part of the SSH key for the admin user. The content specified here will be placed into the admin user's authorized_keys file. If not set, then `password` must be defined.

server_name Optional. The hostname of the instantiated virtual machine.

vnet_name Optional. Name of the virtual network to use for the VM. If not specified, the Azure resource plugin will allocate a virtual network.

nic_name Optional. The name of the network interface to use for the VM. If not specified, the Azure resource plugin will allocate a network interface.

subnet_name Optional. The name of the subnet to use for the VM. If not specified, the Azure resource plugin will allocate a subnet.

public_ip_needed Optional. If specified with the value True, the Azure resource plugin will allocate a public IP address for the VM.

Azure ACI

type: azure_aci Selects the Azure ACI (Azure Container Instances) resource handler.

endpoint The endpoint (url) of the Azure interface, e.g. https://management.azure.com

resource_group The resource group to allocate Azure resources in.

location The location where the resources should be allocated, e.g. francecentral.

image The public image to be used from Docker Hub, e.g. bde2020/spark-worker:2.4.5-hadoop2.7.

network_type The type of network to be used. Value "public" allocates a public address for the container, whereas value "private" uses a private network. When the value "public" is specified, then Occopus will allocate an FQDN for the container, which will also be set as the environment variable `_OCCOPUS_ALLOCATED_FQDN`.

memory The memory in GB to allocate for the container, e.g. 2.

cpu_cores The number of vCPU cores to allocate for the container, e.g. 4.

os_type The operating system type required by the container. Possible values are "linux" and "windows".

gpu_type Optional. Specifies the GPU type to be allocated for the container. Currently usable values are "K80", "P100" and "V100".

gpu_count Optional when GPU type is set. Specifies the number of GPUs to allocate for the container.

vnet_name Optional in case the network type is "private". Name of the virtual network to use for the container. If not specified, the Azure ACI resource plugin will allocate a virtual network.

subnet_name Optional in case the network type is "private". The name of the subnet to use for the container. If not specified, the Azure ACI resource plugin will allocate a subnet.

ports The list of ports to be exposed from the container. This is required to have at least one element defined (e.g. 8080).

CloudBroker

type: cloudbroker Selects the cloudbroker resource handler.

endpoint The endpoint (url) of the cloudbroker REST API interface.

name Optional. A user-defined name for this resource. Used in logging and can be referred to in the *authentication file* to distinguish authentication to be applied among resources having the same type.

description Description of the virtual machine to be started by CloudBroker. This is a subsection containing further keywords. The available keywords in this section is documented in the REST Web Service API documentation of CloudBroker on page 49. However, the most important ones are detailed below.

Obligatory keywords to be defined under *description* are as follows:

deployment_id Id of the deployment registered in CloudBroker. A deployment defines the cloud, the image, etc. to be instantiated.

instance_type_id Id of an instance type registered in CloudBroker and valid for the selected deployment. Instance type specifies the capabilities of the virtual machine to be instantiated.

Important/suggested keywords to be defined under *description* are as follows:

key_pair_id The ID of the (ssh) key pair to be deployed on the virtual machine. Key pairs can be registered in the CloudBroker platform behind the 'Users'/'Key Pairs' menu after login.

opened_port Determines if a port to be opened to the world. This is a list of numbers separated by comma.

Example for a resource section including the description subsection:

```
resource:
  type: cloudbroker
  endpoint: https://cola-prototype.cloudbroker.com/
  description:
    deployment_id: bcbdca8e-2841-45ae-884e-d3707829f548
    instance_type_id: c556cb53-7e79-48fd-ae71-3248133503ba
    key_pair_id: d865f75f-d32b-4444-9fbb-3332bcedeb75
    opened_port: 22,80
```

Docker

type: docker Selects the docker resource handler.

endpoint The endpoint (url) of the docker/swarm interface.

origin The URL of an image or leave it empty and default will be set to dockerhub.

image The name of the image, e.g ubuntu, debian, mysql ..

tag Docker tag. (default = latest)

name Optional. A user-defined name for this resource. Used in logging and can be referred to in the *authentication file* to distinguish authentication to be applied among resources having the same type.

CloudSigma

type: cloudsigma Selects the cloudsigma resource handler.

endpoint The endpoint (URL) of the CloudSigma interface, e.g. https://zrh.cloudsigma.com/api/2.0

libdrive_id The UUID of the library drive image to use. After login to CloudSigma UI at https://zrh.cloudsigma.com/ui, select the menu `Storage/Library`, select a library on page at https://zrh.cloudsigma.com/ui/#/library and use the uuid from the url of the selected item e.g. 40aa6ce2-5198-4e6b-b569-1e5e9fbaf488 for `Ubuntu 15.10 (Wily)` found at page https://zrh.cloudsigma.com/ui/#/library/40aa6ce2-5198-4e6b-b569-1e5e9fbaf488 .

name Optional. A user-defined name for this resource. Used in logging and can be referred to in the *authentication file* to distinguish authentication to be applied among resources having the same type.

description Description of the virtual machine to be started in CloudSigma (e.g. CPU, memory, network, public key). This is a section containing further keywords. The available keywords in this section is defined in the schema definition of CloudSigma VMs under the top-level keyword `fields`.

Obligatory keywords to be defined under *description* are as follows:

cpu Server's CPU Clock speed measured in MHz, e.g.: 2000

mem Server's Random Access Memory measured in bytes, e.g.: 1073741824 (for 1 GByte)

vnc_password VNC Password to connect to server, e.g. "secret"

Example for a typical description section, using 2GHz CPU, 1GB RAM with public ip address.

```yaml
description:
  cpu: 2000
  mem: 1073741824
  vnc_password: the_password
  name: the_hostname
  pubkeys:
    -
      the_uuid_of_an_uploaded_keypair
  nics:
    -
      firewall_policy: the_uuid_of_a_predefined_firewall_policy
      ip_v4_conf:
        conf: dhcp
        ip: null
```

(continues on next page)

(continued from previous page)

```
    runtime:
      interface_type: public
```

2.5.3.2 Collecting Resource Attributes

The following subsections detail how the string values (identifiers, settings, etc.) for the different attributes/keywords under the resource section of the node definition can be collected using the user interface of a particular resource.

Amazon (EC2)

This tutorial helps users how the attributes for the resource section in the node definition can be collected from the web interface of the Amazon cloud.

First of all, you need of course an Amazon AWS account. Using Amazon AWS is implemented using the EC2 interface, thus the *EC2-Helloworld* tutorial is a good starting point.

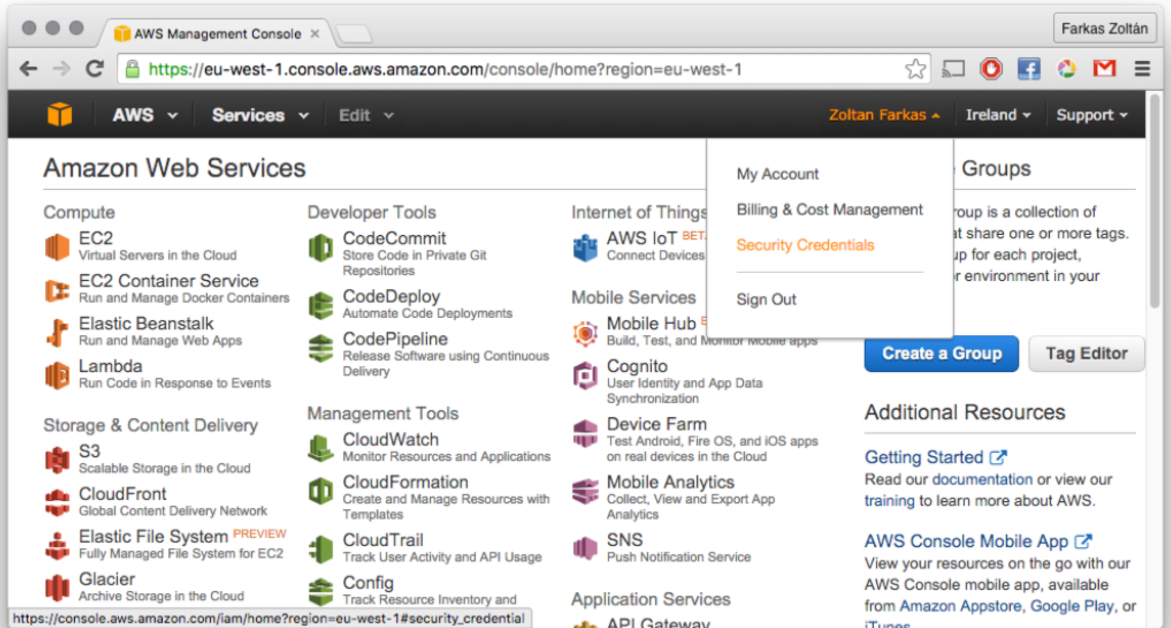
In case of Amazon EC2, the following information is necessary to start up a node in an Occopus infrastructure:

- security credentials (access key and secret key)
- Amazon region name and its EC2 endpoint
- an image ID (AMI)
- an instance type
- at least one security group ID
- a key pair name
- a subnet identifier.

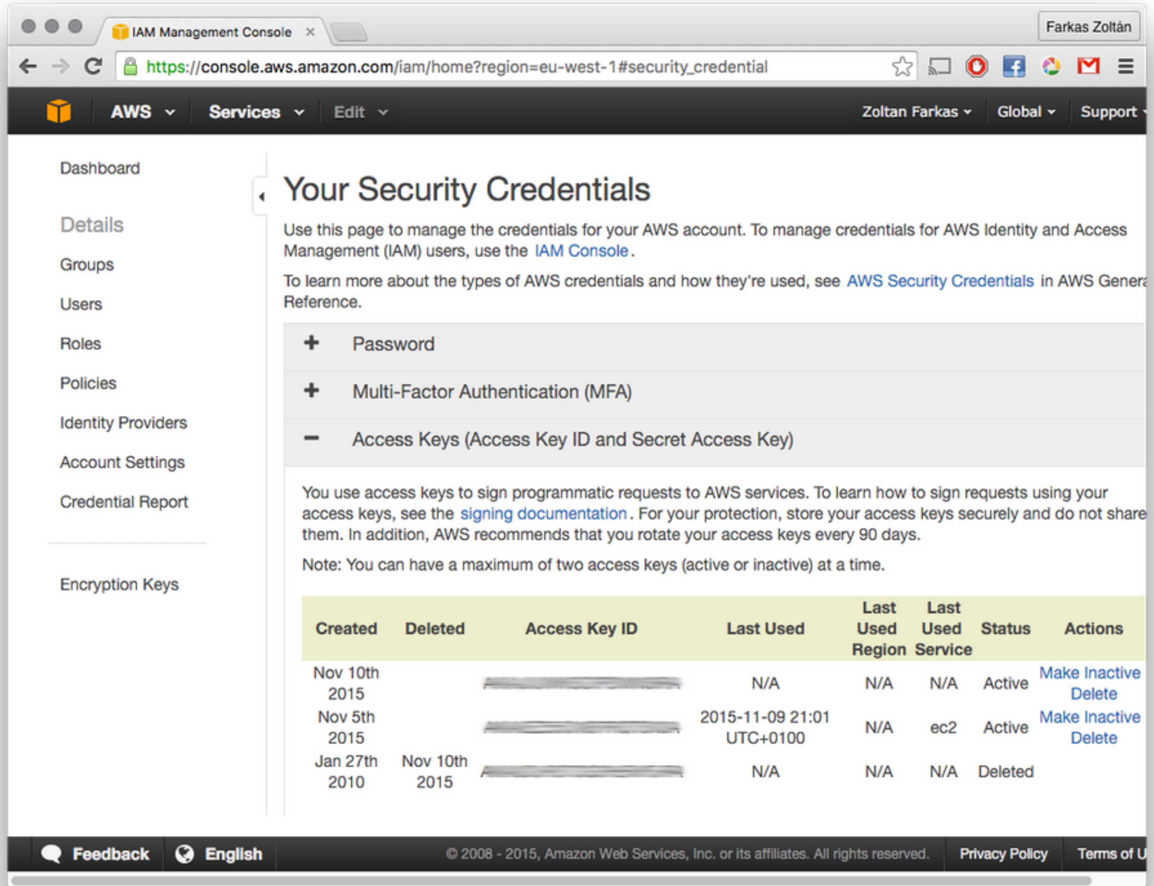
Security credentials (access key and secret key)

You can get your access key and secret key through the web interface of Amazon AWS:

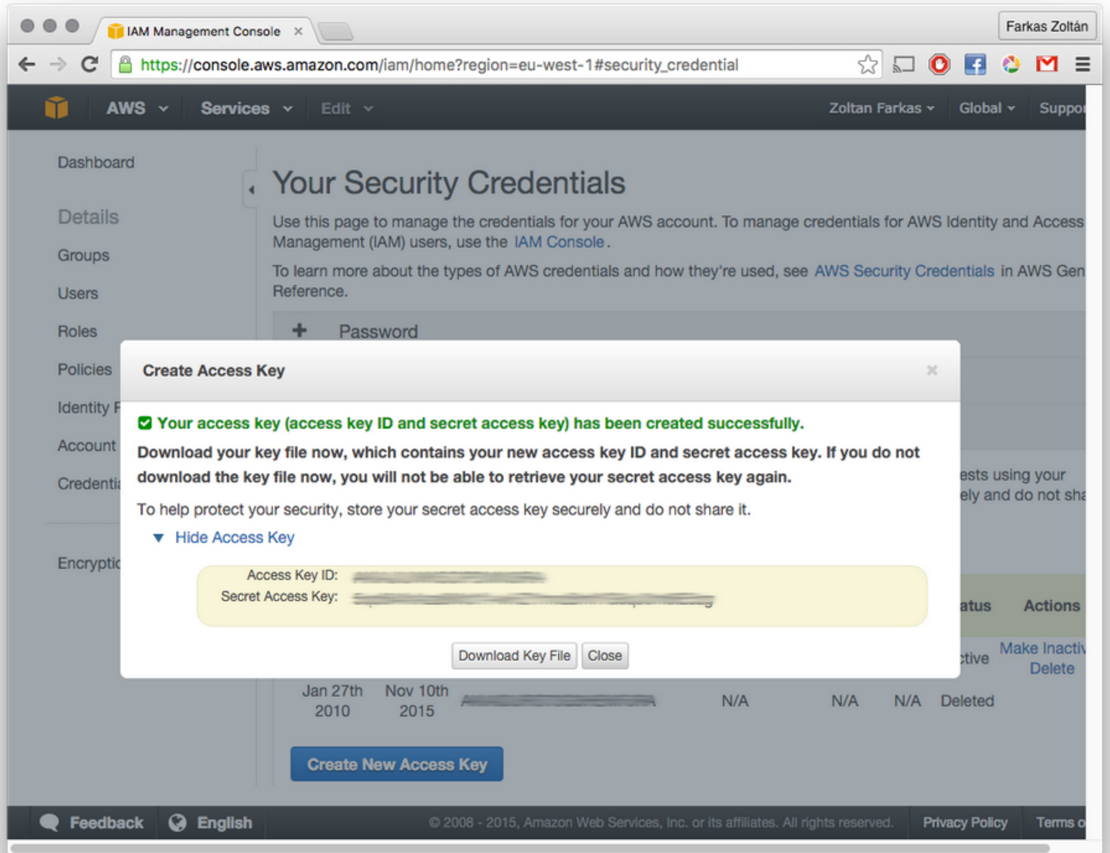
1. Visit the AWS console.
2. In the top right corner, select "Security credentials" under your profile as shown in the following screenshot:



3. Expand the Access keys menu, as shown in the following screenshot:



4. Click on the "Create New Access Key" button to create new credentials if you don't know the Secret Access Key of your already existing key(s). A window similar to the following screenshot will appear. Here you can make your **Access Key ID** and **Secret Access Key** appear, but you can also download your credentials for later use.



Amazon region name and its EC2 endpoint

Amazon hosts its services in multiple regions. There are two possible ways to get region names and their relevant EC2 endpoints: using the EC2 command line tools or the web interface.

Use the web interface to get region names and EC2 endpoints

The Amazon Documentation Amazon Documentation has a list of available regions and their EC2 endpoints. In order to get the complete EC2 endpoint URL for Occopus, simply add `https://` before the Endpoint specified by the table shown in the Amazon Documentation's table. For example, the EC2 endpoint URL of the `eu-west-1` region is `https://ec2.eu-west-1.amazonaws.com`. Simple as that.

Use the EC2 command line tools to get region names and EC2 endpoints

Follow the EC2 command line tool setup guide to set up and configure EC2 command line tools onto your machine. Once done, you can use the `ec2-describe-regions` command to list available regions and EC2 endpoints:

```
$ ec2-describe-regions  -H
REGION      Name            Endpoint
REGION      eu-west-1       ec2.eu-west-1.amazonaws.com
REGION      ap-southeast-1  ec2.ap-southeast-1.amazonaws.com
REGION      ap-southeast-2  ec2.ap-southeast-2.amazonaws.com
REGION      eu-central-1    ec2.eu-central-1.amazonaws.com
REGION      ap-northeast-1  ec2.ap-northeast-1.amazonaws.com
REGION      us-east-1       ec2.us-east-1.amazonaws.com
```

(continues on next page)

(continued from previous page)

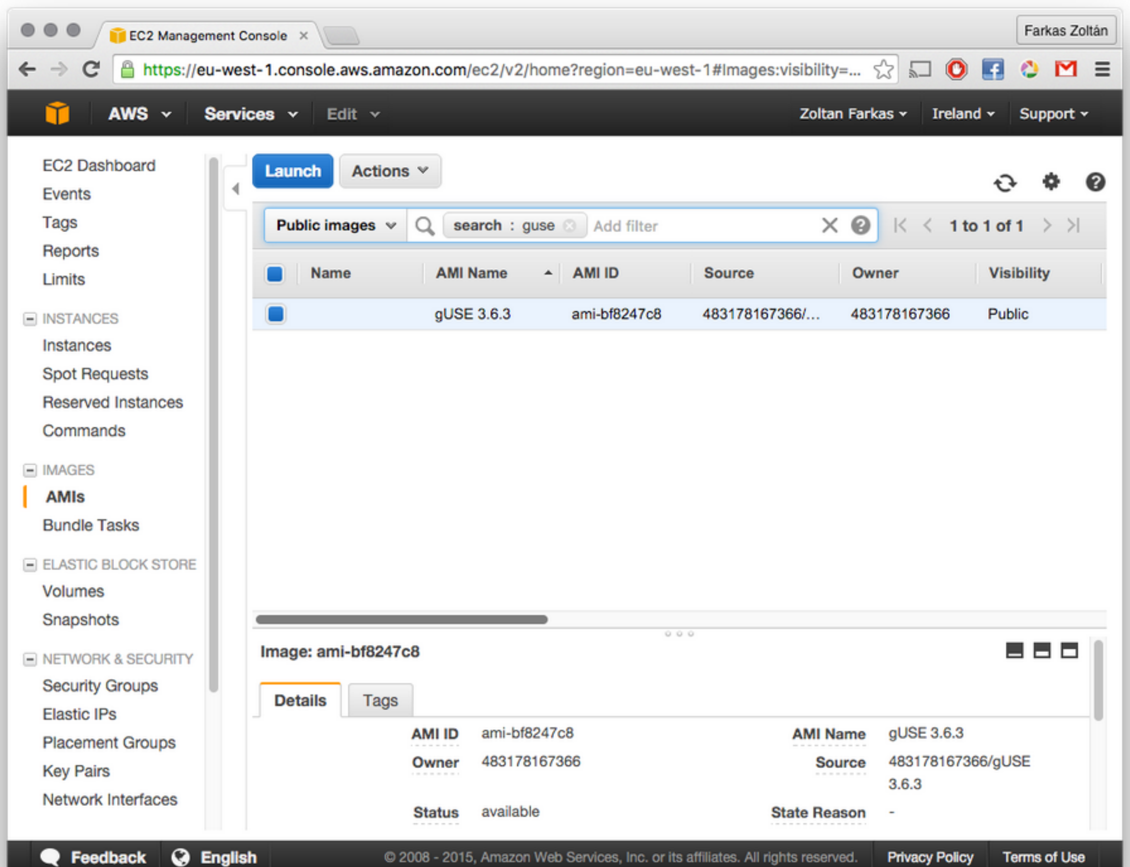
```
REGION      sa-east-1       ec2.sa-east-1.amazonaws.com
REGION      us-west-1       ec2.us-west-1.amazonaws.com
REGION      us-west-2       ec2.us-west-2.amazonaws.com
```

Here, the second column shows the region name, the third column shows the EC2 endpoint for the given region. You should prefix the endpoint name with `https://` for receiving the endpoint URL for Occopus.

Get image ID

Two possible methods are available to get a proper image ID: using the EC2 CLI tools' `ec2-describe-images -a` command and the web interface. The second one is preferred, as one can get a more user-friendly description of the picked on image.

In the AWS EC2 management console, select **AMIs** from the **IMAGES** menu. Search for an AMI, as shown in the screenshot below:



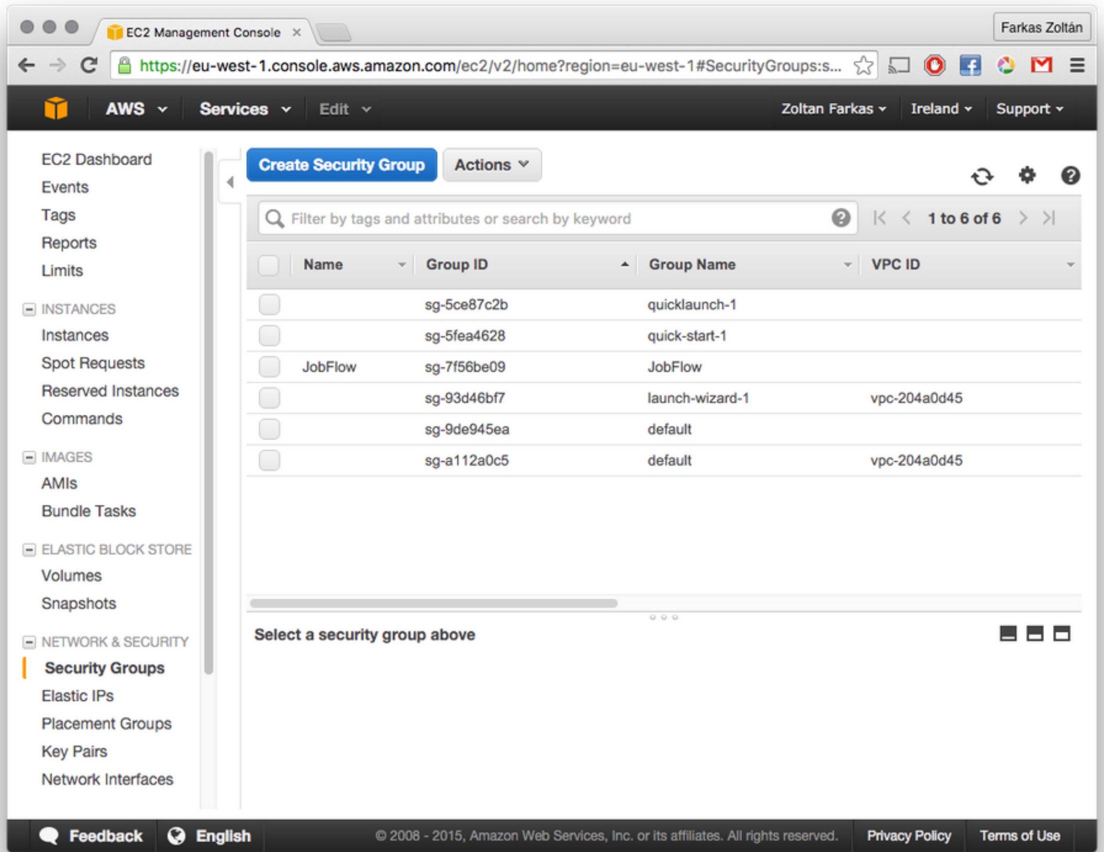
Here, the value of the **AMI ID** column contains the image identifier.

Get instance type

The instance type determines the characteristics (CPU, memory) of the VM created. You can get the names and properties of the instance types supported by Amazon through the Instance types documentation.

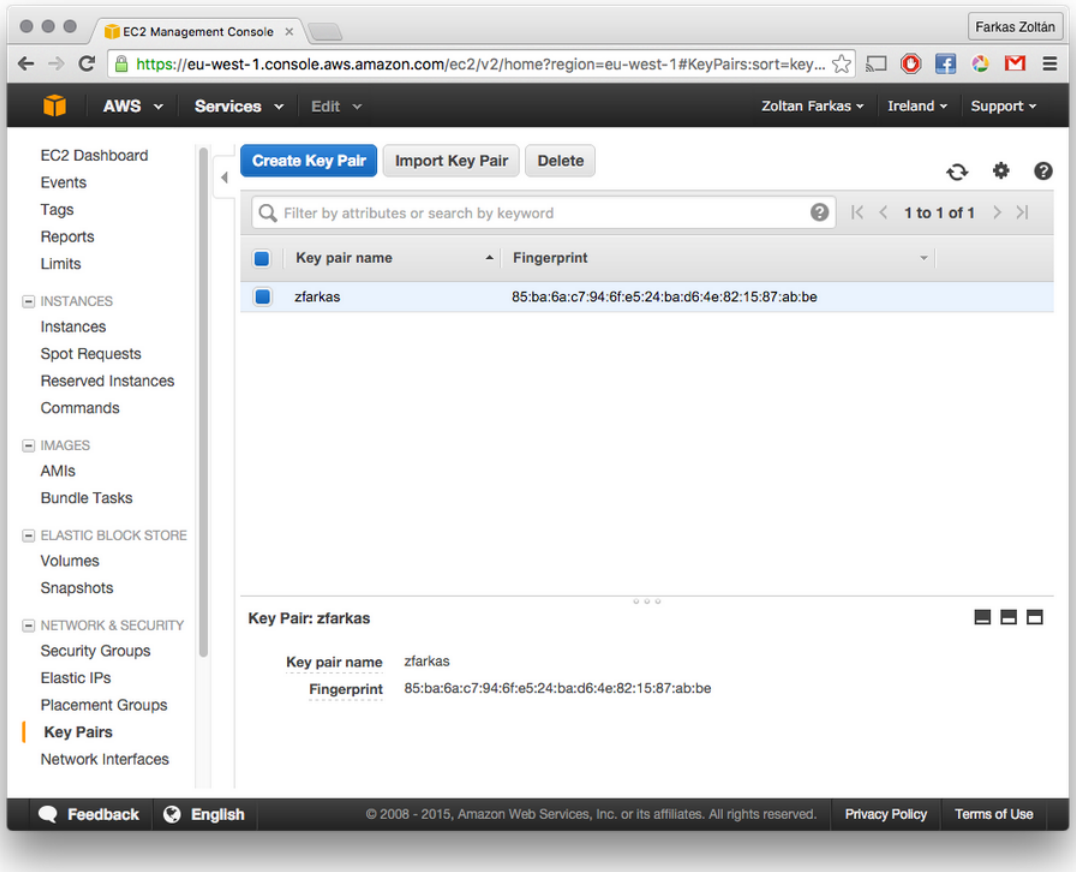
Get security group IDs

Security groups define the network traffic allowed for the instances to be started. Thus, you should create security groups in order to enable SSH or HTTP traffic into your VM. The following screenshot shows a number of security groups already defined. Select those you'd like to attach to the VM started by Occous. The value of the **Group ID** column contains the values which are needed by Occopus.



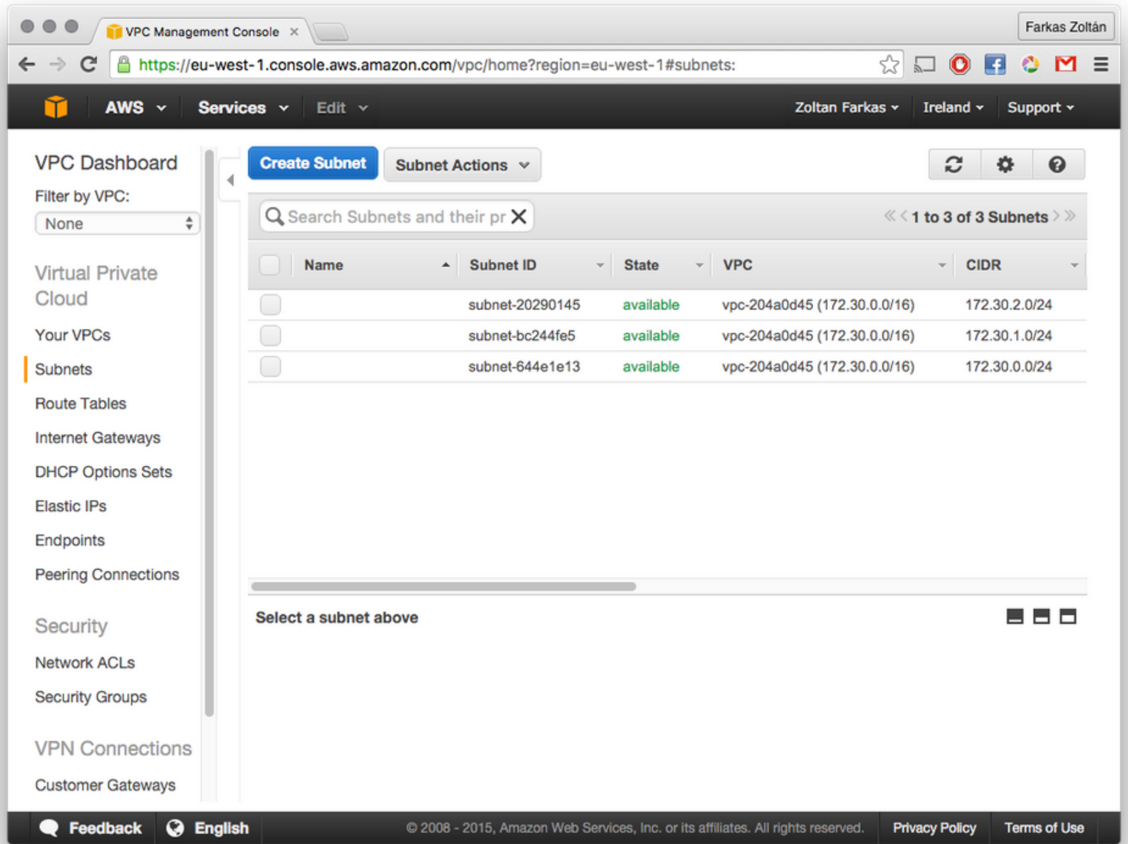
Get keypair name

Key pairs are importd into your running VM so SSH access is possible. You can check the name of available keypairs in the AWS EC2 management console, under the **Key Pairs** menu as shown in the following screenshot. The value of the **Key pair name** is the one Occopus needs.



Get Subnet identifier

You can get the list of available subnets through the AWS VPC dashboard, by selecting **Subnets** from the menu as shown in the following screenshot. You should use the value of the **Subnet ID** column for Occopus.



Closing

With all the above values, now you can modify the *EC2-Helloworld* tutorial to run on Amazon.

Cloudbroker

This tutorial helps users how the attribute values under the resource section in node definition for the cloudbroker plugin can be collected from the web interface of CloudBroker.

A minimal version of the resource section for CloudBroker may look like as follows:

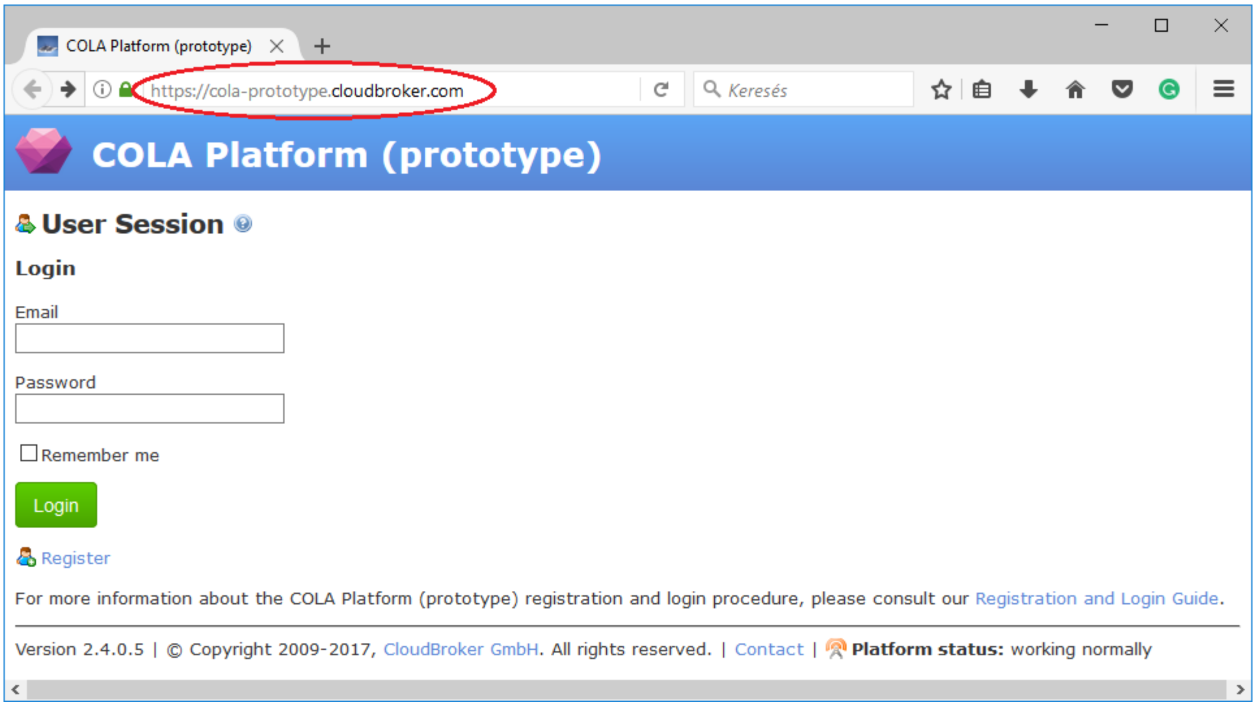
```
resource:
  type: cloudbroker
  endpoint: replace_with_endpoint_of_cloudbroker_interface
  description:
     deployment_id: replace_with_deployment_id
     instance_type_id: replace_with_instance_type_id
     key_pair_id: replace_with_keypair_id
     opened_port: replace_with_list_of_ports_separated_with_comma
contextualisation:
  ...
```

You need to collect the following attributes to complete the resource section:

1. `endpoint`
2. `deployment_id`
3. `instance_type_id`
4. `key_pair_id`
5. `opened_port`

endpoint

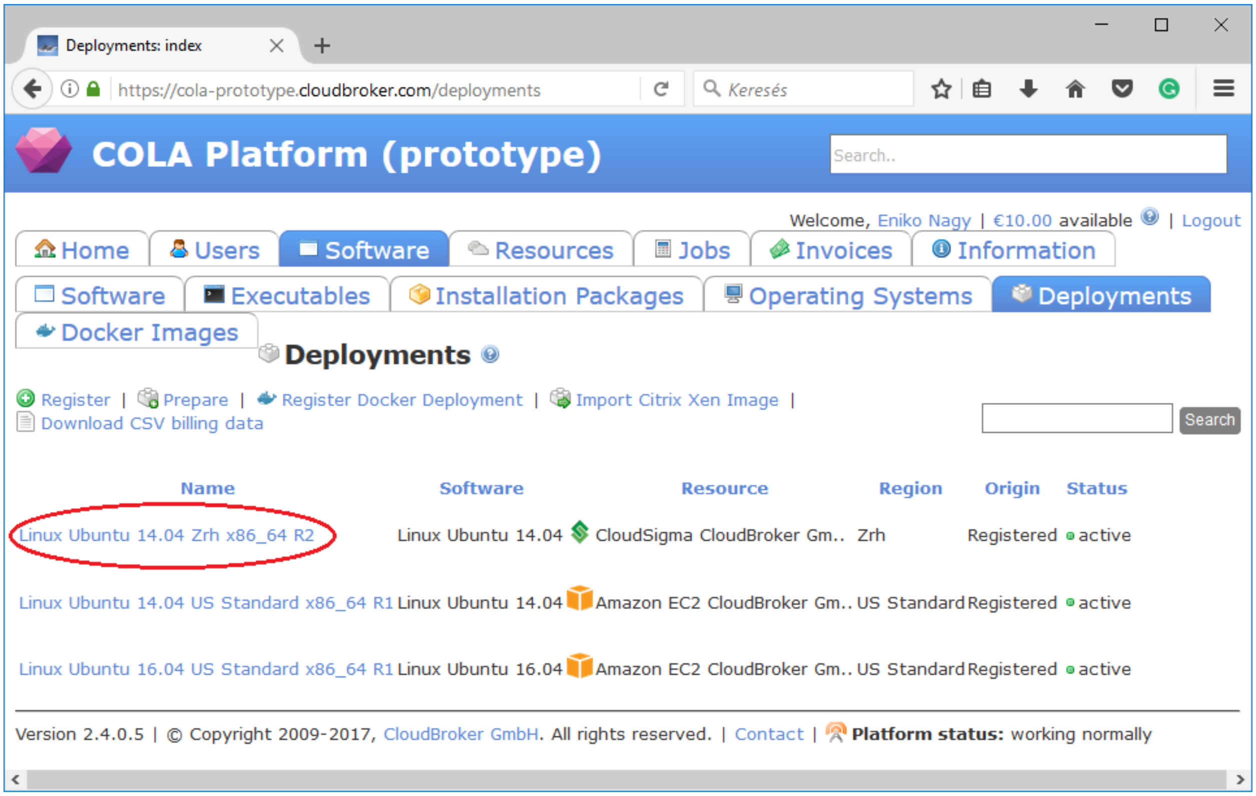
The value of this attribute is the url of the CloudBroker REST API interface, which is usually the same as the login url.



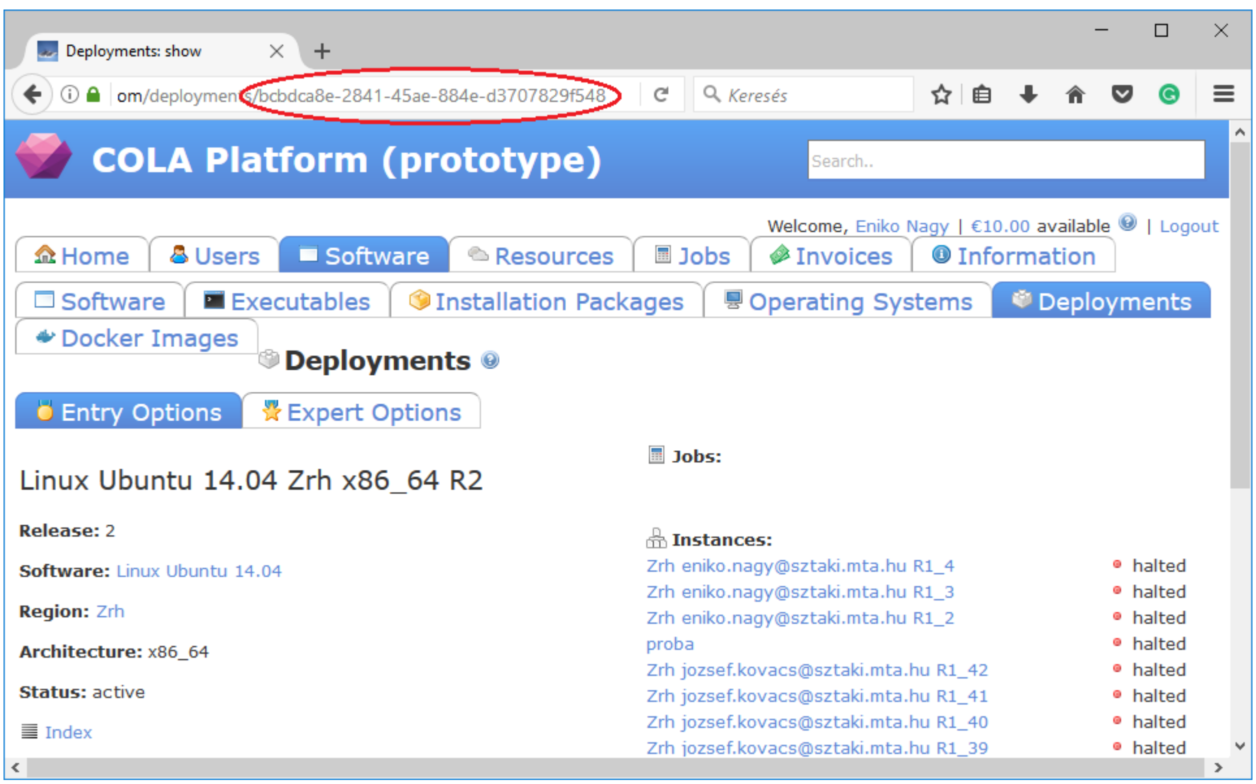
As a result, in our case the `endpoint` attribute in the resource section will be `https://cola-prototype.cloudbroker.com`.

deployment_id

The value of this attribute is the id of a preregistered deployment in CloudBroker referring to a cloud, image, region, etc. After login to the CloudBroker Web UI, select `Software/Deployments` menu.



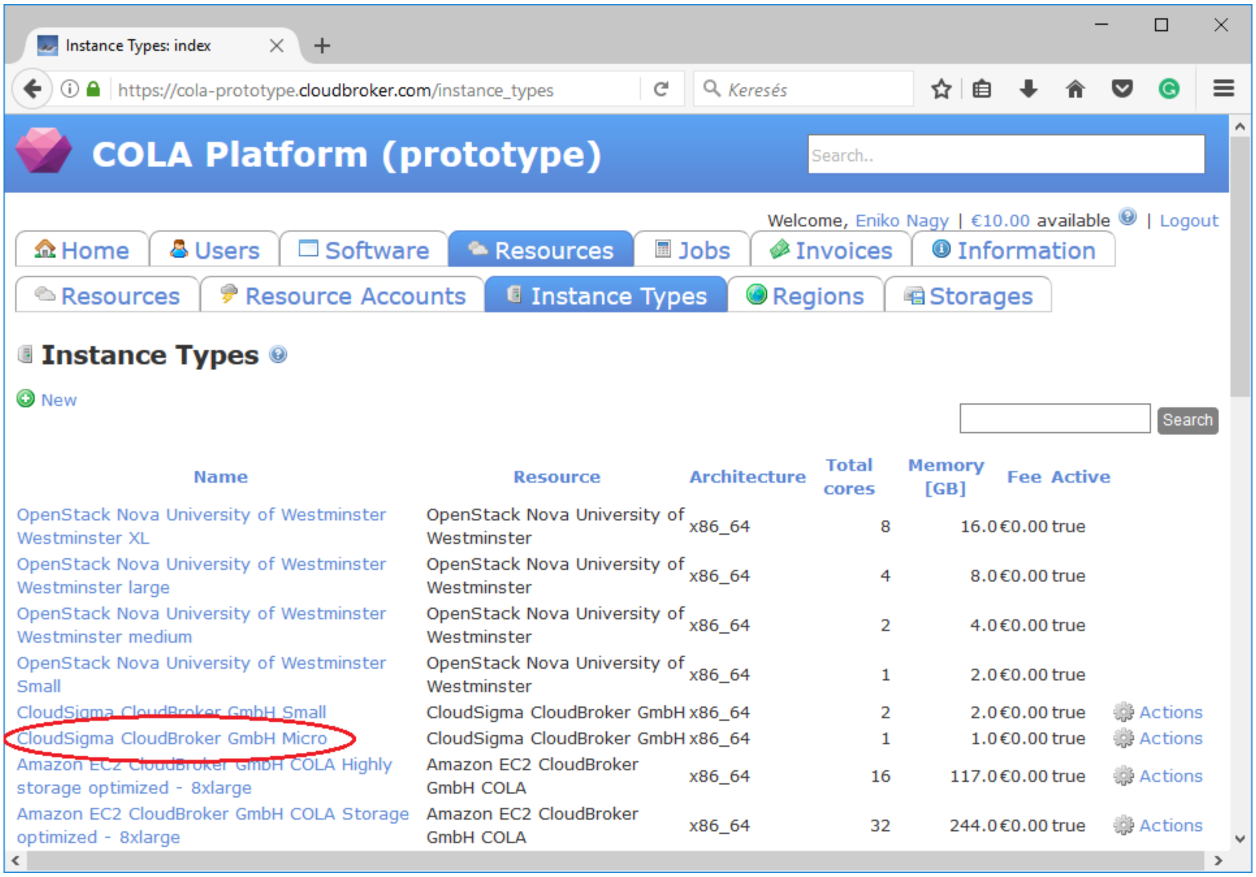
On this page you can see the list of the preregistered deployments. Make sure the image contains a base os (preferably Ubuntu) installation with cloud-init support! Assuming we need a `Linux Ubuntu 14.04 on CloudSigma`, click on the name of the deployment. The id is the UUID of the deployment which can be seen in the address bar of your browser.



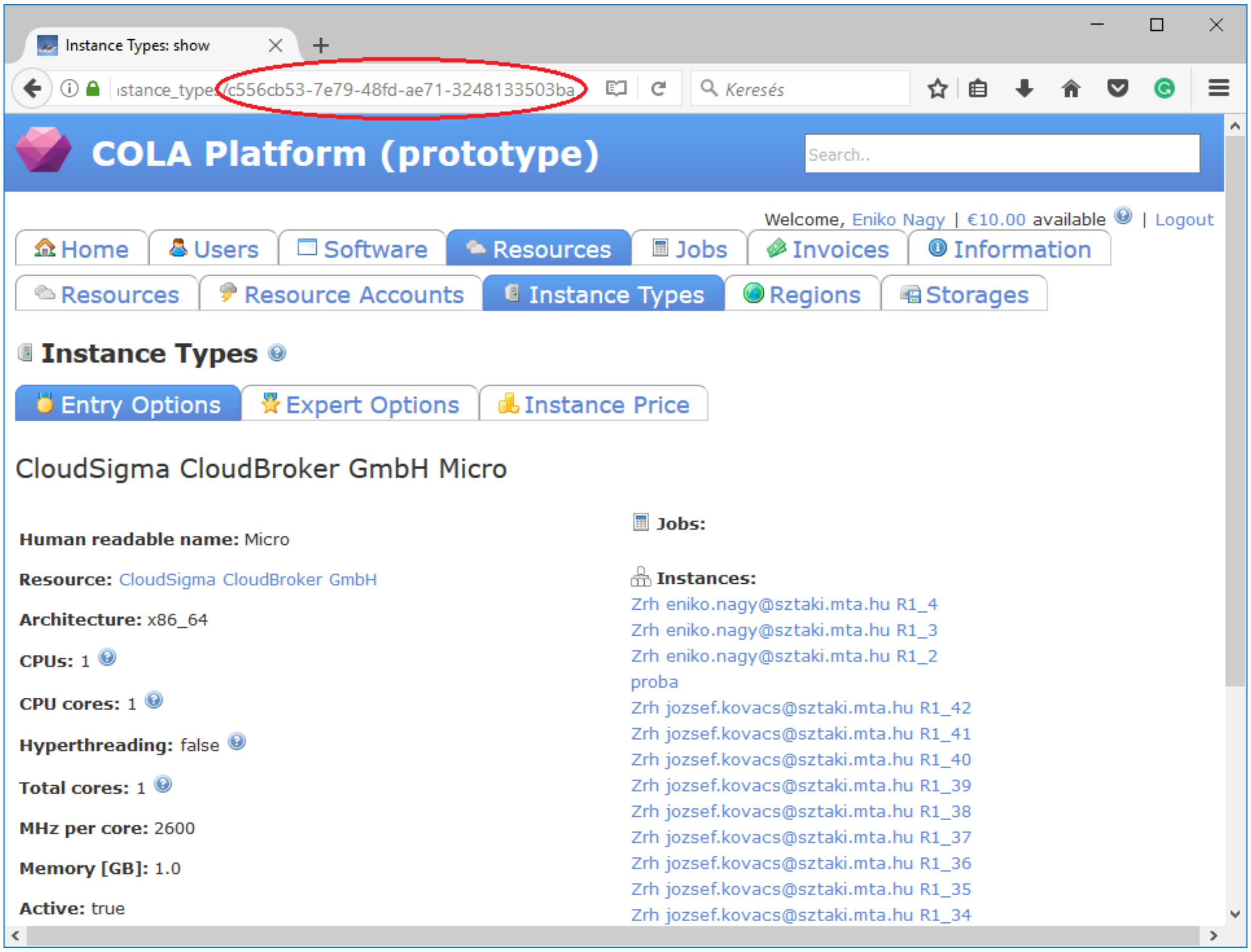
As a result, the `deployment_id` attribute in the resource section will be `bcbdca8e-2841-45ae-884e-d3707829f548`.

instance_type_id

The value of this attribute is the id of a preregistered instance type in CloudBroker referring to the capacity of the virtual machine to be deployed. Select `Resources/Instance Types` menu. On this page you can see the list of available instance types.



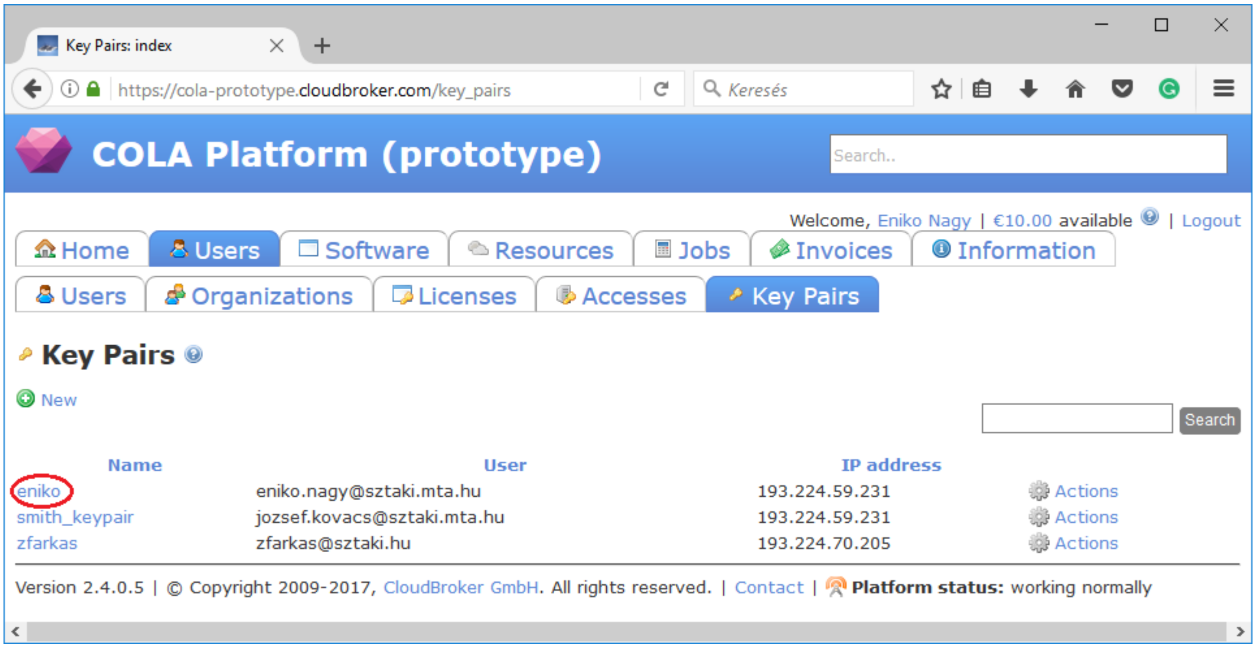
Assuming we need a `Micro instance type for CloudSigma`, select and click on the instance type. The id is the UUID of the instance type which can be seen in the address bar of your browser when inspecting the details of the instance type.



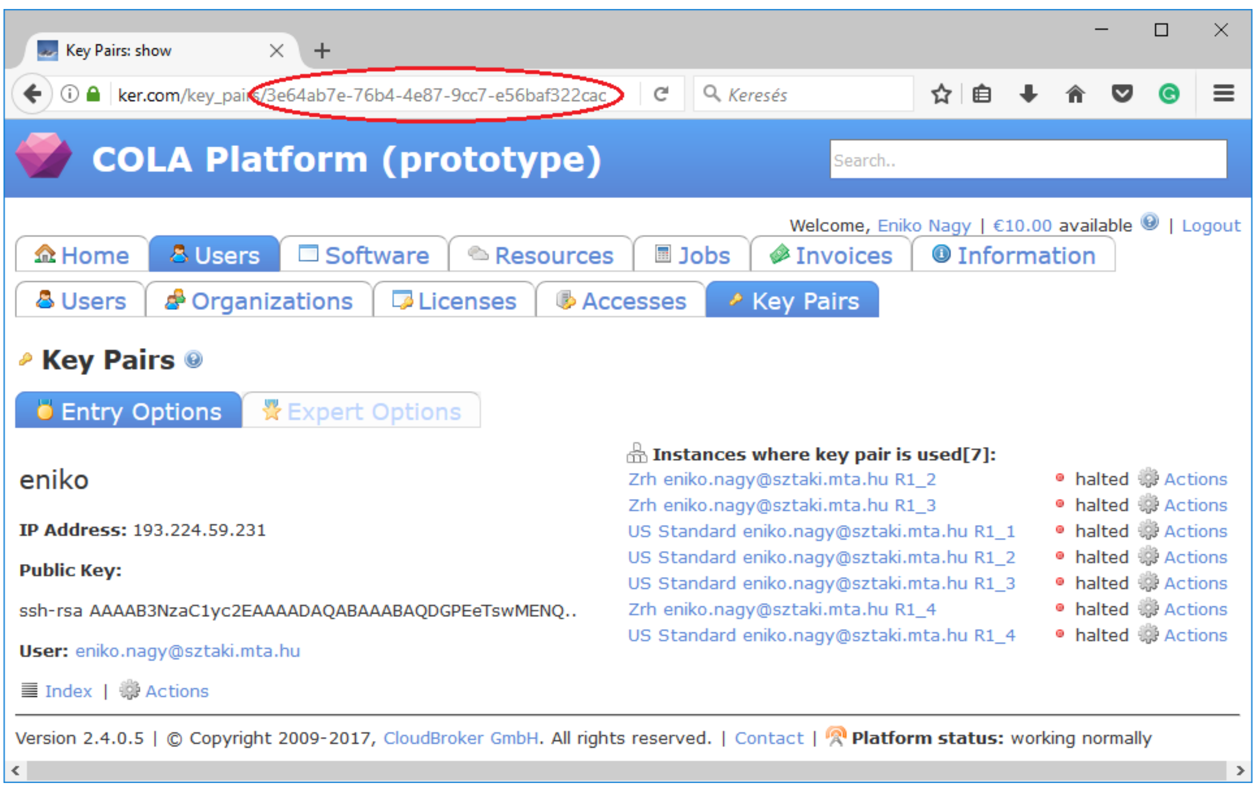
As a result, the `instance_type_id` attribute in the resource section will be `c556cb53-7e79-48fd-ae71-3248133503ba`.

key_pair_id:

The value of this attribute is id of a preregistered ssh public key in CloudBroker which will be deployed on the virtual machine. To register a new ssh public key, upload one on page under the `Users/Key Pairs` menu.



On this page you can see the list of registered keys. Assuming we need the key with name "eniko", click on the name of the key. The id is the UUID of the key pair which can be seen in the address bar of your browser when inspecting the details of the key pair.



As a result, the `key_pair_id` attribute in the resource section will be `3e64ab7e-76b4-4e87-9cc7-e56baf322cac`.

opened_port:

The opened_port is one or more ports to be opened to the world. This is a string containing numbers separated by comma. Assuming we would like to open ports 80 and 443 for our web server, the `opened_port` attribute in the resource section will be '80, 443'.

The finalised resource section with the IDs collected in the example above will look like this:

```
resource:
  type: cloudbroker
  endpoint: https://cola-prototype.cloudbroker.com/
  description:
    deployment_id: bcbdca8e-2841-45ae-884e-d3707829f548
    instance_type_id: c556cb53-7e79-48fd-ae71-3248133503ba
    key_pair_id: 3e64ab7e-76b4-4e87-9cc7-e56baf322cac
    opened_port: '80, 443'
contextualisation:
  ...
```

CloudSigma

The following tutorial will help users how the attributes for the resource section in the node definition can be collected from the web interface of the CloudSigma cloud. In the following example we will use the Zurich site of CloudSigma.

A minimal version of the resource section for CloudSigma may look like as follows:

```
resource:
  type: cloudsigma
  endpoint: https://zrh.cloudsigma.com/api/2.0
  libdrive_id: <uuid_of_selected_drive_from_library>
  description:
    cpu: 2000
    mem: 2147483648
    pubkeys:
      -
        <uuid_of_your_registered_public_key>
    nics:
      -
        firewall_policy: <uuid_of_your_registered_firewall_policy>
        ip_v4_conf:
          conf: dhcp
contextualisation:
  ...
```

The example above assumes the followings:

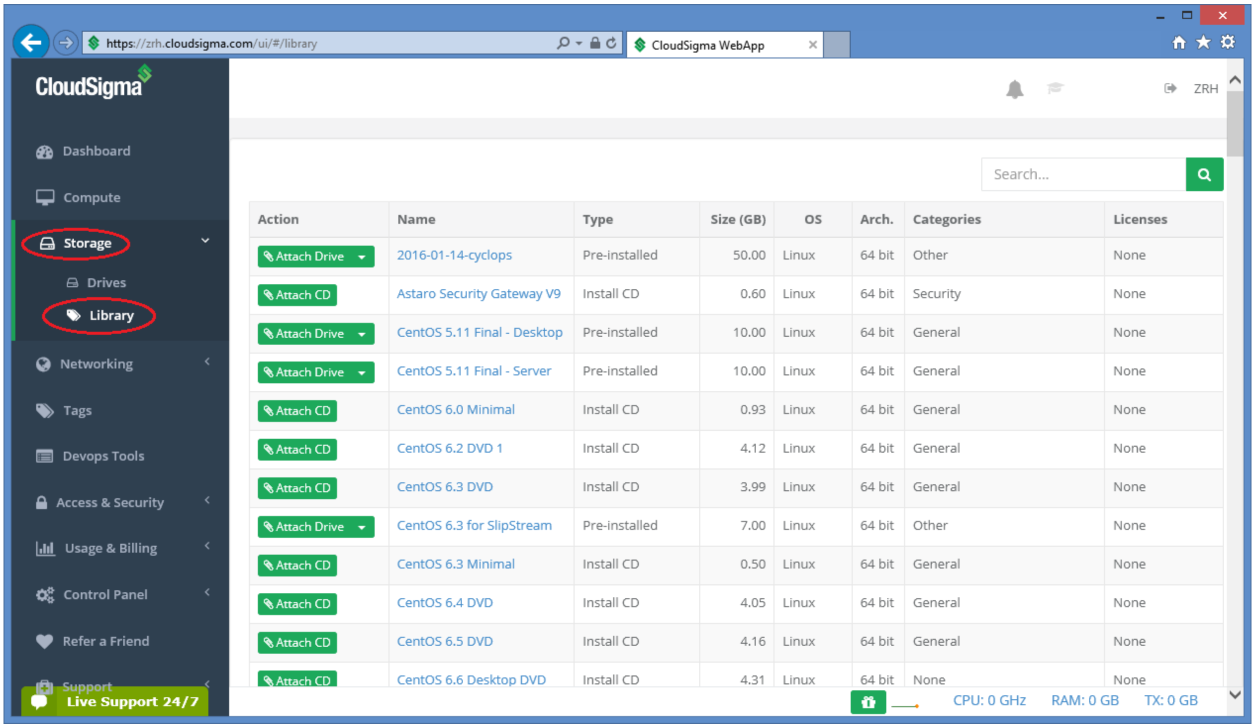
1. Virtual machine will be started at the Zurich site, see `endpoint` attribute. To use an alternative location, select one from the cloudsigma documentation on API endpoints.
2. CPU speed will be 2000Mhz. See `cpu` attribute.
3. Memory size will be 2GByte. See `mem` attribute.
4. VM will have a public ip address defined by dhcp. See `ip_v4_conf` attribute.

You need to collect the following 3 more attributes to complete the section:

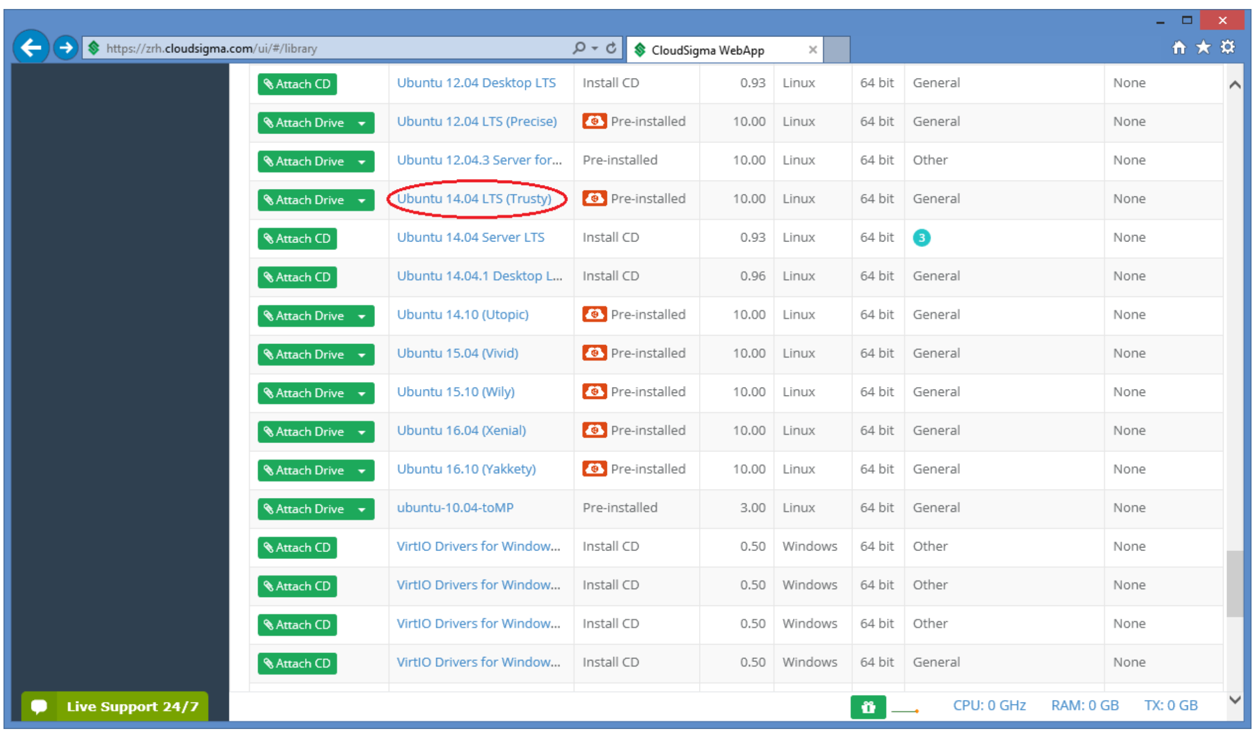
1. `libdrive_id`
2. `pubkeys`
3. `firewall_policy`

libdrive_id

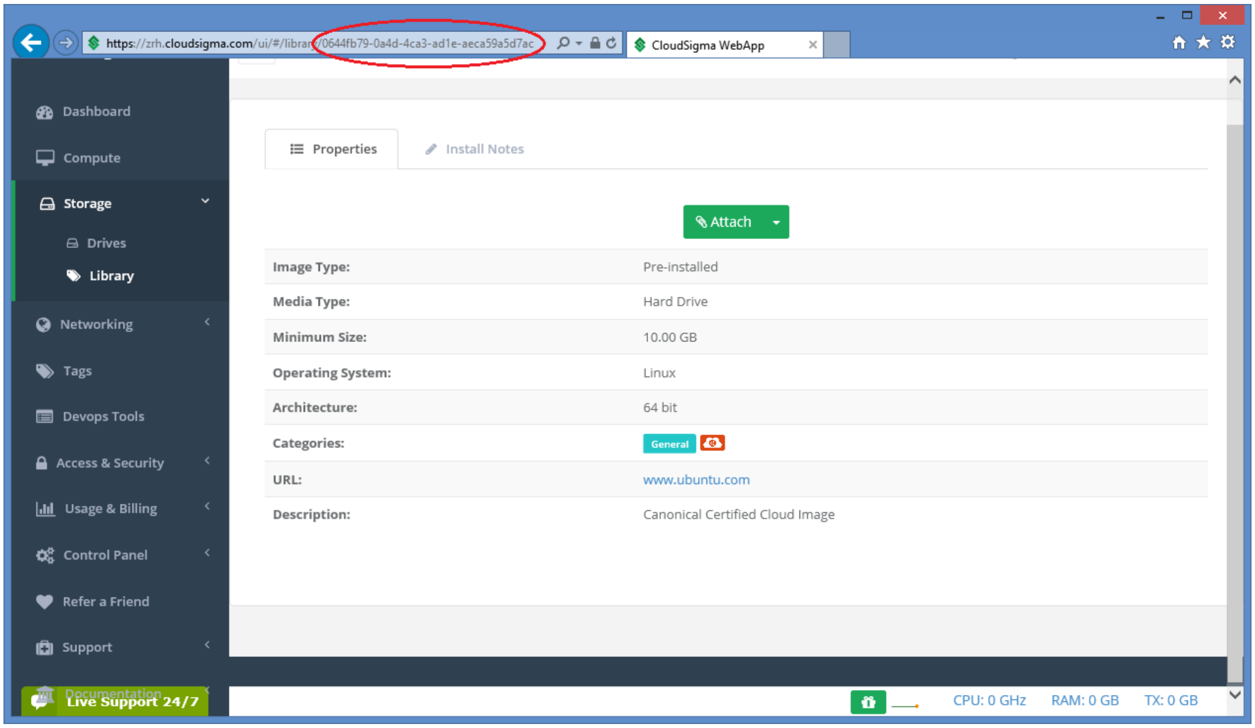
The value of this attribute is an uuid refering to a particular drive in the storage library on which an operating system is preinstalled usually. After login to the CloudSigma Web UI, select `Storage/Library` menu and a full list of available drives will be listed.



Assuming we need an `Ubuntu 14.04 LTS(Trusty)`, scroll down and search for that drive.



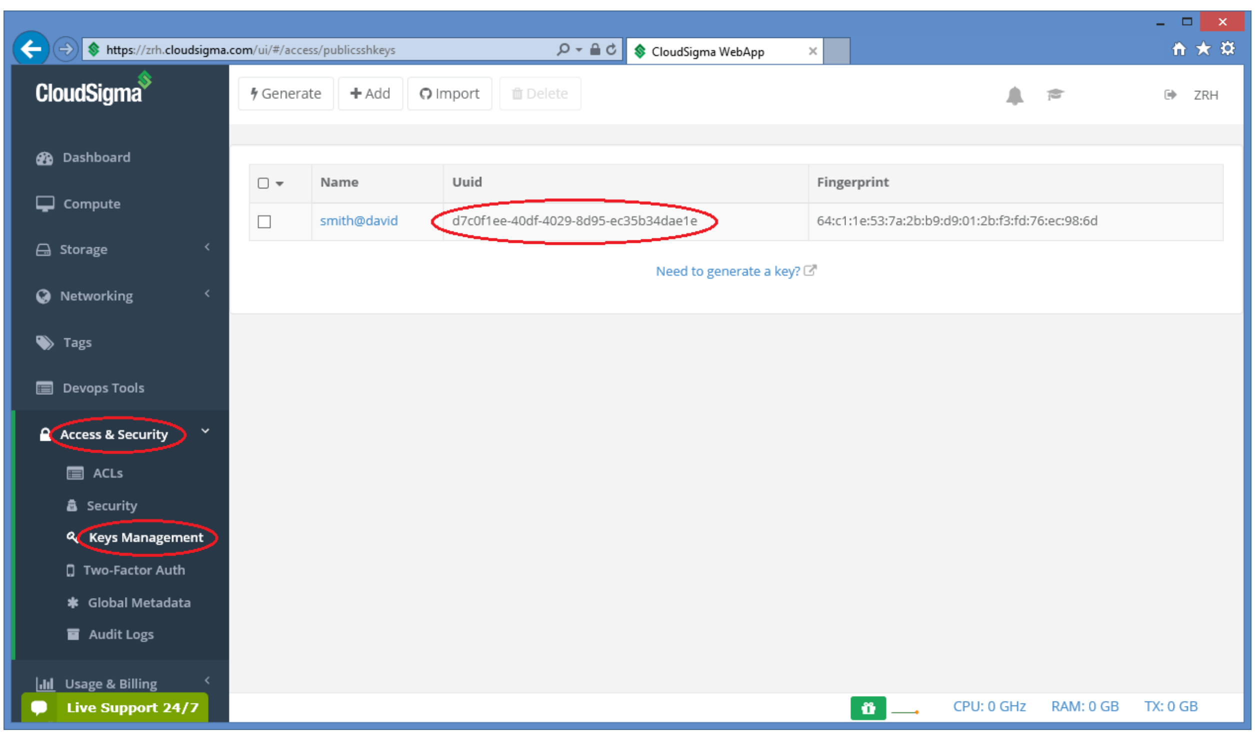
Then click on the item and copy its uuid from the address bar.



As a result, the `libdrive_id` attribute in the resource section will be `0644fb79-0a4d-4ca3-ad1e-aeca59a5d7ac` referring to the drive containing an `Ubuntu 14.04 LTS(Trusty)` operating system.

pubkeys

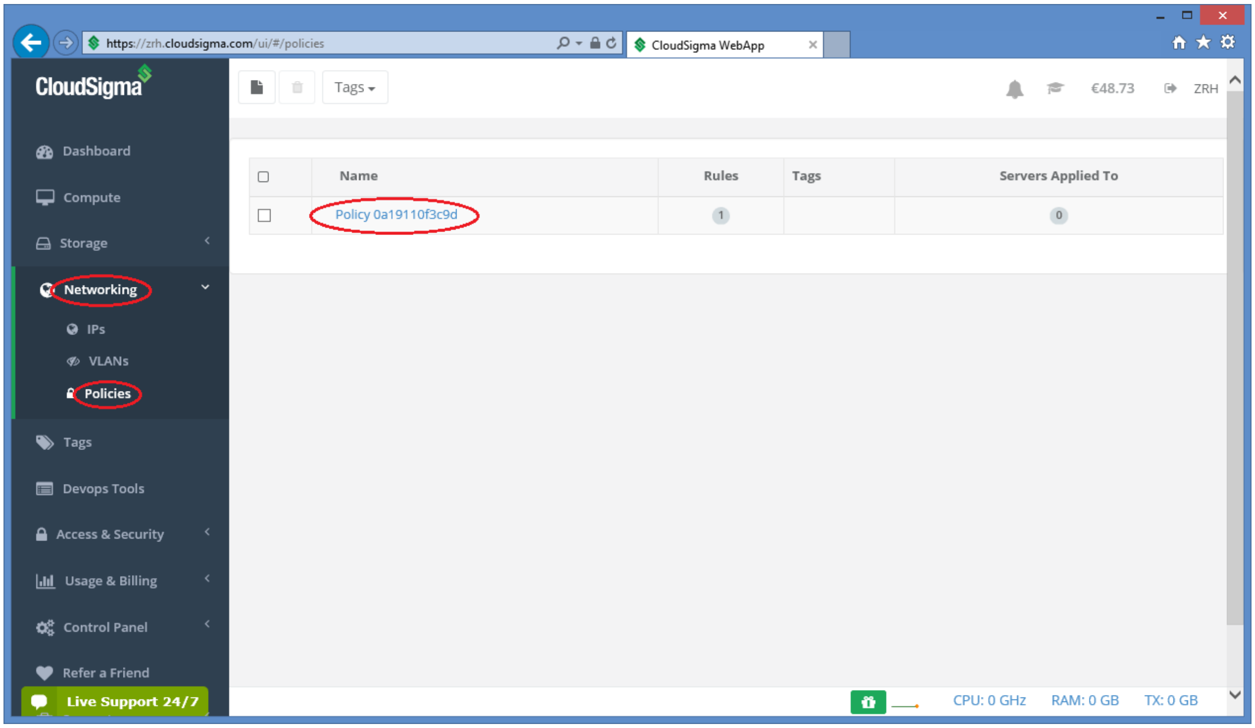
The value of this attribute is the uuid refering to a particular public key registered under your CloudSigma account. To register a new ssh keypair, generated or upload one at page under the `Access & Security/Keys Management` menu. On this page you can see the list of registered keys and their uuid.



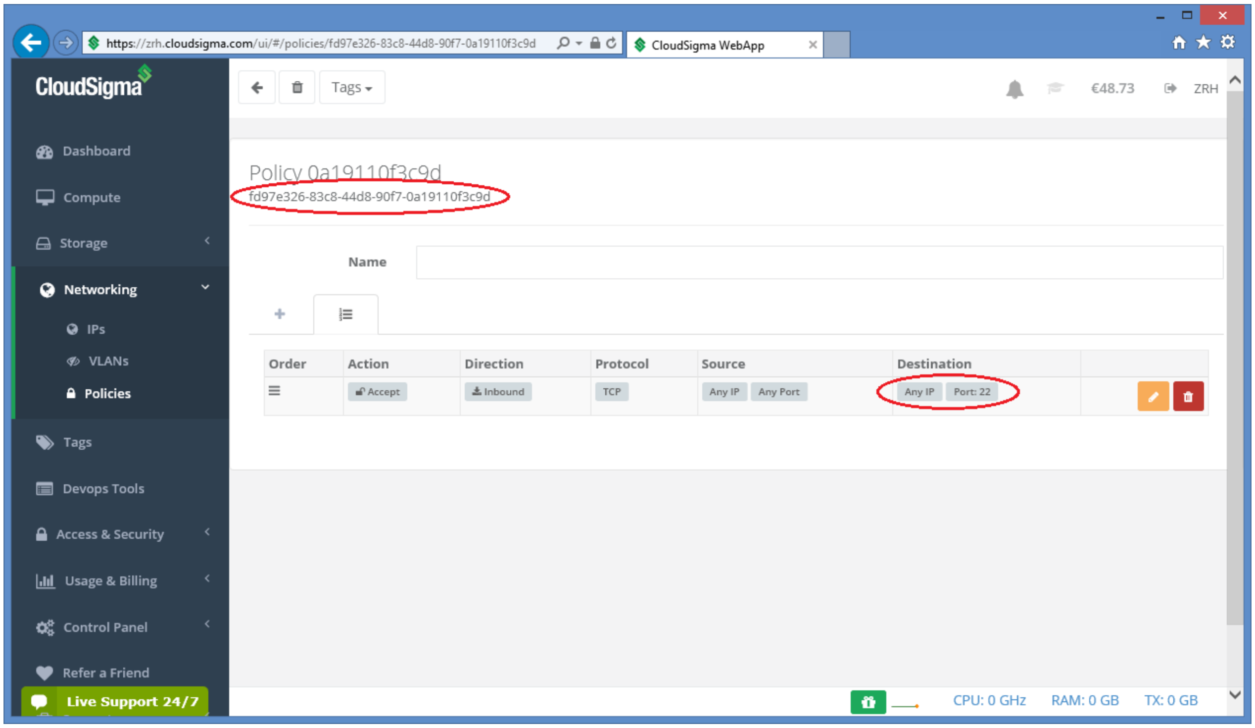
As a result, the `pubkeys` attribute in the resource section will be `d7c0f1ee-40df-4029-8d95-ec35b34dae1e` in this case refering to the selected key. Multiple keys can be specified, if necessary.

firewall_policy

The value of this attribute is the uuid refering to a particular firewall policy registered under your CloudSigma account. To register a new firewall policy, use the page under the `Networking/Policies` menu. On this page you can see the list of registered firewall policies.



Click on the firewall policy to be applied on the VM, the new page will show the uuid of the policy.



As a result, the `firewall_policy` attribute in the resource section will be `fd97e326-83c8-44d8-90f7-0a19110f3c9d` in this case refering to the selected policy. In this policy, port 22 is open for ssh. Multiple policies can be specified, if necessary.

The finalised resource section with the uuids collected in the example above will look like this:

```
resource:
  type: cloudsigma
  endpoint: https://zrh.cloudsigma.com/api/2.0
  libdrive_id: 0644fb79-0a4d-4ca3-ad1e-aeca59a5d7ac
  description:
    cpu: 2000
    mem: 2147483648
    pubkeys:
      -
        d7c0f1ee-40df-4029-8d95-ec35b34dae1e
    nics:
      -
        firewall_policy: fd97e326-83c8-44d8-90f7-0a19110f3c9d
        ip_v4_conf:
          conf: dhcp
contextualisation:
  ...
```

Important: Collect the uuids under your account instead of using the ones in this example!

Important: The resource section must follow YAML syntax! Make sure indentation is proper, avoid using <tab>, use spaces!

OpenStack Horizon (Nova)

This tutorial helps users how the attribute values under the resource section in node definition for the nova plugin can be collected from the Horizon web interface of OpenStack. In this help the hungarian MTA Cloud will be taken as an example to show the procedure.

A minimal version of the resource section for MTA Cloud may look like as follows:

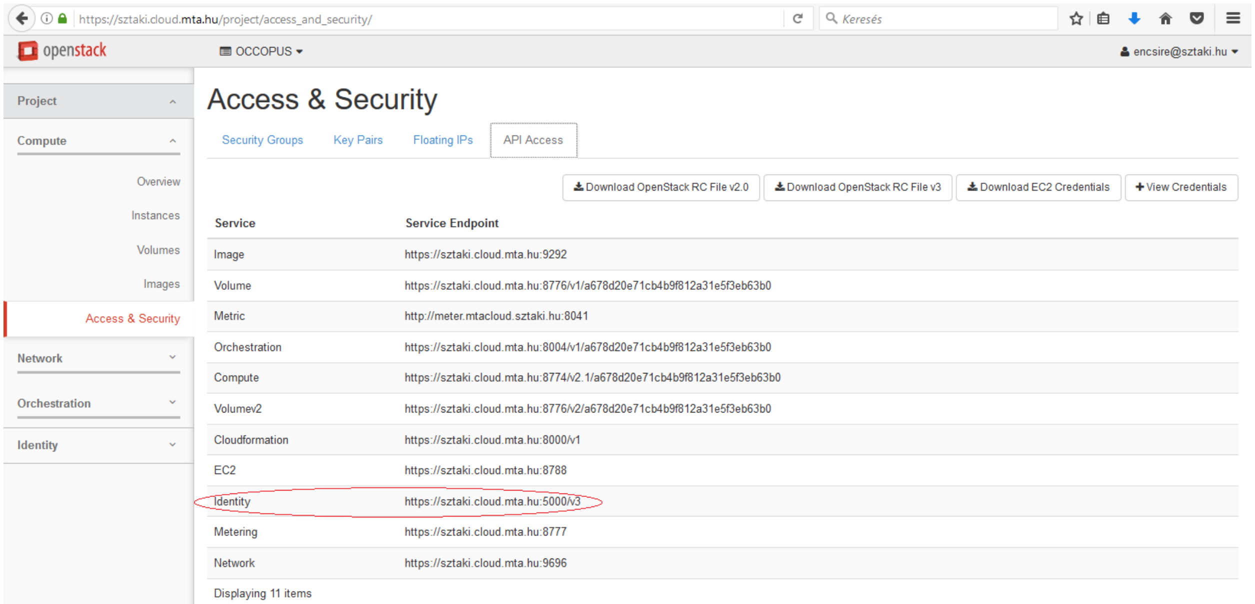
```
resource:
  type: nova
  endpoint: replace_with_endpoint_of_nova_interface_of_your_cloud
  project_id: replace_with_projectid_to_use
  user_domain_name: Default
  image_id: replace_with_id_of_your_image_on_your_target_cloud
  network_id: replace_with_id_of_network_on_your_target_cloud
  flavor_name: replace_with_id_of_the_flavor_on_your_target_cloud
  key_name: replace_with_name_of_keypair_or_remove
  security_groups:
      -
          replace_with_security_group_to_add_or_remove_section
  floating_ip: add_yes_if_you_need_floating_ip_or_remove
  floating_ip_pool: replace_with_name_of_floating_ip_pool_or_remove
contextualisation:
  ...
```

You need to collect the following attributes to complete the resource section:

1. endpoint
2. project_id
3. image_id
4. network_id
5. flavor_name
6. key_name
7. security_groups

endpoint

The endpoint is an url of the nova interface of your OpenStack cloud. After login to the Horizon Web UI, select `Project/Compute/Access & Security/API Access` menu. The value of the endpoint is the service endpoint of the *Identity* service.

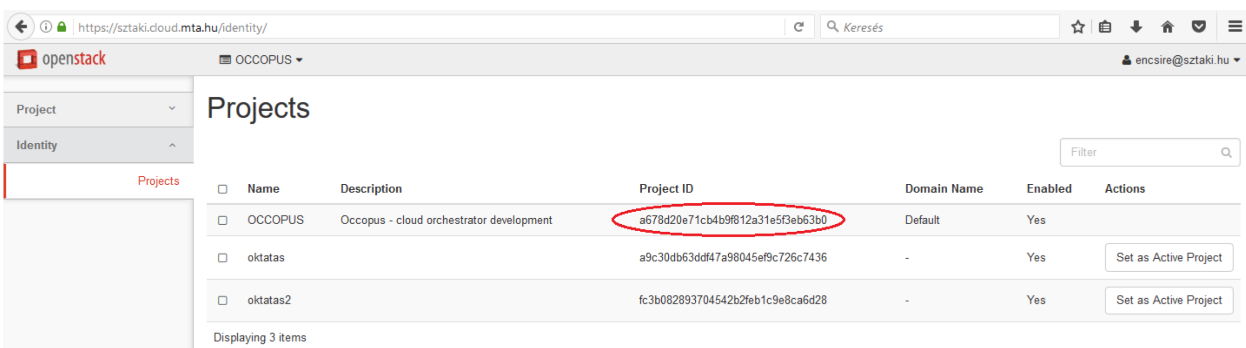


Note: The nova endpoint for MTA Cloud is: `https://sztaki.cloud.mta.hu:5000/v3`.

---

project_id

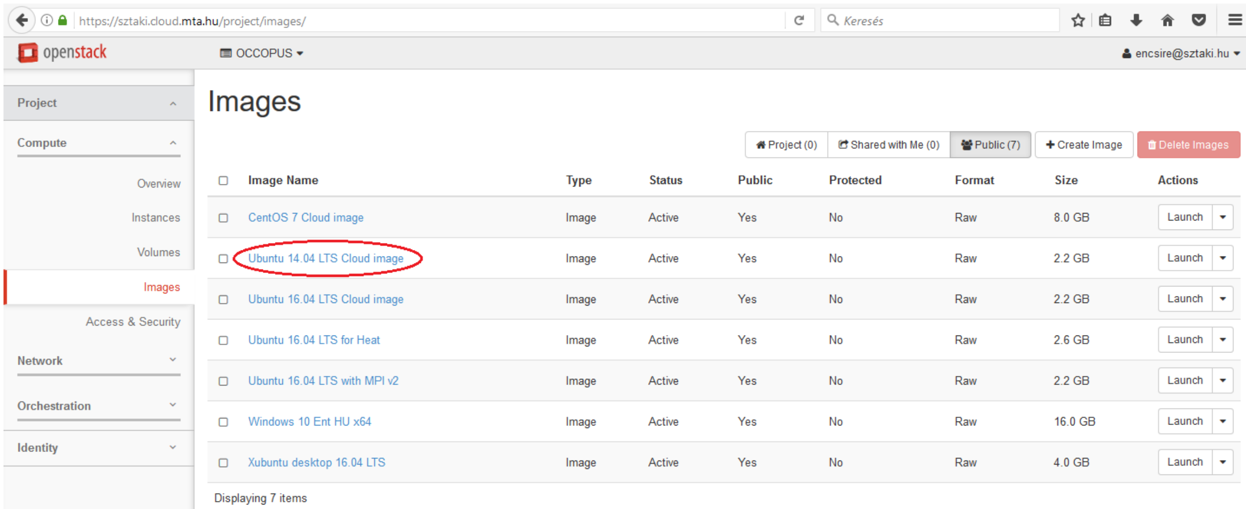
The value of this attribute is an ID referring to a project registered under your account. Select `Identity/Projects` menu and a full list of available projects will be listed. Select the proper project and copy its ID found at the *Project ID* column.



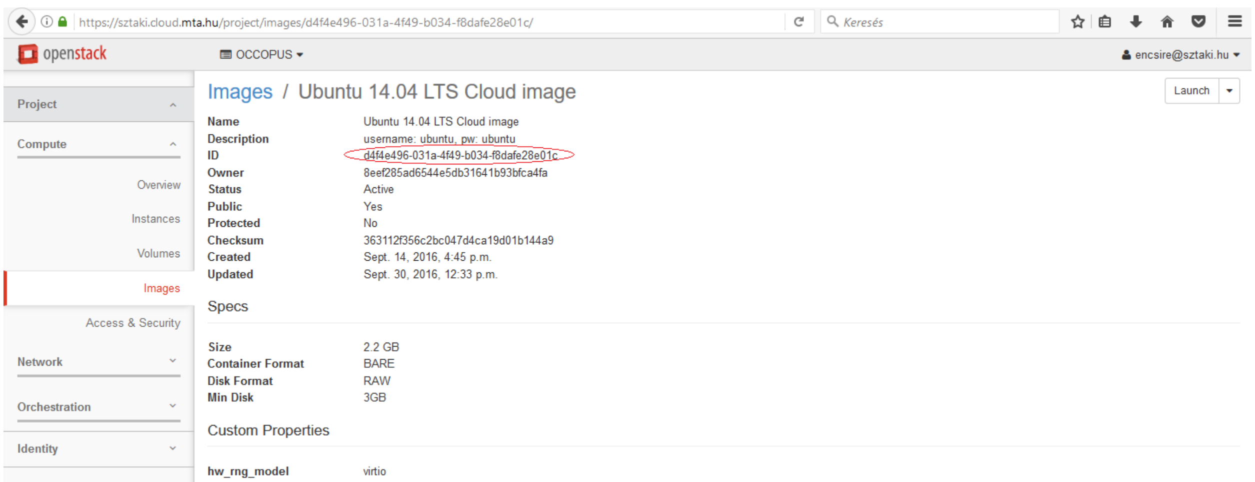
We have chosen the OCCOPUS project for which the `project_id` attribute in the resource section will be `a678d20e71cb4b9f812a31e5f3eb63b0`.

image_id

The value of this attribute is an ID referring to an image on the cloud to be instantiated to realize a virtual machine. Select `Project/Compute/Images` menu and a full list of available images will be listed.



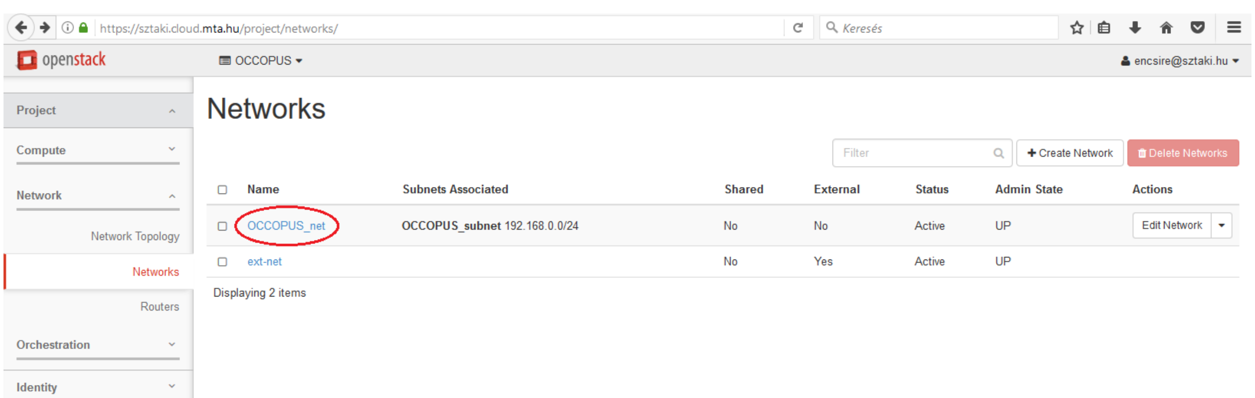
Assuming we need an `Ubuntu 14.04 LTS`, click on the name of the image. On the appearing page the ID attribute contains the value we are looking for.



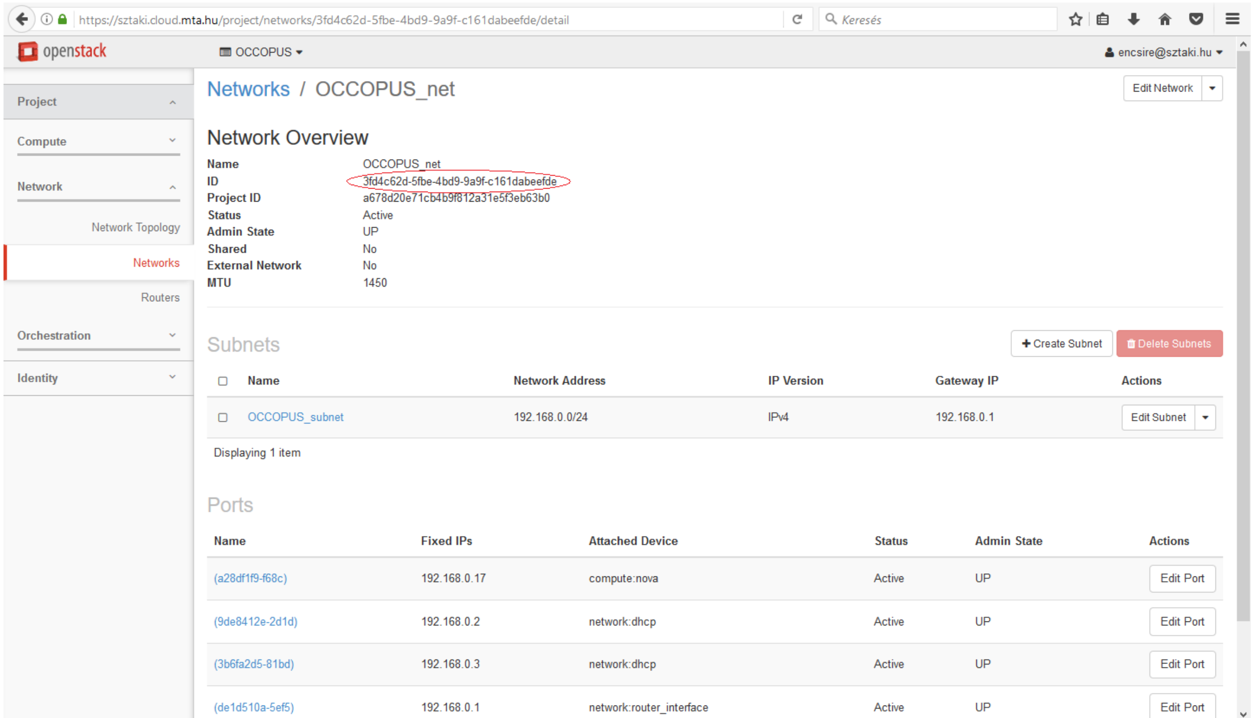
As a result, the `image_id` attribute in the resource section will be `d4f4e496-031a-4f49-b034-f8dafe28e01c`.

network_id

The value of this attribute is an ID refering to the ID of the network to attach to the virtual machine. Select `Project/Network/Networks`. On this page you can see the list of available networks of your project.



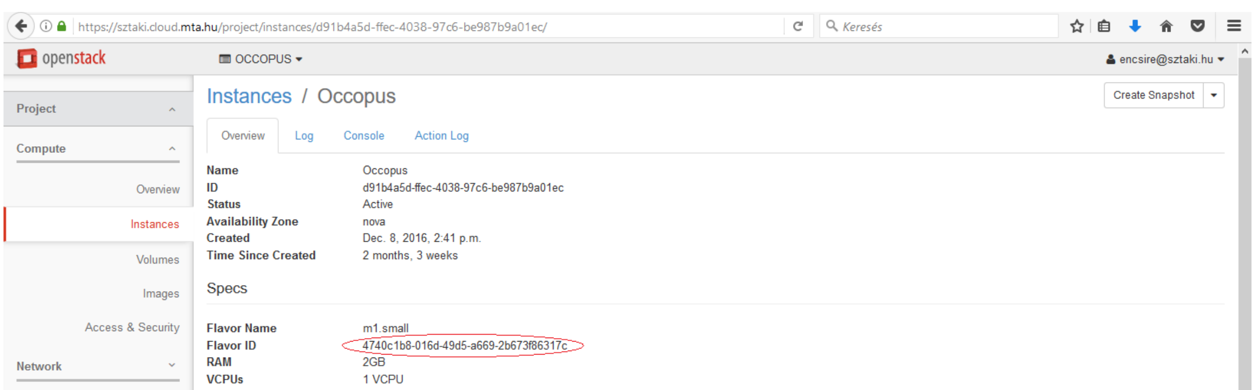
Assuming we need the OCCOPUS_net network, select and click on the network. On the appearing page the ID attribute contains the value we are looking for.



As a result, the `network_id` attribute in the resource section will be `3fd4c62d-5fbe-4bd9-9a9f-c161dabeefde`.

flavor_name

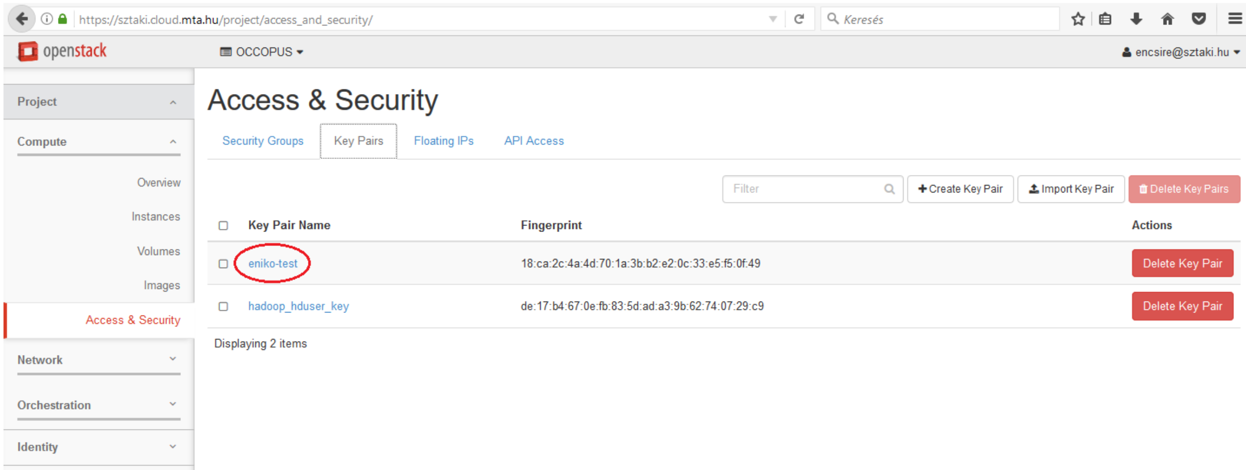
The value of this attribute is the ID referring to the type of flavor to be instantiated through nova when realizing a virtual machine. It determines the resources (CPU, memory, storage, networking) of the node. Unfortunately flavor IDs cannot be listed on the webpage, but they can be seen on an instance's overview page (Choose the `Project/Compute/Instances` menu and select one of your instances).



Note: For MTA Cloud users the following flavor IDs are defined: m1.small („4740c1b8-016d-49d5-a669-2b673f86317c"), m1.medium („3"), m1.large („4"), m1.xlarge („41316ba3-2d8b-4099-96d5-efa82181bb22")

key_name

The value of this attribute is a name refering to a particular public key registered under your account. To register a new ssh keypair, generate or upload one on page under the `Project/Compute/Access&Security/Key Pairs` menu.

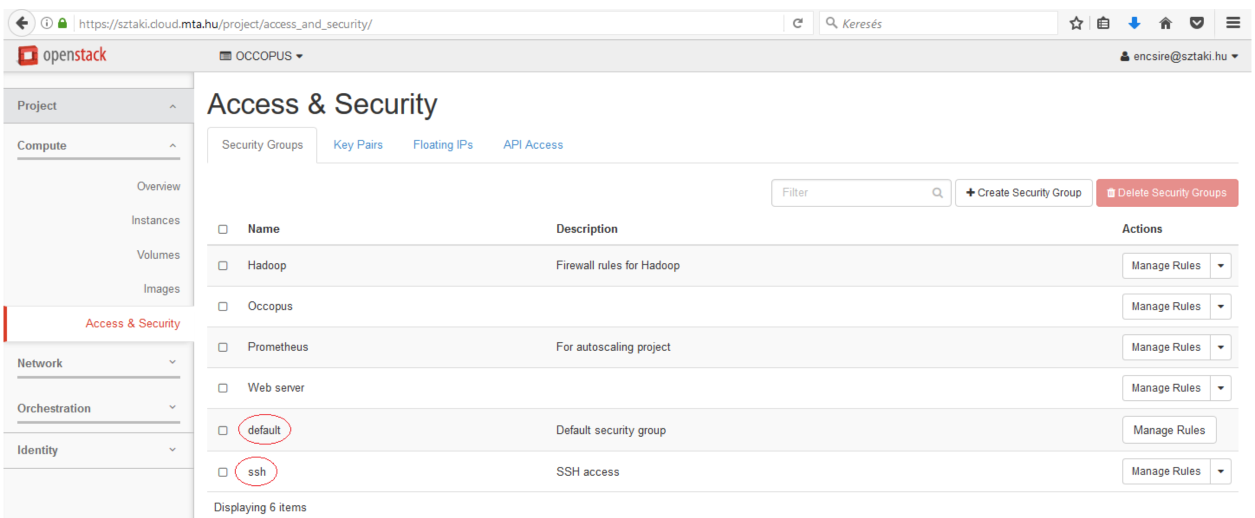


On this page you can see the list of registered keys and their fingerprint. Copy the name of your key from the *Key Pair Name* column.

As a result, the `key_name` attribute in the resource section will be `eniko-test`.

security_groups

The value of this attribute is a list of security groups referring to particular firewall policies registered under your project. To register a new firewall policy, use the page under the `Project/Compute/Access & Security` menu. On this page you can see the list of registered firewall policies.



As a result, the `security_groups` attribute in the resource section will be `default` and `ssh`. In ssh policy, port 22 is open.

The finalised resource section with the IDs collected in the example above will look like this:

```
resource:
  type: nova
  endpoint: https://sztaki.cloud.mta.hu:5000/v3
  project_id: a678d20e71cb4b9f812a31e5f3eb63b0
  user_domain_name: Default
  image_id: d4f4e496-031a-4f49-b034-f8dafe28e01c
  network_id: 3fd4c62d-5fbe-4bd9-9a9f-c161dabeefde
  flavor_name: 3
```

(continues on next page)

(continued from previous page)

```
  key_name: eniko-test
  security_groups: [ default, ssh]
  floating_ip: yes
  floating_ip_pool: ext-net
contextualisation:
  ...
```

Important: Collect the IDs under your account instead of using the ones in this example!

Important: The resource section must follow YAML syntax! Make sure indentation is proper, avoid using <tab>, use spaces!

2.5.3.3 Contextualisation

In this section, the attributes (keywords) are listed and explained which can be used for the different contextualisation plugins.

Cloudinit

type: cloudinit Selects the cloudinit contextualisation plugin. Can be used with the following resource handlers: ec2, nova, cloudsigma, azure.

context_template This section can contain a cloud init configuration template. It must follow the syntax of cloud-init. See the Cloud-init website for examples and details. Please note that Amazon AWS currently limits the length of this data in 16384 bytes.

attributes Optional. Any user-defined attributes. Used for specifying values of attributes in chef recipes.

Docker

type: docker Selects the docker contextualisation plugin. Can be used with the following resource handlers: docker, azure_aci.

env Environment variables to be passed to containers.

command Command to be executed inside the container once the container come to life. In case of the azure_aci resource handler, this is required to be a list.

2.5.3.4 Contextualisation variables and methods

Contextualization plugins

In Occopus, each node can have contextualization which is processed at the startup phase during building the node. Occopus has a pluggable contextualization module, currently there are plugins called "cloudinit" and "docker". The docker contextualization plugin can be used with docker containers to specify command, environment variables, etc. The cloudinit contextualization plugin can be used to specify user data passed to the cloud-init tool on the launched virtual machine. The required keywords for activating the cloudinit contextualization plugin is described in the manual.

Cloud-init plugin

The contextualization script for the cloudinit plugin can be dynamically updated with information Occopus has on the infrastructure and on its living nodes. The script may contain references to constants or even to methods which represent placeholders for dynamically resolvable strings. The script containing these placeholders are considered as template. Occopus performs the resolution of the template just before starting the virtual machine and passes it as user data to the Cloud API.

Jinja2 templating

For handling the contextualization script as template, Occopus uses Jinja2. Jinja2 is a designer-friendly full featured template engine. For detailed information on Jinja2, visit the website at Jinja Desinger Documentation. Since the content of the contextualization can be considered as a Jinja2 template, Jinja syntax can be used. The details of the syntax can be found on the Jinja webpage, however we provide a short summary for the simplest cases.

General rules

A template contains variables and/or expressions, which get replaced with values when a template is rendered. For variables and/or expressions a pair of double brackets (e.g. "{{foo}}") can be used, while statements are marked with bracket-percentage pair (e.g. "{% for item in seq %}").

How/where to define own variables

Variables can be defined in Occopus in the infrastructure description in the node description or at global level. Variables can be defined to be valid only for a given node (see "foo" variable in the code below) or can be defined to be valid in the entire infrastructure (see "bar" variable in the code below).

myinfra.yaml:

```
nodes:
    -
        name: mynode
        ...
        variables:
            foo: local
variables:
    bar: global
```

How/where to refer to own variables

In the text/yaml file containing the contextualisation/ user data, one may refer to predefined variables in the following way:

mycloudinit_context_file:

```
write_files:
- content: "foo: {{variables.foo}}\nbar: {{variables.bar}}\n"
  path: /tmp/myvars.txt:
```

As a result the cloud-init will create the following content:

/tmp/myvars.txt:

```
foo: local
bar: global
```

Enable/disable jinja syntax

If you do not want Jinja to process a part of your text, put your text between the following two jinja commands. As a result Jinja will ignore to translate the text within this section.

```
{% raw %}
...
{% endraw %}
```

System level constants and methods

Constants:

infra_name string containing the name of the infrastructure (as defined in infra description)

infra_id string containing the identifier of the infrastructure (generated by Occopus or user defined)

name string containing the name of the node (as defined in infra description)

node_id string containing the identifier of the node instance (uuid generated by Occopus)

Methods:

getip

- Usage: getip(<name of node defined in infra description>)
- Output: string containing an ip address of the (first) instance of the given node
- Example: getip(„master"), getip(variables.masterhostname)

getprivip

- Usage: getprivip(<name of node defined in infra description>)
- Output: string containing a private ip address of the (first) instance of the given node
- Example: getprivip(„master"), getprivip(variables.masterhostname)

getipall

- Usage: getipall(<name of node defined in infra description>)

- Output: string list containing ip addresses of the instances of the given node
- Example: getipall(„master"), getipall(variables.masterhostname)

cut

- Usage: cut(<string to be sliced>,<startindex>,<endindex>)
- Output: substring of the given string between the indexes
- Example: cut(infra_id,0,7)

cmd

- Usage: cmd('command with options')
- Output: string returned by the command
- Example: cmd('curl -X GET http://localhost/message.txt'), cmd('cat /etc/hosts')

2.5.3.5 Config management

In this section, the attributes (keywords) are listed and explained which can be used for the different config manager plugins.

Chef

type: chef Selects chef as config manager.

endpoint The endpoint (url) of the chef server containing the recipes.

run_list The list of recipes to be executed by chef on the node after startup.

Puppet-solo

type: puppet_solo Selects puppet (server-free version) as config manager.

manifests The location (url) of the puppet manifest files to be deployed.

modules The list module names to be of deployed by puppet.

attributes List of attribute-value pairs defining the values of the attributes.

2.5.3.6 Health-check

In this section, the attributes (keywords) are listed and explained which can be used for to specify the way of health monitoring of the node.

Ping

```
ping: True
```

Optional. Health check includes ping test against the node if turned on. Default is on.

Ports

```
ports:
    - 22
    - 1234
```

Optional. Health check includes testing against open ports if list of ports are specified. Default is none.

Urls

```
urls:
    - http://{{ip}}:5000/myserviceOne
    - http://{{ip}}:6000/myserviceTwo
```

Optional. Health check includes testing against web services if urls are specified. Default is none. The `{{ip}}` in the url means the ip address of the node being specified.

MysqlDBs

```
mysqldbs:
    - { name: 'mydbname1', user: 'mydbuser1', pass: 'mydbpass1' }
    - { name: 'mydbname2', user: 'mydbuser2', pass: 'mydbpass2' }
```

Optional. Health check includes testing available and accessible mysql database connection if name, user, pass triples are specified. Default is none. If specified mysql database connecticity check is performed with the given parameters.

Timeout

```
timeout: 600
```

Optional. Specifies a period in seconds after which continuous failure results in the node considered as failed. The current protocol in Occopus is to restart failed nodes. Default is 600.

2.5.3.7 Multiple node implementations

When creating node definitions, you can create multiple implementations for the same node. These implementations can differ in any parameter listed in the sections before, including but not limited to: resource backend, image id, instance type, contextualization, configuration management, health-check services used, etc. To create multiple implementations, just list them using hyphens. Make sure to watch for the indentation of the blocks.

The following example shows a node definition with multiple different implementations.

```
'node_def:example_node':
    -
        resource:
            name: my_opennebula_ec2
            type: ec2
            endpoint: my_opennebula_endpoint
            ...
        ...
        config_management:
            type: chef
            ...
    -
        resource:
            name: my_aws_ec2
            type: ec2
            endpoint: my_aws_endpoint
            ...
        ...
    -
        resource:
            name: my_nova
            type: nova
            endpoint: my_nova_endpoint
            ...
        ...
        config_management:
            type: puppet_solo
            ...
        ...
```

If there are multiple implementations for a node definition, you can filter them in the *Node description*, in the *Infrastructure description* file. Occopus will automatically select an available implementation to launch from those fulfilling the filtering parameters.

2.5.3.8 Examples

Examples can be found in the *tutorial section* of the User Guide.

2.6 Usage

2.6.1 Command line tools

Occopus can be used via CLI commands to build, maintain, scale and destroy infrastructures. The commands and their usages are described below.

2.6.1.1 occopus-build

This command deploys an infrastructure based on an infrastructure description.

On error during creating the infrastructure it rolls back everything to the initial state. The user can also stop the process manually by executing a SIGINT (Ctrl + C). Allocation of resources will be rolled back in this case as well.

Once the infrastructure is successfully built, Occopus exits. This command provides no lifecycle-management.

Usage:

```
occopus-build [-h]
              [--cfg CFG_PATH]
              [--auth_data_path AUTH_DATA_PASS]
              [--parallelize]
              [-i INFRA_ID]
              infra_def
```

Parameters:

- `-h, --help:` (optional) shows help message
- `--cfg CFG_PATH:` (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--auth_data_path AUTH_DATA_PATH:` (optional) path to Occopus authentication file (default: None) if undefined, file named *auth_data.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--parallelize:` (optional) parallelize processing instructions e.g. independent nodes are created parallel (default: sequential)
- `-i INFRA_ID:` (optional) identifier of a previously built, existing infrastructure - if provided, occopus will reconfigure the existing infrastructure instead of building a new one. Use with caution! Occopus may build/destroy nodes based on the difference between the existing and the new infrastructure defined by `infra_def`!
- `infra_def:` file containing an infrastructure definition to be built

Return type: On successful finish it returns the identifier of the infrastructure. The identifier can be stored or listed by the occopus-maintain command.

2.6.1.2 occopus-destroy

This command destroys an infrastructure including all of its nodes built previously by Occopus. No recover is possible.

Usage:

```
occopus-destroy [-h]
                [--cfg CFG_PATH]
                [--auth_data_path AUTH_DATA_PATH]
                -i INFRA_ID
```

Parameters:

- `-h, --help:` (optional) shows help message
- `--cfg CFG_PATH:` (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--auth_data_path AUTH_DATA_PATH:` (optional) path to Occopus authentication file (default: None) if undefined, file named *auth_data.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `-i INFRA_ID:` identifier of the infrastructure to destroy

2.6.1.3 occopus-maintain

This command is capable of maintaining an infrastructure built by Occopus. Maintenance includes health checking, recovery and scaling. It can also list available infrastructure or can provide details on an infrastructure.

Usage:

```
occopus-maintain [-h]
                 [--cfg CFG_PATH]
                 [--auth_data_path AUTH_DATA_PATH]
                 [--parallelize]
                 [-l|--list]
                 [-r|--report]
                 [-i INFRA_ID]
                 [-c|--cyclic]
                 [-t INTERVAL]
                 [-o|--output OUTPUT]
                 [-f|--filter FILTER]
```

Parameters:

- `-h, --help:` (optional) shows help message
- `--cfg CFG_PATH:` (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--auth_data_path AUTH_DATA_PATH:` (optional) path to Occopus authentication file (default: None) if undefined, file named *auth_data.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--parallelize:` (optional) parallelize processing instructions e.g. independent nodes are created parallel (default: sequential)
- `-l, --list:` (optional) list the built pieces of infrastructure
- `-r, --report:` (optional) reports about an infrastructure
- `-i INFRA_ID:` (optional) identifier of the infrastructure to maintain

- `-c, --cyclic`: (optional) performs continuous maintenance
- `-t INTERVAL`: (optional) specifies the time in seconds between maintenance sessions (default: 10)
- `-o OUTPUT`: (optional) defines output file name for reporting on an infra (default: None)
- `-f FILTER`: (optional) defines the nodename to be included in reporting (default: None)

2.6.1.4 occopus-scale

This command registers scaling requests for a given node in an infrastructure. With scaling the instance count of a node can be increased or decreased by a given number. Scaling requests are handled and realized by the occopus-maintain command.

Usage:

```
occopus-scale [-h]
              [--cfg CFG_PATH]
              [--auth_data_path AUTH_DATA_PATH]
              -i INFRA_ID
              -n|--node NODE
              [-c|--changescale CHANGESCALE]
              [-s|--setscale SETSCALE]
              [-f|--filter FILTER]
```

Parameters:

- `-h, --help`: (optional) shows help message
- `--cfg CFG_PATH`: (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--auth_data_path AUTH_DATA_PATH`: (optional) path to Occopus authentication file (default: None) if undefined, file named *auth_data.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `-i INFRA_ID`: identifier of the infrastructure which contains the node to scale
- `-n NODE, --node NODE`: name of the node to scale
- `-c CHANGESCALE, --changescale CHANGESCALE`: positive/negative number expressing the direction and magnitude of scaling (positive: scale up; negative: scale down)
- `-s SETSCALE, --setscale SETSCALE`: positive number expressing the number of nodes to scale to
- `-f FILTER, --filter FILTER`: filter for selecting nodes for downscaling; filter can be nodeid or ip address (default: None)

2.6.1.5 occopus-import

This command imports i.e. loads the node definitions from file to the database of Occopus.

Important: Each time a node definition file changes, this command must be executed since Occopus takes node definitions from its database!

Usage:

```
occopus-import [-h]
               [--cfg CFG_PATH]
               datafile
```

Parameters:

- `-h, --help:` (optional) shows help message
- `--cfg CFG_PATH:` (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `datafile:` file containing node definition(s)

2.6.1.6 occopus-rest-service

This command launches occopus as a web service. The occopus rest service can create, maintain, scale and destroy any infrastructure built by the service. This service provides a restful interface described by *REST API*.

Usage:

```
occopus-rest-service [-h]
                     [--cfg CFG_PATH]
                     [--auth_data_path AUTH_DATA_PATH]
                     [--host HOST]
                     [--port PORT]
                     [--parallelize]
```

Parameters:

- `-h, --help:` (optional) shows help message
- `--cfg CFG_PATH:` (optional) path to Occopus config file (default: None) if undefined, file named *occopus_config.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--auth_data_path AUTH_DATA_PATH:` (optional) path to Occopus authentication file (default: None) if undefined, file named *auth_data.yaml* is searched at predefined locations, e.g. $HOME/.occopus
- `--host HOST:` (optional) sets the host for the service to be assigned to (default: 127.0.0.1)
- `--port PORT:` (optional) sets the port for the service to be assigned to (default: 5000)
- `--parallelize:` (optional) parallelize processing instructions (default: sequential)

2.6.2 REST API

2.6.2.1 POST /infrastructures/

Create a new infrastructure and returns the identifier of the infrastructure. The returned identifier can be used as `infraid` parameter in the infrastructure-related commands.

Requires an *infrastructure description* as POST data.

Return type:

```
{
    "infraid": "<infraid_in_uuid_format>"
}
```

Example:

```
curl -X POST http://127.0.0.1:5000/infrastructures/ --data-binary @my_infrastructure_
↪description.yaml
```

2.6.2.2 GET /infrastructures/

List the identifier of infrastructures currently maintained by the service.

Return type:

```
{
    "infrastructures": [
        "<infraid_in_uuid_format_for_an_infrastructure>",
        "<infraid_in_uuid_format_for_another_infrastructure>"
        ]
}
```

2.6.2.3 POST /infrastructures/(infraid)/scaledown/(nodename)/(nodeid)

Scales down a node in an infrastructure by destroying one of its instances specified.

Parameters:

- `infraid` The identifier of the infrastructure.
- *nodename:* The name of the node which is to be scaled down.
- *nodeid:* The identifier of the selected instance.

Return type:

```
{
    "infraid": "<infraid>",
    "method": "scaledown",
    "nodeid": "<nodeid>",
    "nodename": "<nodename>"
}
```

2.6.2.4 POST /infrastructures/(infraid)/scaleup/(nodename)/(int: count)

Scales up a node in an infrastructure by creating the specified number of instances of the node.

Parameters:

- `infraid:` The identifier of the infrastructure.
- `nodename:` The name of the node to be scaled up.
- `count:` The number of instances to be created.

Return type:

```
{
    "count": "<count>",
    "infraid": "<infraid>",
    "method": "scaleup",
    "nodename": "<nodename>"
}
```

2.6.2.5 POST /infrastructures/(infraid)/scaleto/(nodename)/(int: count)

Scales a node in an infrastructure to a given count by creating or destroying instances of the node depending on the actual number of instances and the required number.

Parameters:

- `infraid`: The identifier of the infrastructure.
- `nodename`: The name of the node to be scaled up.
- `count`: The number of instances to scale the node to.

Return type:

```
{
    "count": "<count>",
    "infraid": "<infraid>",
    "method": "scaleto",
    "nodename": "<nodename>"
}
```

2.6.2.6 POST /infrastructures/(infraid)/scaledown/(nodename)

Scales up a node in an infrastructure by creating a new instance of the node.

Parameters:

- `infraid`: The identifier of the infrastructure.
- `nodename`: The name of the node to be scaled up.

Return type:

```
{
    "count": 1,
    "infraid": "<infraid>",
    "method": "scaleup",
    "nodename": "<nodename>"
}
```

2.6.2.7 POST /infrastructures/(infraid)/scaleup/(nodename)

Scales up a node in an infrastructure by creating a new instance of the node.

Parameters:

- `infraid`: The identifier of the infrastructure.
- `nodename`: The name of the node to be scaled up.

Return type:

```
{
    "count": 1,
    "infraid": "<infraid>",
    "method": "scaleup",
    "nodename": "<nodename>"
}
```

2.6.2.8 POST /infrastructures/(infraid)/attach

Starts maintaining an existing infrastructure.

Parameters:

- `infraid`: The identifier of the infrastructure.

Return type:

```
{
    "infraid": "<infraid>"
}
```

2.6.2.9 POST /infrastructures/(infraid)/detach

Stops maintaining an infrastructure.

Parameters:

- `infraid`: The identifier of the infrastructure.

Return type:

```
{
    "infraid": "<infraid>"
}
```

2.6.2.10 POST /infrastructures/(infraid)/notify

Sets notification properties for an infrastructure.

Parameters:

- `infraid`: The identifier of the infrastructure. Requires a notification description in JSON format as the POST data.

Return type:

```
{
    "infraid": "<infraid>",
}
```

2.6.2.11 GET /infrastructures/(infraid)

Report the details of an infrastructure.

Parameters:

- `infraid`: The identifier of the infrastructure.

Return type:

```
{
    "<nodename>": {
        "instances": {
            "<nodeid>": {
                "resource_address": "<ipaddress>",
                "state": "<state>"
            }
        },
        "scaling": {
            "actual": "<current_number_of_instances>",
            "max": "<maximum_number_of_instances>",
            "min": "<minimum_number_of_instances>",
            "target": "<target_number_of_instances>"
        }
    },
}
```

2.6.2.12 DELETE /infrastructures/(infraid)

Shuts down an infrastructure.

Parameters:

- `infraid`: The identifier of the infrastructure.

Return type:

```
{
    "infraid": "<infraid>"
}
```

2.6.3 Python API

Occopus provides a Python API which can be used to implement Occopus-based applications in a unified way. The API gives the possibility to utilise Occopus functionalities inside an application. To read about this possibility, please go to the *API section of the Developer guide*.

2.7 Release Notes

2.7.1 v1.10 (30 Nov 2021)

- Deprecate novaclient, relying on openstacksdk only in nova plugin
- Drop voms support, boot volume support in nova plugin
- Remove voms auth type checking for nova plugin
- Add support for booting from volumes in case of diskless flavors
- Add support for endpoints with/wo version number in nova plugin
- Add support to pass FQDN as environment variable in azure container plugin
- Add support for server naming and ssh key in azure vm plugin
- Add cost query to cloudbroker plugin and to API endpoint

2.7.2 v1.9 (25 May 2021)

- Fixing protocol id in azure vm plugin
- Fixing authentication check in the azure container and vm plugins
- Refactor floating ip handling in nova plugin
- Update nova client library to latest version in nova plugin

2.7.3 v1.8 (17 Aug 2020)

- Add Azure ACI (container) plugin
- Remove OCCI plugin

2.7.4 v1.7 (30 Apr 2020)

- Add Azure plugin
- Upgrade to Python 3
- Add reporting multiple node addresses
- Add -f parameter for downscale to select node
- Add ApplicationCredential auth type handling to nova
- Add region selection to nova
- Add downscale by any ip of the node

2.7.5 v1.6 (05 Apr 2019)

- New REST methods: notify, scaleto
- Export ip addresses by occopus-maintain
- Select youngest instance at downscaling when unspecified
- Added FCM event logging and notification
- Add cmd() and getprivip() methods in cloud-init resolution
- Add tagging for ec2 clouds
- Fixes in cloudsigma plugin (start server timeout, error code, ip retrieval)
- Fixes in docker plugin (version, dependencies, local image source support)
- Fixes in ec2 plugin (boto v2.48.0. with dependencies)

2.7.6 v1.5 (23 May 2017)

- Reimplemented cloudbroker plugin: handle instances, not jobs
- Remove cloud-broker node resolver (replaced by cloud-init)
- Add multiuser support in handling redis server
- Improve error handling and logging in cloudsigma, ec2 and occi plugins
- Improve nova plugin to handle interruption
- Add infra and node name syntax checking
- Add new Occopus installer script
- Improve parallel node creation

2.7.7 v1.4 (27 March 2017)

- Improve node handling in cloudsigma plugin
- Improve floating ip handling in nova plugin
- Precise syntax error reporting for descriptors
- Unique VM name for nodes as default
- Introduce user defined VM name templates
- Improve error/exception handling and reporting
- Fix logging and evaluation in schema checker
- Fix calculating default scaling min,max
- Restructure health-check reporting
- Deprecate 'network_mode' attribute in docker plugin
- Introduce attach and detach functions in rest
- Compatible REST and cmd-line functions

2.7.8 v1.3 (09 January 2017)

- New Puppet config-manager plugin: server-free, called "puppet_solo"
- Remove external redis config for occopus-import command
- Remove attribute dependency from plugins
- Reimplement floating_ip handling in nova plugin
- Fix bug in filtering
- New tutorial for puppet_solo plugin
- New tutorial to introduce autoscaling with prometheus

2.7.9 v1.2 (11 August 2016)

- Support for keystone v3 password-based authentication in Nova plugin
- Infrastructure dynamic reconfiguration
- More logs in occi plugin
- Small fixes

2.7.10 v1.1 (5 June 2016)

- New CloudSigma resource-handler plugin
- New tutorials for CloudSigma
- New getipall() method for cloud-init
- Fix yaml_import in node definition
- Bugfixes in plugins: occi, docker, nova

2.7.11 v1.0 (6 April 2016)

- Restructure node definition format
- Introduce schema checking
- Mixed config-manager support
- Refactor plugin names
- Reorganise config and authentication
- New authenticator selection mechanism
- New filtering mechanism in node description
- Simplification of health_check
- Introduce getip() in context templates
- Update occopus commands
- Introduce occopus-maintain command for maintenance
- Introduce occopus-scale command for scaling

- Support for multi infrastructure handling
- Refactor occopus-import command parameters

2.7.12 v0.3.0 (15 Jan 2016)

- introduce periodical service health checking
- new service health check mechanism: database check
- new service health check mechanism: port check
- add timeout for service unavailability
- improved nova plugin: voms based authorization
- new plugin: handling docker cluster
- new plugin: occi cloud interface for EGI FedClouds
- tutorials to demonstrate chef, docker and occi plugins
- node definition 'synch_strategy' keyword renamed to 'service_health_check'

2.7.13 v0.2.1 (10 Nov 2015)

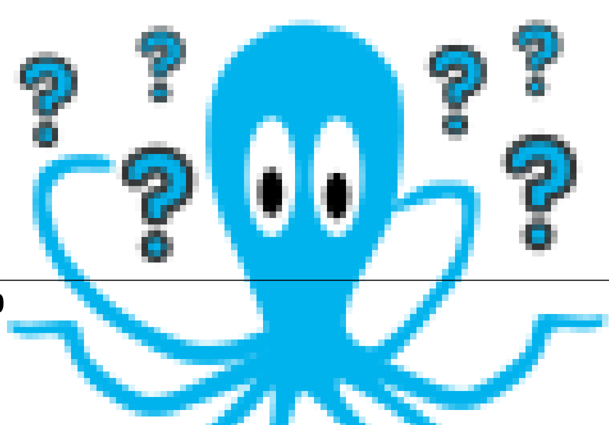
Improved EC2 handling:

- support for security group, subnet and keypairs in EC2 plugin
- two ec2 tutorials updated

2.7.14 v0.2.0 (4 Nov 2015)

- multi-cloud support
- basic command line utils and REST interface
- support for cloud interfaces: EC2, NOVA, CloudBroker
- support for configuration manager: Chef
- initial version of error detection and recovery
- manual scaling through REST API
- tutorials for EC2, NOVA and CloudBroker

2.8 Contact Us



Have any Occopus questions?

Feel free to contact us occopus **at** lists.lpds.sztaki **dot** hu or use our GitHub issues like a communication channel.

2.9 Resource plugins

In this section, simple examples will be shown. The examples will focus on introducing the different resource types and will have two categories:

- The helloworld examples will only show how a single node can be created and how very simple contextualisation information (e.g. message string) can be passed to a virtual machine (VM)
- The ping examples will focus on to introduce how dependency can be created and how can connection between two nodes can be built up by passing the ip of a node to another.

Please, note that the following examples require a properly configured Occopus, therefore we suggest to continue this section if you already followed the instructions written in the *Installation* section.

2.9.1 EC2-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

In this example, the following feature(s) will be demonstrated:

- creating a node with basic contextualisation
- using the ec2 resource handler

Prerequisites

- accessing a cloud through EC2 interface (access key, secret key, endpoint, regionname)
- target cloud contains a base OS image with cloud-init support (image id, instance type)

Download

You can download the example as tutorial.examples.ec2-helloworld .

Steps

1. Edit `nodes/node_definitions.yaml`. For `ec2_helloworld_node` set the followings in its `resource` section:

- `endpoint` is an url of an EC2 interface of a cloud (e.g. *https://ec2.eu-west-1.amazonaws.com*).
- `regionname` is the region name within an EC2 cloud (e.g. *eu-west-1*).
- `image_id` is the image id (e.g. *ami-12345678*) on your EC2 cloud. Select an image containing a base os installation with cloud-init support!
- `instance_type` is the instance type (e.g. *m1.small*) of your VM to be instantiated.
- `key_name` optionally specifies the keypair (e.g. *my_ssh_keypair*) to be deployed on your VM.
- `security_group` optionally specifies security settings (you can define multiple security groups in the form of a list, e.g. *sg-93d46bf7*) of your VM.
- `subnet_id` optionally specifies subnet identifier (e.g. *subnet-644e1e13*) to be attached to the VM.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:ec2_helloworld_node':
    -
        resource:
            type: ec2
            endpoint: replace_with_
↪endpoint_of_ec2_interface_of_your_cloud
            regionname: replace_
↪with_regionname_of_your_ec2_interface
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
    ␣
↪       instance_type: replace_with_instance_
↪type_of_your_image_on_your_target_cloud
            key_name:␣
↪replace_with_key_name_on_your_target_cloud
            security_group_ids:
                - replace_with_
↪security_group_id1_on_your_target_cloud
                - replace_with_
↪security_group_id2_on_your_target_cloud
            subnet_id:␣
↪replace_with_subnet_id_on_your_target_cloud
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your access key and secret key in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `ec2_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-ec2-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.2 EC2-Ping

This tutorial builds an infrastructure containing two nodes. The ping-sender node will ping the ping-receiver node. The sender node will store the outcome of ping in `/tmp` directory.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)
- querying a node's ip address and passing the address to another

- using the ec2 resource handler

Prerequisites

- accessing a cloud through EC2 interface (access key, secret key, endpoint, regionname)
- target cloud contains a base OS image with cloud-init support (image id, instance type)

Download

You can download the example as tutorial.examples.ec2-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `ec2_ping_receiver_node` and for `ec2_ping_sender_node` set the followings in their `resource` section:

- `endpoint` is an url of an EC2 interface of a cloud (e.g. *https://ec2.eu-west-1.amazonaws.com*).
- `regionname` is the region name within an EC2 cloud (e.g. *eu-west-1*).
- `image_id` is the image id (e.g. *ami-12345678*) on your EC2 cloud. Select an image containing a base os installation with cloud-init support!
- `instance_type` is the instance type (e.g. *m1.small*) of your VM to be instantiated.
- `key_name` optionally specifies the keypair (e.g. *my_ssh_keypair*) to be deployed on your VM.
- `security_group` optionally specifies security settings (you can define multiple security groups in the form of a list, e.g. *sg-93d46bf7*) of your VM.
- `subnet_id` optionally specifies subnet identifier (e.g. *subnet-644e1e13*) to be attached to the VM.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:ec2_ping_receiver_node':
    -
        resource:
            type: ec2
            endpoint: replace_with_
↪endpoint_of_ec2_interface_of_your_cloud
            regionname: replace_
↪with_regionname_of_your_ec2_interface
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
```

(continues on next page)

(continued from previous page)

```
     ␣
↪       instance_type: replace_with_instance_
↪type_of_your_image_on_your_target_cloud
            key_name:␣
↪replace_with_key_name_on_your_target_cloud
            security_group_ids:
                -
                    replace_with_
↪security_group_id1_on_your_target_cloud
                -
                    replace_with_
↪security_group_id2_on_your_target_cloud
            subnet_id:␣
↪replace_with_subnet_id_on_your_target_cloud
        ...
'node_def:ec2_ping_sender_node':
    -
        resource:
            type: ec2
            endpoint: replace_with_
↪endpoint_of_ec2_interface_of_your_cloud
            regionname: replace_
↪with_regionname_of_your_ec2_interface
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
     ␣
↪       instance_type: replace_with_instance_
↪type_of_your_image_on_your_target_cloud
            key_name:␣
↪replace_with_key_name_on_your_target_cloud
            security_group_ids:
                -
                    replace_with_
↪security_group_id1_on_your_target_cloud
                -
                    replace_with_
↪security_group_id2_on_your_target_cloud
            subnet_id:␣
↪replace_with_subnet_id_on_your_target_cloud
        ...
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your access key and secret key in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `ec2_ping_receiver_node` and `ec2_ping_sender_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g.

contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-ec2-ping.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of ip addresses:
ping-receiver:
    192.168.xxx.
↪xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
ping-sender:
    192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)

30f566d1-9945-42be-b603-795d604b362f
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/message.txt
Hello World!
↪ I am the sender node created by Occopus.
# cat /tmp/ping-result.txt
PING 192.168.xxx.
↪xxx (192.168.xxx.xxx) 56(84) bytes of data.
64 bytes from 192.
↪168.xxx.xxx: icmp_seq=1 ttl=64 time=2.74 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=2 ttl=64 time=0.793 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=3 ttl=64 time=0.865 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=4 ttl=64 time=0.882 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=5 ttl=64 time=0.786 ms

--- 192.168.xxx.xxx ping statistics ---
5 packets transmitted,
↪ 5 received, 0% packet loss, time 4003ms
rtt min/
↪avg/max/mdev = 0.786/1.215/2.749/0.767 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.9.3 Nova-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

- creating a node with basic contextualisation
- using the nova resource handler

Prerequisites

- accessing an OpenStack cloud through its Nova interface (username/pasword or X.509 VOMS proxy, endpoint, tenant_name or project_id and user_domain_name)
- id of network to be associated to the virtual machine (network_id)
- security groups to be associated to the virtual machine (security_groups)
- name of keypair on the target cloud to be associated with the vm (key_name)
- target cloud contains a base OS image with cloud-init support (image_id, flavor_name)
- optionally, name of floating ip pool from which ip should be taken for the vm (floating_ip_pool)

Download

You can download the example as tutorial.examples.nova-helloworld .

Steps

1. Edit `nodes/node_definitions.yaml`. For `nova_helloworld_node` set the followings in its `resource` section:

- `endpoint` must point to the endpoint (url) of your target Nova cloud.
- `project_id` is the id of project you would like to use on your target Nova cloud.
- `user_domain_name` is the user domain name you would like to use on your target Nova cloud.
- `image_id` is the image id on your Nova cloud. Select an image containing a base os installation with cloud-init support!

- `flavor_name` is the name of flavor to be instantiated on your Nova cloud.
- `server_name` optionally defines the hostname of VM (e.g.:"helloworld").
- `key_name` optionally sets the name of the keypair to be associated to the instance. Keypair name must be defined on the target nova cloud before launching the VM.
- `security_groups` optionally specifies security settings (you can define multiple security groups in the form of a list) for your VM.
- `floating_ip` optionally allocates new floating IP address to the VM if set to any value.
- `floating_ip_pool` optionally specifies the name of pool from which the floating ip must be selected.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:nova_helloworld_node':
    -
        resource:
            type: nova
            endpoint: replace_with_
↪endpoint_of_nova_interface_of_your_cloud
        ␣
↪  project_id: replace_with_projectid_to_use
            user_domain_name: Default
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
            network_id: replace_
↪with_id_of_network_on_your_target_cloud
            flavor_name: replace_
↪with_id_of_the_flavor_on_your_target_cloud
            server_name: myhelloworld
            key_name:␣
↪replace_with_name_of_keypair_or_remove
            security_groups:
                -
                    replace_with_
↪security_group_to_add_or_remove_section
            floating_ip:␣
↪add_yes_if_you_need_floating_ip_or_remove
            floating_ip_pool: replace_
↪with_name_of_floating_ip_pool_or_remove
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your username/password or in case of x509 voms authentication the

path of your VOMS proxy in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `nova_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-nova-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    aaa.bbb.ccc.
↪ddd (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.4 Nova-Ping

This tutorial builds an infrastructure containing two nodes. The ping-sender node will ping the ping-receiver node. The sender node will store the outcome of ping in `/tmp` directory.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)

- querying a node's ip address and passing the address to another
- using the nova resource handler

Prerequisites

- accessing an OpenStack cloud through its Nova interface (username/pasword or X.509 VOMS proxy, endpoint, tenant_name or project_id and user_domain_name)
- id of network to be associated to the virtual machine (network_id)
- security groups to be associated to the virtual machine (security_groups)
- name of keypair on the target cloud to be associated with the vm (key_name)
- target cloud contains a base OS image with cloud-init support (image_id, flavor_name)
- optionally, name of floating ip pool from which ip should be taken for the vm (floating_ip_pool)

Download

You can download the example as tutorial.examples.nova-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `nova_ping_receiver_node` and for `nova_ping_sender_node` set the followings in their `resource` section:

- `endpoint` must point to the endpoint (url) of your target Nova cloud.
- `project_id` is the id of project you would like to use on your target Nova cloud.
- `user_domain_name` is the user domain name you would like to use on your target Nova cloud.
- `image_id` is the image id on your Nova cloud. Select an image containing a base os installation with cloud-init support!
- `flavor_name` is the name of flavor to be instantiated on your Nova cloud.
- `server_name` optionally defines the hostname of VM (e.g.:"helloworld").
- `key_name` optionally sets the name of the keypair to be associated to the instance. Keypair name must be defined on the target nova cloud before launching the VM.
- `security_groups` optionally specifies security settings (you can define multiple security groups in the form of a list) for your VM.

- `floating_ip` optionally allocates new floating IP address to the VM if set to any value.
- `floating_ip_pool` optionally specifies the name of pool from which the floating ip must be selected.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:nova_ping_receiver_node':
    -
        resource:
            type: nova
            endpoint: replace_with_
↪endpoint_of_nova_interface_of_your_cloud
         ␣
↪   project_id: replace_with_projectid_to_use
            user_domain_name: Default
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
            network_id: replace_
↪with_id_of_network_on_your_target_cloud
            flavor_name: replace_
↪with_id_of_the_flavor_on_your_target_cloud
            server_name: mypingreceiver
            key_name:␣
↪replace_with_name_of_keypair_or_remove
            security_groups:
                -
                    replace_with_
↪security_group_to_add_or_remove_section
            floating_ip:␣
↪add_yes_if_you_need_floating_ip_or_remove
            floating_ip_pool: replace_
↪with_name_of_floating_ip_pool_or_remove
        ...
'node_def:nova_ping_sender_node':
    -
        resource:
            type: nova
            endpoint: replace_with_
↪endpoint_of_nova_interface_of_your_cloud
         ␣
↪   project_id: replace_with_projectid_to_use
            user_domain_name: Default
            image_id: replace_
↪with_id_of_your_image_on_your_target_cloud
            network_id: replace_
↪with_id_of_network_on_your_target_cloud
            flavor_name: replace_
↪with_id_of_the_flavor_on_your_target_cloud
```

(continues on next page)

(continued from previous page)

```
            server_name: mypingsender
            key_name:␣
↪replace_with_name_of_keypair_or_remove
            security_groups:
                -
                    replace_with_
↪security_group_to_add_or_remove_section
            floating_ip:␣
↪add_yes_if_you_need_floating_ip_or_remove
            floating_ip_pool: replace_
↪with_name_of_floating_ip_pool_or_remove
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your username/password or in case of x509 voms authentication the path of your VOMS proxy in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `nova_ping_receiver_node` and `nova_ping_sender_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-nova-ping.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of ip addresses:
ping-receiver:
    192.168.xxx.
↪xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
ping-sender:
    192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)

30f566d1-9945-42be-b603-795d604b362f
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/message.txt
Hello World!
↪ I am the sender node created by Occopus.
# cat /tmp/ping-result.txt
PING 192.168.xxx.
↪xxx (192.168.xxx.xxx) 56(84) bytes of data.
64 bytes from 192.
↪168.xxx.xxx: icmp_seq=1 ttl=64 time=2.74 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=2 ttl=64 time=0.793 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=3 ttl=64 time=0.865 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=4 ttl=64 time=0.882 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=5 ttl=64 time=0.786 ms

--- 192.168.xxx.xxx ping statistics ---
5 packets transmitted,
↪ 5 received, 0% packet loss, time 4003ms
rtt min/
↪avg/max/mdev = 0.786/1.215/2.749/0.767 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.9.5 Azure-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

- creating a node with basic contextualisation
- using the azure resource handler

Prerequisites

- accessing Microsoft Azure interface (Tenant ID, Client ID, Client Secret, Subscription ID)
- resource group name inside Azure
- location to use inside Azure
- virtual machine specifications (size, publisher, offer, sku and version)

Download

You can download the example as tutorial.examples.azure-helloworld .

Steps

1. Edit nodes/node_definitions.yaml. For azure_helloworld_node set the followings in its resource section:

- resource_group must contain the name of the resource group to allocate resources in.
- location is the name of the location (region) to use.
- vm_size is the size of the VM to allocate.
- publisher is the name of the publisher of the image to use.
- offer is the offer of the image to use.
- sku is the sku of the image to use.
- version is the version of the image to use.
- username the name of the admin user to create.
- password the password to set for thr admin user.
- public_ip_needed optional, when set to True, a public IP is allocated for the resource.

Important: You can get help on collecting identifiers for the resources section at https://docs.microsoft.com/hu-hu/azure/developer/python/azure-sdk-authenticate. Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:azure_helloworld_node':
    -
      resource:
          type: azure_vm
          endpoint: https://management.azure.com
          resource_group: replace_with_resource_group_name
          location : replace_with_location
          vm_size: replace_with_vm_size
          publisher : replace_with_publisher_name
          offer : replace_with_offer
          sku : replace_with_sku
          version : replace_with_version
          username : replace_with_admin_username
          password : replace_with_admin_password
          # Optional - Existing VNet's name to use
```

(continues on next page)

(continued from previous page)

```
        #vnet_
↪name: replace_with_virtual_network_name
   ␣
↪     # Optional - Existing NIC's name to use
        #nic_name: replace_with_nic_name
        # Optional - Subnet name
   ␣
↪     #subnet_name: replace_with_subnet_name
        # Optional␣
↪- Set to True if public IP is needed
        #public_ip_needed : True
```

2. Make sure your authentication information is set correctly in your authentication file. Setting authentication information is described *here*.
3. Load the node definition for `azure_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-azure-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    aaa.bbb.ccc.
↪ddd (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.6 Azure-Ping

This tutorial builds an infrastructure containing two nodes. The ping-sender node will ping the ping-receiver node. The sender node will store the outcome of ping in `/tmp` directory.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)
- querying a node's ip address and passing the address to another
- using the azure resource handler

Prerequisites

- accessing Microsoft Azure interface (Tenant ID, Client ID, Client Secret, Subscription ID)
- resource group name inside Azure
- location to use inside Azure
- virtual machine specifications (size, publisher, offer, sku and version)

Download

You can download the example as tutorial.examples.azure-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `azure_ping_receiver_node` and for `azure_ping_sender_node` set the followings in their `resource` section:

- `resource_group` must contain the name of the resource group to allocate resources in.
- `location` is the name of the location (region) to use.
- `vm_size` is the size of the VM to allocate.
- `publisher` is the name of the publisher of the image to use.
- `offer` is the offer of the image to use.
- `sku` is the sku of the image to use.
- `version` is the version of the image to use.
- `username` the name of the admin user to create.
- `password` the password to set for thr admin user.

- `public_ip_needed` optional, when set to True, a public IP is allocated for the resource.

Important: You can get help on collecting identifiers for the resources section at https://docs.microsoft.com/hu-hu/azure/developer/python/azure-sdk-authenticate. Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:azure_ping_receiver_node':
    -
        resource:
        type: azure_vm
  ␣
↪      endpoint: https://management.azure.com
        resource_
↪group: replace_with_resource_group_name
        location : replace_with_location
        vm_size: replace_with_vm_size
   ␣
↪     publisher : replace_with_publisher_name
        offer : replace_with_offer
        sku : replace_with_sku
        version : replace_with_version
  ␣
↪      username : replace_with_admin_username
  ␣
↪      password : replace_with_admin_password
   ␣
↪     # Optional - Existing VNet's name to use
        #vnet_
↪name: replace_with_virtual_network_name
  ␣
↪      # Optional - Existing NIC's name to use
        #nic_name: replace_with_nic_name
        # Optional - Subnet name
  ␣
↪      #subnet_name: replace_with_subnet_name
        # Optional␣
↪- Set to True if public IP is needed
        #public_ip_needed : True
      ...
'node_def:azure_ping_sender_node':
    -
        resource:
        type: azure_vm
  ␣
↪      endpoint: https://management.azure.com
        resource_
↪group: replace_with_resource_group_name
        location : replace_with_location
        vm_size: replace_with_vm_size
```

(continues on next page)

(continued from previous page)

```
    publisher : replace_with_publisher_name
      offer : replace_with_offer
      sku : replace_with_sku
      version : replace_with_version
     username : replace_with_admin_username
     password : replace_with_admin_password
    # Optional - Existing VNet's name to use
      #vnet_name: replace_with_virtual_network_name
     # Optional - Existing NIC's name to use
      #nic_name: replace_with_nic_name
      # Optional - Subnet name
     #subnet_name: replace_with_subnet_name
      # Optional - Set to True if public IP is needed
      #public_ip_needed : True
```

2. Make sure your authentication information is set correctly in your authentication file. Setting authentication information is described *here*.
3. Load the node definition for `azure_ping_receiver_node` and `azure_ping_sender_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-azure-ping.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of ip addresses:
ping-receiver:
    192.168.xxx.xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
```

(continues on next page)

(continued from previous page)

```
ping-sender:
    192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)

30f566d1-9945-42be-b603-795d604b362f
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/message.txt
Hello World!
↪ I am the sender node created by Occopus.
# cat /tmp/ping-result.txt
PING 192.168.xxx.
↪xxx (192.168.xxx.xxx) 56(84) bytes of data.
64 bytes from 192.
↪168.xxx.xxx: icmp_seq=1 ttl=64 time=2.74 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=2 ttl=64 time=0.793 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=3 ttl=64 time=0.865 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=4 ttl=64 time=0.882 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=5 ttl=64 time=0.786 ms

--- 192.168.xxx.xxx ping statistics ---
5 packets transmitted,
↪ 5 received, 0% packet loss, time 4003ms
rtt min/
↪avg/max/mdev = 0.786/1.215/2.749/0.767 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.9.7 Azure-ACI-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

- creating a node with basic contextualisation
- using the azure_aci resource handler

Prerequisites

- accessing Microsoft Azure interface (Tenant ID, Client ID, Client Secret, Subscription ID)
- resource group name inside Azure
- location to use inside Azure

Download

You can download the example as tutorial.examples.azure-aci-helloworld .

Steps

1. Edit `nodes/node_definitions.yaml`. For `azure_aci_helloworld_node` set the followings in its `resource` section:

- `resource_group` must contain the name of the resource group to allocate resources in.
- `location` is the name of the location (region) to use.
- `memory` must contain the amount of memory to allocate for the container in GB (e.g. 1).
- `cpu_cores` must contain the amount of CPU cures to allocate for the container in GB (e.g. 1).

Important: You can get help on collecting identifiers for the resources section at https://docs.microsoft.com/hu-hu/azure/developer/python/azure-sdk-authenticate. Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:azure_aci_helloworld_node':
    -
        resource:
            type: azure_aci
    ␣
↪       endpoint: https://management.azure.com
            resource_
↪group: replace_with_resource_group_name
            location: replace_with_location
            memory: replace_with_memory
            cpu_cores: replace_with_cpu_cores
            os_type: linux
            image: alpine
            network_type: Private
            ports:
                - 8080
        contextualisation:
            type: docker
    ␣
↪       env: ["message={{variables.message}}"]
            command: ["sh
↪", "-c", "echo \"$message\" > /tmp/message.
↪txt; while true; do sleep 1000; done"]
```

(continues on next page)

(continued from previous page)

```
        health_check:
           ping: False
```

2. Make sure your authentication information is set correctly in your authentication file. Setting authentication information is described *here*.

3. Load the node definition for `azure_aci_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-azure-aci-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    aaa.bbb.ccc.
↪ddd (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on the Azure portal. When you open the Azure portal, you can find your container instance inside all resources. From there, you can navigate to the connect panel of the container, and can use /bin/sh to gain root shell access inside the running container:

```
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.8 Azure-ACI-Nginx

This tutorial builds an infrastructure containing two nodes. The nginx-client node will fetch the HTML content served by the nginx-server node, and store the outcome in the `/tmp` directory. The nginx-server node uses the Alpine Linux-based Nginx image from the Docker hub, whereas the nginx-client node is run on top of a stock Alpine Linux image, also from the Docker hub.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)
- querying a node's ip address and passing the address to another
- using the azure_aci resource handler

Prerequisites

- accessing Microsoft Azure interface (Tenant ID, Client ID, Client Secret, Subscription ID)
- resource group name inside Azure
- location to use inside Azure

Download

You can download the example as tutorial.examples.azure-aci-nginx .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `azure_aci_nginx_node` and for `azure_aci_client_node` set the followings in their `resource` section:

- `resource_group` must contain the name of the resource group to allocate resources in.
- `location` is the name of the location (region) to use.
- `memory` must contain the amount of memory to allocate for the container in GB (e.g. 1).
- `cpu_cores` must contain the amount of CPU cures to allocate for the container in GB (e.g. 1).

Important: You can get help on collecting identifiers for the resources section at https://docs.microsoft.com/hu-hu/azure/developer/python/azure-sdk-authenticate. Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:azure_aci_nginx_node':
    -
        resource:
        type: azure_aci
  ␣
↪     endpoint: https://management.azure.com
          resource_
↪group: replace_with_resource_group_name
          location: replace_with_location
          memory: replace_with_memory
          cpu_cores: replace_with_cpu_cores
          os_type: linux
          image: nginx:alpine
          network_type: Public
          ports:
             - 80
      ...
'node_def:azure_aci_client_node':
    -
        resource:
        type: azure_aci
  ␣
↪     endpoint: https://management.azure.com
          resource_
↪group: replace_with_resource_group_name
          location: replace_with_location
          memory: replace_with_memory
          cpu_cores: replace_with_cpu_cores
          os_type: linux
          image: alpine
          network_type: Public
          ports:
             - 8080
```

2. Make sure your authentication information is set correctly in your authentication file. Setting authentication information is described *here*.

3. Load the node definition for `azure_aci_nginx_node` and `azure_aci_client_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-azure-aci-nginx.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of ip addresses:
nginx-server:
    192.168.xxx.
↪xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
nginx-client:
    192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)

30f566d1-9945-42be-b603-795d604b362f
```

#. Check the result on the Azure portal. When you open the Azure portal, you can find your container instances inside all resources. From there, you can navigate to the connect panel of the nginx-client container, and can use /bin/sh to gain root shell access inside the running container:

```
/ # ls -1 /tmp
message.txt
nginx_content.html
/ # cat /tmp/message.txt
Hello World!
↪ I am the client node created by Occopus.
```

1. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.9.9 Docker-Helloworld

This tutorial builds an infrastructure containing a single node implemented by a Docker container. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/root/message.txt` file.

Features

- creating a node with basic contextualisation
- using the docker resource handler

Prerequisites

- accessing a Docker host or a Swarm cluster (endpoint)
- having a docker image to be instantiated or using the one predefined in this example (origin, image)

- command to be executed on the image and the required environment variables or using the one predefined in this example (command, environment variables)

Important: Encrypted connection is not supported yet!

Download

You can download the example as tutorial.examples.docker-helloworld .

Steps

1. Edit `nodes/node_definitions.yaml`. For `docker_helloworld_node` set the followings in its `resource` section:

- `endpoint` is the endpoint of your docker cluster (e.g. *tcp://1.2.3.4:2375* or *unix://var/run/docker.sock*).

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:docker_helloworld_node':
    -
        resource:
            type: docker
            ␣
↪endpoint: replace_with_your_docker_endpoint
            origin: https://s3.lpds.
↪sztaki.hu/docker/busybox_helloworld.tar
            image: busybox_helloworld
            tag: latest
```

2. Make sure your authentication information is set correctly in your authentication file. The docker plugin in Occopus does not apply authentication, however a dummy authentication block is needed. The instructions for setting the authentication properly is described at the *authentication page*. There you can download a default authentication file containing the docker section already.

3. Load the node definition for `docker_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-docker-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
# docker ps
CONTAINER ID ␣
↪       IMAGE                       COMMAND ␣
↪                  CREATED              STATUS␣
↪              PORTS               NAMES
13bb8c94b5f4        busybox_
↪helloworld:latest   "sh -c /root/start.
↪sh"   3 seconds ago       Up 2 seconds␣
↪                             admiring_joliot

# docker␣
↪exec -it 13bb8c94b5f4 cat /root/message.txt
Hello World! I have been created by Occopus.
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.10 Docker-Ping

This tutorial builds an infrastructure containing a two nodes implemented by Docker containers. The ping-sender node will ping the ping-receiver node to demonstrate the connection between the two nodes. The sender node will store the outcome of ping in `/root/ping-result.txt` file.

Features

- creating two nodes with dependencies (i.e ordering or deployment)

- querying a node's ip address and passing the address to another
- using the docker resource handler

Prerequisites

- accessing a Docker host or a Swarm cluster (endpoint)
- having a docker image to be instantiated or using the one predefined in this example (origin, image)
- command to be executed on the image and the required environment variables or using the one predefined in this example (command, env)

Important: Encrypted connection is not supported yet!

Download

You can download the example as tutorial.examples.docker-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `docker_ping_receiver_node` and for `docker_ping_sender_node` set the followings in their `resource` section:

- `endpoint` is the endpoint of your docker cluster (e.g. *tcp://1.2.3.4:2375* or *unix://var/run/docker.sock*).

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:docker_ping_receiver_node':
  -
        resource:
            type: docker
            ␣
↪endpoint: replace_with_your_docker_endpoint
            origin: https://s3.lpds.
↪sztaki.hu/docker/busybox_helloworld.tar
            image: busybox_helloworld
            tag: latest
        ...
'node_def:docker_ping_sender_node':
    -
        resource:
            type: docker
            ␣
↪endpoint: replace_with_your_docker_endpoint
```

(continues on next page)

(continued from previous page)

```
            origin: https:/
↪/s3.lpds.sztaki.hu/docker/busybox_ping.tar
            image: busybox_ping
            tag: latest
```

2. Make sure your authentication information is set correctly in your authentication file. The docker plugin in Occopus does not apply authentication, however a dummy authentication block is needed. Instructions for setting the authentication properly is described at the *authentication page*. There you can download a default authentication file containing the docker section already.

3. Load the node definition for `docker_ping_receiver_node` and `docker_ping_sender_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-docker-ping.yaml
```

5. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
ping-receiver:
  10.0.
↪0.2 (552fe5b2-23a6-4c12-a4e2-077521027832)
ping-sender:
  10.0.
↪0.3 (eabc8d2f-401b-40cf-9386-4739ecd99fbd)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
# ssh ...
# docker ps
CONTAINER ID ␣
↪        IMAGE                       COMMAND ␣
↪                   CREATED              STATUS␣
↪              PORTS                NAMES
```

(continues on next page)

(continued from previous page)

```
4e83c45e8378        busybox_
↪ping:latest         "sh -c /root/start.
↪sh"   16 seconds ago      Up 15 seconds␣
↪                          romantic_brown
10b27bc4d978        busybox_
↪helloworld:latest   "sh -c /root/start.
↪sh"   17 seconds ago      Up 16 seconds␣
↪                          jovial_mayer

# docker exec␣
↪-it 4e83c45e8378 cat /root/ping-result.txt
PING 172.17.0.2 (172.17.0.2): 56 data bytes
64 bytes␣
↪from 172.17.0.2: seq=0 ttl=64 time=0.195 ms
64 bytes␣
↪from 172.17.0.2: seq=1 ttl=64 time=0.105 ms
64 bytes␣
↪from 172.17.0.2: seq=2 ttl=64 time=0.124 ms
64 bytes␣
↪from 172.17.0.2: seq=3 ttl=64 time=0.095 ms
64 bytes␣
↪from 172.17.0.2: seq=4 ttl=64 time=0.085 ms

--- 172.17.0.2 ping statistics ---
5 packets transmitted,
↪ 5 packets received, 0% packet loss
round-trip min/avg/max = 0.085/0.120/0.195 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.11 CloudSigma-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

- creating a node with basic contextualisation
- using the cloudsigma resource handler

Prerequisites

- accessing a cloud through CloudSigma interface (email, password, endpoint)
- target cloud contains a base OS image with cloud-init support (library drive identifier)

Download

You can download the example as tutorial.examples.cloudsigma-helloworld .

Steps

1. Edit nodes/node_definitions.yaml. For cloudsigma_helloworld_node set the followings in its resource section:

- endpoint is an url of a CloudSigma interface of a cloud (e.g. *https://zrh.cloudsigma.com/api/2.0*).
- libdrive_id is the image id (e.g. *40aa6ce2-5198-4e6b-b569-1e5e9fbaf488*) on your CloudSigma cloud. Select an image containing a base os installation with cloud-init support!
- cpu is the speed of CPU (e.g. *2000*) in terms of MHz of your VM to be instantiated.
- mem is the amount of RAM (e.g. *1073741824*) in terms of bytes to be allocated for your VM.
- vnc_password set the password for your VNC session.
- pubkeys optionally specifies the keypairs (e.g. *f80c3ffb-3ab5-461e-ad13-4b253da122bd*) to be assigned to your VM.
- firewall_policy optionally specifies network policies (you can define multiple security groups in the form of a list, e.g. *8cd00652-c5c8-4af0-bdd6-0e5204c66dc5*) of your VM.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```yaml
'node_def:cloudsigma_helloworld_node':
    -
      resource:
          type: cloudsigma
          endpoint: replace_with_endpoint_
↪of_cloudsigma_interface_of_your_cloud
          libdrive_id: replace_with_id_
↪of_your_library_drive_on_your_target_cloud
          description:
              cpu: 2000
              mem: 1073741824
              vnc_password: secret
              pubkeys:
                  -
                      replace_
↪with_id_of_your_pubkey_on_your_target_cloud
              nics:
                  -
```

(continues on next page)

(continued from previous page)

```
                    ␣
↪           firewall_policy: replace_with_id_
↪of_your_network_policy_on_your_target_cloud
                        ip_v4_conf:
                            conf: dhcp
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `cloudsigma_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-
↪build infra-cloudsigma-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.12 CloudSigma-Ping

This tutorial builds an infrastructure containing two nodes. The ping-sender node will ping the ping-receiver node. The sender node will store the outcome of ping in `/tmp` directory.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)
- querying a node's ip address and passing the address to another
- using the cloudsigma resource handler

Prerequisites

- accessing a cloud through CloudSigma interface (email, password, endpoint)
- target cloud contains a base OS image with cloud-init support (library drive identifier)

Download

You can download the example as tutorial.examples.cloudsigma-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `cloudsigma_ping_receiver_node` and for `cloudsigma_ping_sender_node` set the followings in their `resource` section:

- `endpoint` is an url of a CloudSigma interface of a cloud (e.g. *https://zrh.cloudsigma.com/api/2.0*).
- `libdrive_id` is the image id (e.g. *40aa6ce2-5198-4e6b-b569-1e5e9fbaf488*) on your CloudSigma cloud. Select an image containing a base os installation with cloud-init support!
- `cpu` is the speed of CPU (e.g. *2000* for 2GHz) in terms of MHz of your VM to be instantiated.
- `mem` is the amount of RAM (e.g. *1073741824*) in terms of bytes to be allocated for your VM.
- `vnc_password` set the password for your VNC session.
- `pubkeys` optionally specifies the keypairs (e.g. *f80c3ffb-3ab5-461e-ad13-4b253da122bd*) to be assigned to your VM.
- `firewall_policy` optionally specifies network policies (you can define multiple security groups in the form of a list, e.g. *8cd00652-c5c8-4af0-bdd6-0e5204c66dc5*) of your VM.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed expla-

nation can be found at the *node definition's resource section*
of the User Guide.

```
'node_def:cloudsigma_ping_receiver_node':
    -
        resource:
            name: my_cloudsigma_cloud
            type: cloudsigma
            endpoint: replace_with_endpoint_
↪of_cloudsigma_interface_of_your_cloud
            libdrive_id: replace_with_id_
↪of_your_library_drive_on_your_target_cloud
            description:
                cpu: 2000
                mem: 1073741824
                vnc_password: secret
                pubkeys:
                    -
                        replace_
↪with_id_of_your_pubkey_on_your_target_cloud
                nics:
                    -
                                                    ␣
↪            firewall_policy: replace_with_id_
↪of_your_network_policy_on_your_target_cloud
                        ip_v4_conf:
                            conf: dhcp
                            ip: null
                        runtime:
                                                                    ␣
↪                       interface_type: public
        ...
'node_def:cloudsigma_ping_sender_node':
    -
        resource:
            name: my_cloudsigma_cloud
            type: cloudsigma
            endpoint: replace_with_endpoint_
↪of_cloudsigma_interface_of_your_cloud
            libdrive_id: replace_with_id_
↪of_your_library_drive_on_your_target_cloud
            description:
                cpu: 2000
                mem: 1073741824
                vnc_password: secret
                pubkeys:
                    -
                        replace_
↪with_id_of_your_pubkey_on_your_target_cloud
                nics:
                    -
                                                    ␣
↪            firewall_policy: replace_with_id_
↪of_your_network_policy_on_your_target_cloud
```

(continues on next page)

(continued from previous page)

```
                ip_v4_conf:
                    conf: dhcp
                    ip: null
                runtime:
    ␣
↪               interface_type: public
    ...
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

3. Load the node definition for `cloudsigma_ping_receiver_node` and `cloudsigma_ping_sender_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-cloudsigma-ping.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of ip addresses:
ping-receiver:
    192.168.xxx.
↪xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
ping-sender:
    192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)

30f566d1-9945-42be-b603-795d604b362f
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/message.txt
Hello World!
↪ I am the sender node created by Occopus.
```

(continues on next page)

(continued from previous page)

```
# cat /tmp/ping-result.txt
PING 192.168.xxx.
↪xxx (192.168.xxx.xxx) 56(84) bytes of data.
64 bytes from 192.
↪168.xxx.xxx: icmp_seq=1 ttl=64 time=2.74 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=2 ttl=64 time=0.793 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=3 ttl=64 time=0.865 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=4 ttl=64 time=0.882 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=5 ttl=64 time=0.786 ms

--- 192.168.xxx.xxx ping statistics ---
5 packets transmitted,
↪ 5 received, 0% packet loss, time 4003ms
rtt min/
↪avg/max/mdev = 0.786/1.215/2.749/0.767 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.9.13 CloudBroker-Helloworld

This tutorial builds an infrastructure containing a single node. The node will receive information (i.e. a message string) through contextualisation. The node will store this information in `/tmp` directory.

Features

- creating a node with basic contextualisation
- using the cloudbroker resource handler

Prerequisites

- accessing a CloudBroker Platform instance (URL, email and password)
- Deployment, Instance type properly registered on the CloudBroker platform

Download

You can download the example as tutorial.examples.cloudbroker-helloworld .

Steps

1. Edit `nodes/node_definitions.yaml`. For `cloudbroker_helloworld_node` set the followings in its `resource` section:

- `endpoint` is the url of the CloudBroker REST API interface (e.g. *https://cola-prototype.cloudbroker.com*).
- `deployment_id` is the id of a preregistered deployment in CloudBroker referring to a cloud, image, region, etc. Make sure the image contains a base os (preferably Ubuntu) installation with cloud-init support! The id is the UUID of the deployment which can be seen in the address bar of your browser when inspecting the details of the deployment.
- `instance_type_id` is the id of a preregistered instance type in CloudBroker referring to the capacity of the virtual machine to be deployed. The id is the UUID of the instance type which can be seen in the address bar of your browser when inspecting the details of the instance type.
- `key_pair_id` is the id of a preregistered ssh public key in CloudBroker which will be deployed on the virtual machine. The id is the UUID of the key pair which can be seen in the address bar of your browser when inspecting the details of the key pair.
- `opened_port` is one or more ports to be opened to the world. This is a string containing numbers separated by comma.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
...
resource:
  type: cloudbroker
  endpoint: replace_
↪with_endpoint_of_cloudbroker_interface
  description:
    deployment_id: replace_with_deployment_id
    instance_
↪type_id: replace_with_instance_type_id
    key_pair_id: replace_with_keypair_id
    opened_port: replace_
↪with_list_of_ports_separated_with_comma
contextualisation:
...
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your `email` and `password` in the authentication file. Setting authentication information is described *here*.
3. Load the node definition for `cloudbroker_helloworld_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is

necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-
↪build infra-cloudbroker-helloworld.yaml
```

5. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
helloworld:
  192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/helloworld.txt
Hello World! I have been created by Occopus
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`.

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.9.14 CloudBroker-Ping

This tutorial sets up an infrastructure containing two nodes on the CloudBroker Platform. The ping-sender node will ping the ping-receiver node. The node will store the outcome of ping in `/tmp` directory.

Features

- creating two nodes with dependencies (i.e. ordering of deployment)
- querying a node's ip address and passing the address to another
- using the cloudbroker resource handler

Prerequisites

- accessing a CloudBroker Platform instance (URL, username and password)
- Software, Executabe, Resource, Region and Instance type properly registered on the CloudBroker platform

Download

You can download the example as tutorial.examples.cloudbroker-ping .

Steps

1. Edit `nodes/node_definitions.yaml`. Both, for `cloudbroker_ping_receiver_node` and for `cloudbroker_ping_sender_node` set the followings in their `resource` section:

- `endpoint` is the url of the CloudBroker REST API interface (e.g. *https://cola-prototype.cloudbroker.com*).
- `deployment_id` is the id of a preregistered deployment in CloudBroker referring to a cloud, image, region, etc. Make sure the image contains a base os (preferably Ubuntu) installation with cloud-init support! The id is the UUID of the deployment which can be seen in the address bar of your browser when inspecting the details of the deployment.
- `instance_type_id` is the id of a preregistered instance type in CloudBroker referring to the capacity of the virtual machine to be deployed. The id is the UUID of the instance type which can be seen in the address bar of your browser when inspecting the details of the instance type.
- `key_pair_id` is the id of a preregistered ssh public key in CloudBroker which will be deployed on the virtual machine. The id is the UUID of the key pair which can be seen in the address bar of your browser when inspecting the details of the key pair.
- `opened_port` is one or more ports to be opened to the world. This is a string containing numbers separated by comma.

Important: You can get help on collecting identifiers for the resources section at *this page* ! Alternatively, detailed explanation can be found at the *node definition's resource section* of the User Guide.

```
'node_def:cloudbroker_ping_receiver_node':
  -
    resource:
      type: cloudbroker
      endpoint: replace_
↪with_endpoint_of_cloudbroker_interface
      description:
        ␣
↪   deployment_id: replace_with_deployment_id
```

(continues on next page)

(continued from previous page)

```
        instance_
↪type_id: replace_with_instance_type_id
        key_pair_id: replace_with_keypair_id
        opened_port: replace_
↪with_list_of_ports_separated_with_comma
    contextualisation:
      type: cloudinit
      context_template: !yaml_import
      ␣
↪   url: file://cloud_init_ping_receiver.yaml
'node_def:cloudbroker_ping_sender_node':
  -
    resource:
      type: cloudbroker
      endpoint: replace_
↪with_endpoint_of_cloudbroker_interface
      description:
      ␣
↪   deployment_id: replace_with_deployment_id
        instance_
↪type_id: replace_with_instance_type_id
        key_pair_id: replace_with_keypair_id
        opened_port: replace_
↪with_list_of_ports_separated_with_comma
    contextualisation:
      type: cloudinit
      context_template: !yaml_import
    ␣
↪     url: file://cloud_init_ping_sender.yaml
```

2. Make sure your authentication information is set correctly in your authentication file. You must set your `email` and `password` in the authentication file. Setting authentication information is described *here*.
3. Load the node definition for `cloudbroker_ping_receiver_node` and `cloudbroker_ping_sender_node` node into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-cloudbroker-ping.yaml
```

5. After successful finish, the nodes with `ip address` and

node id are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
ping-receiver:
  192.168.xxx.
↪xxx (f639a4ad-e9cb-478d-8208-9700415b95a4)
ping-sender:
  192.168.yyy.
↪yyy (99bdeb76-2295-4be7-8f14-969ab9d222b8)
30f566d1-9945-42be-b603-795d604b362f
```

6. Check the result on your virtual machine.

```
ssh ...
# cat /tmp/message.txt
Hello World!
↪ I am the sender node created by Occopus.
# cat /tmp/ping-result.txt
PING 192.168.xxx.
↪xxx (192.168.xxx.xxx) 56(84) bytes of data.
64 bytes from 192.
↪168.xxx.xxx: icmp_seq=1 ttl=64 time=2.74 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=2 ttl=64 time=0.793 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=3 ttl=64 time=0.865 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=4 ttl=64 time=0.882 ms
64 bytes from 192.168.
↪xxx.xxx: icmp_seq=5 ttl=64 time=0.786 ms

--- 192.168.xxx.xxx ping statistics ---
5 packets transmitted,
↪ 5 received, 0% packet loss, time 4003ms
rtt min/
↪avg/max/mdev = 0.786/1.215/2.749/0.767 ms
```

7. Finally, you may destroy the infrastructure using the infrastructure id returned by occopus-build.

```
occopus-destroy␣
↪-i 30f566d1-9945-42be-b603-795d604b362f
```

2.10 Config manager plugins

In this section more advanced solutions will be shown. The examples will introduce complex infrastructures or will introduce more complex features of the Occopus tool.

Please, note that the following examples require a properly configured Occopus, therefore we suggest to continue this section if you already followed the instructions written in the *Installation* section.

2.10.1 Chef-Apache2

This tutorial uses Chef as a configuration management tool to deploy a one-node infrastructure containing an Apache2 web server.

Features

- using Chef as a configuration management tool to deploy services
- assembling the run-lists of the chef-clients on the nodes

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g. EC2, Azure, Nova, etc.)
- target cloud contains a base OS image with cloud-init support (image id, instance type)
- accessing the Chef server as user by Occopus (user name, user key)
- accessing the Chef server as client by the nodes (validator client name, validator client key)
- `apache2` community recipe (available at Chef Supermarket) and its dependencies uploaded to target Chef Server

Download

You can download the example as tutorial.examples.chef-apache2 .

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the EC2 plugin.

2. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

3. Edit `nodes/node_definitions.yaml`. Configure the `config_management`. Set the `endpoint` to the url of your Chef Server.

```
'node_def:chef_apache2_node':
    -
        resource:
            ...
        ...
        config_management:
            type: chef
            endpoint: replace_with_url_of_chef_server
            run_list:
                - recipe[apache2]
        ...
```

4. Edit the `nodes/cloud_init_chef.yaml` contextualization file. Set the following attributes:

- `server_url` is the url of your Chef Server (e.g. *"https://chef.yourorg.com:4000"*).
- `validation_name` the name of the validator client through which nodes register to your chef server.
- `validation_key` the public key belonging to the validator client.

```
Example:

validation_name: "yourorg-validator"
validation_key: |
    -----BEGIN RSA PRIVATE KEY-----
    YOUR-ORGS-VALIDATION-KEY-HERE
    -----END RSA PRIVATE KEY-----
```

Important: Make sure you do not mix the `validator client` with `user` belonging to the Chef Server.

```
...
chef:
   install_type: omnibus
   omnibus_url: "https://www.opscode.com/chef/install.sh"
   force_install: false
```

(continues on next page)

(continued from previous page)

```
    server_
↪url: "replace_with_your_chef_server_url"
    environment: {{infra_id}}
    node_name: {{node_id}}
    validation_name:␣
↪"replace_with_chef_validation_client_name"
    validation_key: |
  ␣
↪      replace_with_chef_validation_client_key
...
```

Important: Do not modify the value of "environment" and "node_name" attributes!

Note: For further explanation of the keywords, please read the cloud-init documentation!

5. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use, as well as the authentication data for the `config_management` section. Setting authentication information for both is described *here*.

Important: Do not forget to set your Chef credentials!

6. Load the node definitions into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

7. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-chef-apache2.yaml
```

8. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
apache2:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
14032858-d628-40a2-b611-71381bd463fa
```

9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.10.2 Chef-Wordpress

This tutorial uses Chef as a configuration management tool to deploy a two-node infrastructure containing a MySQL server node and a Wordpress node. The Wordpress node will connect to the MySQL database.

Features

- using Chef as a configuration management tool to deploy services
- passing variables to Chef through Occopus
- assembling the run-lists of the chef-clients on the nodes
- checking MySQL database availability on a node
- checking url availability on a node

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g. EC2, Azure, Nova, etc.)
- target cloud contains a base OS image with cloud-init support (image id, instance type)
- accessing the Chef server as user by Occopus (user name, user key)
- accessing the Chef server as client by the nodes (validator client name, validator client key)
- `wordpress` community recipe (available at Chef Supermarket) and its dependencies uploaded to target Chef Server
- `database-setup` recipe (provided in example package at Download) uploaded to target Chef server

Download

You can download the example as tutorial.examples.chef-wordpress .

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

  The downloadable package for this example contains a resource template for the EC2 plugin.

2. Edit `nodes/node_definitions.yaml`. For each node, configure the `config_management`. Set the `endpoint` to the url of your Chef Server.

```
'node_def:chef_mysql_node':
    -
        resource:
           ...
        ...
        config_management:
           type: chef
           endpoint: replace_with_url_of_chef_server
           run_list:
               - recipe[database-setup::db]
        ...
'node_def:chef_wordpress_node':
    -
        resource:
           ...
        ...
        config_management:
           type: chef
           endpoint: replace_with_url_of_chef_server
           run_list:
               - recipe[wordpress]
        ...
```

3. Edit the `nodes/cloud_init_chef.yaml` contextualization file. Set the following attributes:

- `server_url` is the url of your Chef Server (e.g. *"https://chef.yourorg.com:4000"*).
- `validation_name` the name of the validator client through which nodes register to your chef server.
- `validation_key` the public key belonging to the validator client.

```
Example:

validation_name: "yourorg-validator"
```

(continues on next page)

(continued from previous page)

```
validation_key: |
    -----BEGIN RSA PRIVATE KEY-----
    YOUR-ORGS-VALIDATION-KEY-HERE
    -----END RSA PRIVATE KEY-----
```

Important: Make sure you do not mix the `validator client` with `user` belonging to the Chef Server.

```
...
chef:
   install_type: omnibus
   omnibus_url: ␣
↪"https://www.opscode.com/chef/install.sh"
   force_install: false
   server_
↪url: "replace_with_your_chef_server_url"
   environment: {{infra_id}}
   node_name: {{node_id}}
   validation_name: ␣
↪"replace_with_chef_validation_client_name"
   validation_key: |
  ␣
↪     replace_with_chef_validation_client_key
...
```

Important: Do not modify the value of "environment" and "node_name" attributes!

Note: For further explanation of the keywords, please read the cloud-init documentation!

4. Edit `infra-chef-wordpress.yaml`. Set your desired root password, database name, username, and user password for your MySQL database in the variables section. These parameters will be applied when creating the mysql database.

```
...
variables:
   mysql_root_password: ␣
↪replace_with_database_root_password
   mysql_
↪database_name: replace_with_database_name
   mysql_dbuser_
↪username: replace_with_database_username
   mysql_dbuser_password: ␣
↪replace_with_database_user_password
```

5. Make sure your authentication information is set correctly

in your authentication file. You must set your authentication data for the `resource` you would like to use, as well as the authentication data for the `config_management` section. Setting authentication information for both is described *here*.

Important: Do not forget to set your Chef credentials!

6. Load the node definitions into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

7. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-chef-wordpress.yaml
```

8. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
mysql-server:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
wordpress:
    192.168.xxx.
↪xxx (894fe127-28c9-4c8f-8c5f-2f120c69b9c3)
14032858-d628-40a2-b611-71381bd463fa
```

9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.10.3 PuppetSolo-Wordpress

This tutorial uses Puppet as a configuration management tool in a server-free mode to deploy a two-node infrastructure containing a MySQL server node and a Wordpress node. The Wordpress node will connect to the MySQL database.

Features

- using server-free Puppet as a configuration management tool to deploy services
- defining puppet manifests and modules
- passing attributes to Puppet through Occopus
- checking MySQL database availability on a node
- checking url availability on a node

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g. EC2, Azure, Nova, etc.)
- target cloud contains a base OS image with cloud-init support (image id, instance type)
- `wordpress-init` puppet recipe (provided in example package at Download)
- `mysql-init` puppet recipe (provided in example package at Download)

Download

You can download the example as tutorial.examples.puppet-solo-wordpress .

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the EC2 plugin.

2. Edit `infra-puppet-solo-wordpress.yaml`. Set your desired root password, database name, username, and user password for your MySQL database in the variables section. These parameters will be applied when creating the mysql database and also used by wordpress node when connecting to mysql.

```
...
variables:
   mysql_root_password:␣
↪replace_with_database_root_password
   mysql_
↪database_name: replace_with_database_name
   mysql_dbuser_
↪username: replace_with_database_username
   mysql_dbuser_password:␣
↪replace_with_database_user_password
```

3. Load the node definitions into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-
↪build infra-puppet-solo-wordpress.yaml
```

5. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
mysql-server:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
wordpress:
    192.168.xxx.
↪xxx (894fe127-28c9-4c8f-8c5f-2f120c69b9c3)
14032858-d628-40a2-b611-71381bd463fa
```

6. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.11 Building clusters

2.11.1 Docker-Swarm cluster

This tutorial sets up a complete Docker infrastructure with Swarm, Docker and Consul software components. It contains a master node and predefined number of worker nodes. The worker nodes receive the ip of the master node and attach to the master node to form a cluster. Finally, the docker cluster can be used with any standard tool talking the docker protocol (on port `2375`).

Features

- creating two types of nodes through contextualisation
- passing ip address of a node to another node
- using the cloudsigma resource handler
- utilising health check against a predefined port
- using parameters to scale up worker nodes

Prerequisites

- accessing an Occopus compatible interface
- target cloud contains an Ubuntu 18.04 image with cloud-init support

Download

You can download the example as tutorial.examples.docker-swarm .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Cloudsigma plugin.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	2375	web listening port (configurable*)
TCP	2377	for cluster management & raft sync communications
TCP and UDP	7946	for “control plane” gossip discovery communication between all nodes

Note: Do not forget to open the ports which are needed for your Docker application!

3. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

4. Load the node definition for `dockerswarm_master_node` and `dockerswarm_worker_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition (file) changes!

```
occopus-import nodes/node_definitions.yaml
```

5. Update the number of worker nodes if necessary. For this, edit the `infra-docker-swarm.yaml` file and modify the `min` parameter under the `scaling` keyword. Currently, it is set to `2`.

```
- &W
    name: worker
    type: dockerswarm_worker_node
    scaling:
        min: 2
```

6. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-docker-swarm.yaml
```

Note: It may take a few minutes until the services on the master node come to live. Please, be patient!

7. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
master:
<ip-address>
↪ (dfa5f4f5-7d69-432e-87f9-a37cd6376f7a)
worker:
<ip-address>
↪ (cae40ed8-c4f3-49cd-bc73-92a8c027ff2c)
<ip-address>
↪ (8e255594-5d9a-4106-920c-62591aabd899)
77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

8. Check the result by submitting docker commands to the docker master node!
9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

2.11.2 Kubernetes cluster

Note: This Occopus-based version is now deprecated in favor of the Kubernetes Reference Architecture based on Terraform and Ansible.

This tutorial sets up a complete Kubernetes infrastructure with Kubernetes Dashboard and Helm package manager. It contains a master node and predefined number of worker nodes. The worker nodes receive the ip of the master node and attach to the master node to form a cluster. Finally, the Kubernetes cluster can be used with any standard tool talking the Kubernetes API server protocol (on port 6443).

Features

- creating two types of nodes through contextualisation
- passing ip address of a node to another node
- using the nova resource handler
- utilising health check against a predefined port
- using parameters to scale up worker nodes

Prerequisites

- accessing an Occopus compatible interface
- target cloud contains an Ubuntu 18.04 image with cloud-init support

Download

You can download the example as tutorial.examples.kubernetes .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Cloudsigma plugin.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	2379-2380	etcd server client API
TCP	6443	Kubernetes API server
TCP	10250	Kubelet API
TCP	10251	kube-scheduler
TCP	10252	kube-controller-manager
TCP	10255	read-only kubelet API
TCP	30000-32767	NodePort Services

Note: Do not forget to open the ports which are needed for your Kubernetes application!

3. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

4. Load the node definition for `kubernetes_master_node` and `kubernetes_slave_node` nodes into the database.

Note: Make sure the proper virtualenv is activated! (source occopus/bin/activate)

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is

necessary whenever the node definition (file) changes!

```
occopus-import nodes/node_definitions.yaml
```

5. Update the number of worker nodes if necessary. For this, edit the `infra-kubernetes.yaml` file and modify the `min` parameter under the `scaling` keyword. Currently, it is set to `2`.

```
- &W
    name: kubernetes-slave
    type: kubernetes_slave_node
    scaling:
        min: 2
```

6. Start deploying the infrastructure.

```
occopus-build infra-kubernetes.yaml
```

Note: It may take a few minutes until the services on the master node come to live. Please, be patient!

7. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
master:
    <ip-address>
↪ (dfa5f4f5-7d69-432e-87f9-a37cd6376f7a)
worker:
    <ip-address>
↪ (cae40ed8-c4f3-49cd-bc73-92a8c027ff2c)
    <ip-address>
↪ (8e255594-5d9a-4106-920c-62591aabd899)
77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

8. You can check the health and statistics of the cluster. Please login to the master node via SSH connection.

Note: Before you run the command below, please make sure you use the correct user (kubeuser).

Switch to kubeuser:

```
$ sudo su - kubeuser
```

Check the nodes added to the cluster with the following command:

```
$ kubectl get nodes
NAME      ␣
↪                                                       ␣
↪            STATUS   ROLES    AGE    VERSION
occopus-kubernetes-
↪cluster-a67dcbea-kubernetes-master-
↪90d7cfdd   Ready    master   12m    v1.18.3
occopus-kubernetes-
↪cluster-a67dcbea-kubernetes-slave-a8962b51␣
↪   Ready    worker   4m7s   v1.18.3
occopus-kubernetes-
↪cluster-a67dcbea-kubernetes-slave-ed210ec4␣
↪   Ready    worker   4m7s   v1.18.3
```

Ensure that Kubernetes services have been set up correctly.

```
$ kubectl get pods --all-namespaces
NAMESPACE     ␣
↪        NAME                                          ␣
↪                                                       ␣
↪            READY   STATUS    RESTARTS   AGE
kube-system   ␣
↪        coredns-66bff467f8-ltkkc                       ␣
↪                                                       ␣
↪            1/1     Running   0          12m
kube-system   ␣
↪        coredns-66bff467f8-ndh88                       ␣
↪                                                       ␣
↪            1/1     Running   0          12m
kube-system         ␣
↪  etcd-occopus-kubernetes-cluster-a67dcbea-
↪kubernetes-master-90d7cfdd             ␣
↪            1/1     Running   0          12m
kube-system          ␣
↪ kube-apiserver-occopus-kubernetes-cluster-
↪a67dcbea-kubernetes-master-90d7cfdd␣
↪            1/1     Running   0          12m
kube-system           ␣
↪kube-controller-manager-occopus-kubernetes-
↪cluster-a67dcbea-kubernetes-master-
↪90d7cfdd   1/1     Running   0          12m
kube-system     ␣
↪      kube-flannel-ds-amd64-5ptjb                     ␣
↪                                                       ␣
↪          1/1     Running   0          4m23s
kube-system   ␣
↪        kube-flannel-ds-amd64-dfczs                    ␣
↪                                                       ␣
↪            1/1     Running   0          12m
kube-system     ␣
↪      kube-flannel-ds-amd64-dqjg2                     ␣
↪                                                       ␣
↪          1/1     Running   0          4m23s
```

(continues on next page)

(continued from previous page)

```
kube-system                  kube-proxy-f8czw                                    1/1     Running   0          12m
kube-system                  kube-proxy-hlvd6                                    1/1     Running   0          4m23s
kube-system                  kube-proxy-vlwk2                                    1/1     Running   0          4m23s
kube-system                  kube-scheduler-occopus-kubernetes-cluster-a67dcbea-kubernetes-master-90d7cfdd    1/1     Running   0          12m
kube-system                  tiller-deploy-55bbcfbbc8-fj8mm                      1/1     Running   0          9m16s
kubernetes-dashboard         dashboard-metrics-scraper-6b4884c9d5-w6rx6          1/1     Running   0          12m
kubernetes-dashboard         kubernetes-dashboard-64794c64b8-sb9m6               1/1     Running   0          12m

You can access Dashboard at ``http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login``.

On the login page please click on the SKIP button.
```

9. Finally, you may destroy the infrastructure using the infrastructure id returned by occopus-build

```
occopus-destroy -i 77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

2.11.3 Slurm cluster

Slurm is an open source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained. As a cluster workload manager, Slurm has three key functions:

- First, it allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work.
- Second, it provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes.
- Finally, it arbitrates contention for resources by managing a queue of pending work.

This tutorial sets up a complete Slurm (version **19.05.5**) infrastructure. It contains a Slurm Management (master) node and Slurm Compoute (worker) nodes, which can be scaled up or down.

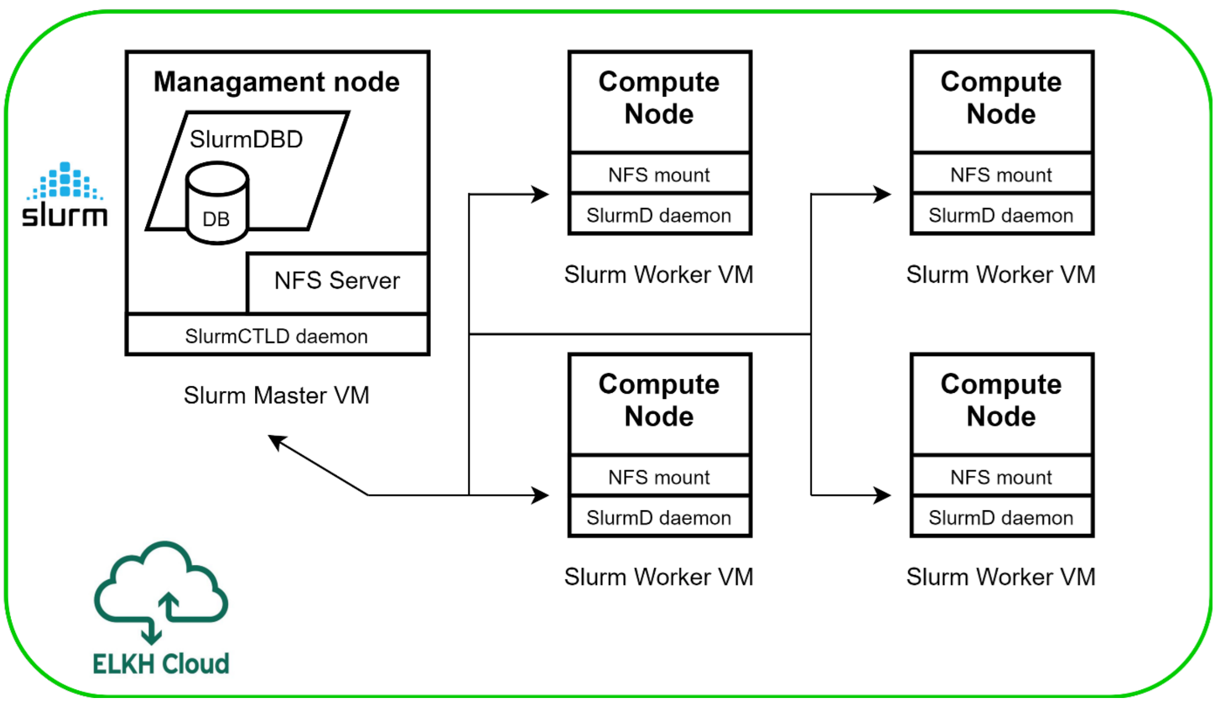


Fig. 1: Figure 2. Slurm cluster architecture

Features

- creating two types of nodes through contextualisation
- utilising health check against a predefined port
- using cron jobs to scale Slurm Compute nodes automatically

Prerequisites

- accessing an Occopus compatible interface
- target cloud contains an Ubuntu 18.04 image with cloud-init support

Download

You can download the example as tutorial.examples.slurm .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the nova plugin.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	111	RPCbind
TCP	2049	NFS Server
TCP	6817	SlurmDbDPort (Master)
TCP	6818	SlurmDPort (Worker)
TCP	6819	SlurmctldPort (Master)

Note: The Slurm Master doesn't work without any worker nodes. You can test the cluster with the sinfo command. If the Master node doesn't recognise this command, you have to wait for the first worker node.

3. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

4. Load the node definition for `slurm_master_node` and `slurm_worker_node` nodes into the database.

Note: Make sure the proper virtualenv is activated! (source occopus/bin/activate)

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition (file) changes!

```
occopus-import nodes/node_definitions.yaml
```

5. Update the number of worker nodes if necessary. For this, edit the `infra-slurm-cluster` file and modify the `min` parameter under the `scaling` keyword. Currently, it is set to `2`.

```
- &W
    name: slurm-worker
    type: slurm_worker_node
    scaling:
        min: 2
```

6. Start deploying the infrastructure.

```
occopus-build infra-slurm-cluster.yaml
```

Note: It may take a few minutes until the services on the master node come to live. Please, be patient!

7. After successful finish, the node with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
master:
    <ip-address>
↪ (dfa5f4f5-7d69-432e-87f9-a37cd6376f7a)
worker:
    <ip-address>
↪ (cae40ed8-c4f3-49cd-bc73-92a8c027ff2c)
    <ip-address>
↪ (8e255594-5d9a-4106-920c-62591aabd899)
77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

8. You can check the health and statistics of the cluster. Please login to the master node via SSH connection.

Note: Before you run the command below, please make sure at least one worker node is connected to the master.

By default, `sinfo` lists the partitions that are available.

```
sinfo
PARTITION␣
↪AVAIL TIMELIMIT NODES STATE  NODELIST
debug*       up   infinite  ␣
↪   2   idle occopus-slurm-cluster-8769d296-
↪slurm-worker-69ed479f,occopus-slurm-
↪cluster-8769d296-slurm-worker-ef6cc071
```

Please run the following command on the master node to check the status of the slurm controller daemon status.

```
sudo systemctl status slurmctld
? slurmctld.service - Slurm controller daemon
    Loaded:␣
↪loaded (/lib/systemd/system/slurmctld.
↪service; enabled; vendor preset: enabled)
    Active: active (running) since␣
↪Wed 2021-07-07 17:39:08 CEST; 1min 5s ago
    Docs: man:slurmctld(8)
    Process: 13401␣
↪ExecStart=/usr/sbin/slurmctld $SLURMCTLD_
↪OPTIONS (code=exited, status=0/SUCCESS)
Main PID: 13423 (slurmctld)
    Tasks: 11
    Memory: 2.2M
    CGroup: /system.slice/slurmctld.service
            L¦13423 /usr/sbin/slurmctld
```

You can also check the slurm daemon status on any of the worker nodes with the following command.

```
sudo systemctld status slurmd
? slurmd.service - Slurm node daemon
   ␣
↪Loaded: loaded (/lib/systemd/system/slurmd.
↪service; enabled; vendor preset: enabled)
    Active: active (running) since␣
↪Wed 2021-07-07 17:39:01 CEST; 3min 44s ago
    Docs: man:slurmd(8)
    Process:␣
↪7491 ExecStart=/usr/sbin/slurmd $SLURMD_
↪OPTIONS (code=exited, status=0/SUCCESS)
Main PID: 7493 (slurmd)
    Tasks: 1
    Memory: 1.6M
    CGroup: /system.slice/slurmd.service
            L¦7493 /usr/sbin/slurmd
```

9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-destroy`

```
occopus-destroy␣
↪-i 77cb026b-2f81-46a5-87c5-2adf13e1b2d3
```

User management

In the Slurm you can use the sacctmgr command for user management. First, you need to create an account. An account is similar to a UNIX group. An account may contain multiple users, or just a single user. Accounts may be organized as a hierarchical tree. A user may belong to multiple accounts, but must have a DefaultAccount.

```
# Create new account
sacctmgr add account␣
↪sztaki Description="Any departments"
# Show all accounts:
sacctmgr show account
```

Note: By default you are the root user in Slurm, so, you have to use sudo before the slurm commands if you use the ubuntu user instead of root.

Important: Before you create a Slurm user, you have to create a real unix user too!

2.11.4 DataAvenue cluster

Data Avenue is a data storage management service that enables to access different types of storage resources (including S3, sftp, GridFTP, iRODS, SRM servers) using a uniform interface. The provided REST API allows of performing all the typical storage operations such as creating folders/buckets, renaming or deleting files/folders, uploading/downloading files, or copying/moving files/folders between different storage resources, respectively, even simply using 'curl' from command line. Data Avenue automatically translates users' REST commands to the appropriate storage protocols, and manages long-running data transfers in the background.

In this tutorial we establish a cluster with two nodes types. On the DataAvenue node the DataAvenue application will run, and an S3 storage will run, in order to be able to try DataAvenue file transfer software such as making buckets, download or copy files. We used MinIO and Docker components to build-up the cluster.

Features

- creating two types of nodes through contextualisation
- using the nova resource handler

Prerequisites

- accessing an Occopus compatible interface

- target cloud contains an Ubuntu image with cloud-init support

Download

You can download the example as tutorial.examples.dataavenue-cluster .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the nova plugin.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	80	HTTP
TCP	443	HTTPS
TCP	8080	DA service

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

4. Optionally edit the "variables" section of the `infra-dataavenue.yaml` file. Set the following attributes:

- `access_key` is the access key of the S3 storage user
- `secret_key` is the secret key of the S3 storage user

5. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is

necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

6. Start deploying the infrastructure.

```
occopus-build infra-dataavenue.yaml
```

7. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
dataavenue:
    192.168.xxx.
↪xxx (34b07a23-a26a-4a42-a5f4-73966b8ed23f)
storage:
    192.168.xxx.
↪xxx (29b98290-c6f4-4ae7-95ca-b91a9baf2ea8)

db0f0047-f7e6-428e-a10d-3b8f7dbdb4d4
```

8. On the S3 storage nodes a user with predefined parameters will be created. The `access_key` will be the Username and the `secret_key` will be the Password, which are predefined in the `infra-dataavenue.yaml` file. Save user credentials into a file named `credentials` use the above command:

```
echo -e 'X-Key: dataavenue-key\nX-Username:␣
↪A8Q2WPCWAELW61RWDGO8\nX-Password:␣
↪FWd1mccBfnw6VHa2vod98NEQktRCYlCronxbO1aQ
↪' > credentials
```

Note: This step will be useful to shorten the curl commands later when using DataAvenue!

9. Save the nodes' ip addresses in variables to simplify the use of commands.

```
export SOURCE_NODE_IP=[storage_a_ip]
export TARGET_NODE_IP=[storage_b_ip]
export DATAAVENUE_NODE_IP=[dataavenue_ip]
```

10. Make bucket on each S3 storage node:

```
curl -H "$(cat credentials)
↪" -X POST -H "X-URI: s3://$SOURCE_NODE_
↪IP:80/sourcebucket/" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

(continues on next page)

(continued from previous page)

```
curl -H "$(cat credentials)
↪" -X POST -H "X-URI: s3://$TARGET_NODE_
↪IP:80/targetbucket/" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

Note: Bucket names should be at least three letter length. Now, the bucket on the source S3 storage node will be `sourcebucket`, and the bucket on the target S3 storage node will be `targetbucket`.

11. Check the bucket creation by listing the buckets on each storage node:

```
curl -H "$(cat credentials)" -H "X-URI: s3:/
↪/$SOURCE_NODE_IP:80/" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

The result should be: `["sourcebucket/"]`

```
curl -H "$(cat credentials)" -H "X-URI: s3:/
↪/$TARGET_NODE_IP:80/" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

The result should be: `["targetbucket/"]`

12. To test the DataAvenue file transfer software you should make a file to be transfered. With this command you can create predefined sized file, now it will be 1 megabyte:

```
dd if=/dev/urandom of=1MB.dat bs=1M count=1
```

13. Upload the generated `1MB.dat` file to the source storage node:

```
curl -H "
↪$(cat credentials)" -X POST -H "X-URI: s3:/
↪/$SOURCE_NODE_IP:80/sourcebucket/1MB.dat" -
↪H 'Content-Type: application/octet-stream'␣
↪--data-binary @1MB.dat http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/file
```

14. Check the uploaded file by listing the `sourcebucket` bucket on the source node:

```
curl -H "$(cat␣
↪credentials)" -H "X-URI: s3://$SOURCE_NODE_
↪IP:80/sourcebucket" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

The result should be: `["1MB.dat"]`

1. Save the target node’s credentials to a `target.json` file to shorten the copy command later:

```
echo "{target:
↪'s3://"$TARGET_NODE_IP":80/targetbucket/',
↪overwrite:true,credentials:{Type:UserPass,
↪ UserID:"A8Q2WPCWAELW61RWDGO8", UserPass:
↪"FWd1mccBfnw6VHa2vod98NEQktRCYlCronxbO1aQ
↪"}}" > target.json
```

2. Copy the uploaded 1MB.dat file from the source node to the target node:

```
curl -H "$(cat credentials)" ␣
↪-X POST -H "X-URI: s3://$SOURCE_NODE_IP:80/
↪sourcebucket/1MB.dat" -H "Content-type:␣
↪application/json" --data "$(cat target.
↪json)"  http://$DATAAVENUE_NODE_IP:8080/
↪dataavenue/rest/transfers > transferid
```

The result should be: `[transfer_id]`

3. Check the result of the copy command by querying the `transfer_id` returned by the copy command:

```
curl -H "$(cat credentials)
↪"  http://$DATAAVENUE_NODE_IP:8080/
↪dataavenue/rest/transfers/$(cat transferid)
```

The following result means a successful copy transfer from the source node to the target node (see status: DONE):

```
"bytesTransferred
↪":1048576,"source":"s3://[storage_
↪a_ip]:80/sourcebucket/1MB.dat","status
↪":"DONE","serverTime":1507637326644,"target
↪":"s3://[storage_b_ip]:80/targetbucket/
↪1MB.dat","ended":1507637273245,
↪"started":1507637271709,"size":1048576
```

4. You can list the files in the target node's bucket, to check the 1MB file:

```
curl -H "$(cat␣
↪credentials)" -H "X-URI: s3://$TARGET_NODE_
↪IP:80/targetbucket" http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/directory
```

The result should be: `["1MB.dat"]`. T

5. Also, you can download the copied file from the target node:

```
curl -H "$(cat credentials)" -H "X-URI:␣
↪s3://$TARGET_NODE_IP:80/targetbucket/1MB.
↪dat" -o download.dat http://$DATAAVENUE_
↪NODE_IP:8080/dataavenue/rest/file
```

6. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i db0f0047-f7e6-428e-a10d-3b8f7dbdb4d4
```

Note: In this tutorial we used HTTP protocol only. DataAvenue also supports HTTPS on port 8443; storages could also be accessed over secure HTTP by deploying e.g. HAPROXY on their nodes.

2.11.5 CQueue cluster

CQueue stands for "Container Queue". Since Docker does not provide pull model for container execution, (Docker Swarm uses push execution model) the CQueue framework provides a lightweight queueing service for executing containers.

Figure 1 shows, the overall architecture of a CQueue cluster. The CQueue cluster contains one Master node (VM1) and any number of Worker nodes (VM2). Worker nodes can be manually scaled up and down with Occopus. The Master node implements a queue (see "Q" box within VM1), where each item (called task in CQueue) represents the specification of a container execution (image, command, arguments, etc.). The Worker nodes (VM2) fetch the tasks one after the other and execute the container specified by the task (see "A" box within VM2). In each task submission a new Docker container will be launched within at CQueue Worker.

Please, note that CQueue is not aware of what happens inside the container, simply executes them one after the other. CQueue does not handle data files, containers are responsible for downloading inputs and uploading results if necessary. For each container CQueue stores the logs (see "DB" box within VM1), and the return value. CQueue retries the execution of failed containers as well.

In case the container hosts an application, CQueue can be used for executing jobs, where each job is realized by one single container execution. To use CQueue for huge number of job execution, prepare your container and generate the list of container execution in a parameter sweep style.

In this tutorial we deploy a CQueue cluster with two nodes: 1) a Master node (see VM1 on Figure 1) having a RabbitMQ (for queuing) (see "Q" box within VM1), a Redis (for storing container logs) (see "DB" within VM1), and a web-based frontend (for providing a REST API and a basic WebUI) component (see "F" in VM1); 2) a Worker node (see VM2 on Figure 1) containing a CQueue worker component (see "W" box within VM2) which pulls tasks from the Master and performs the execution of containers specified by the tasks (see "A" box in VM2).

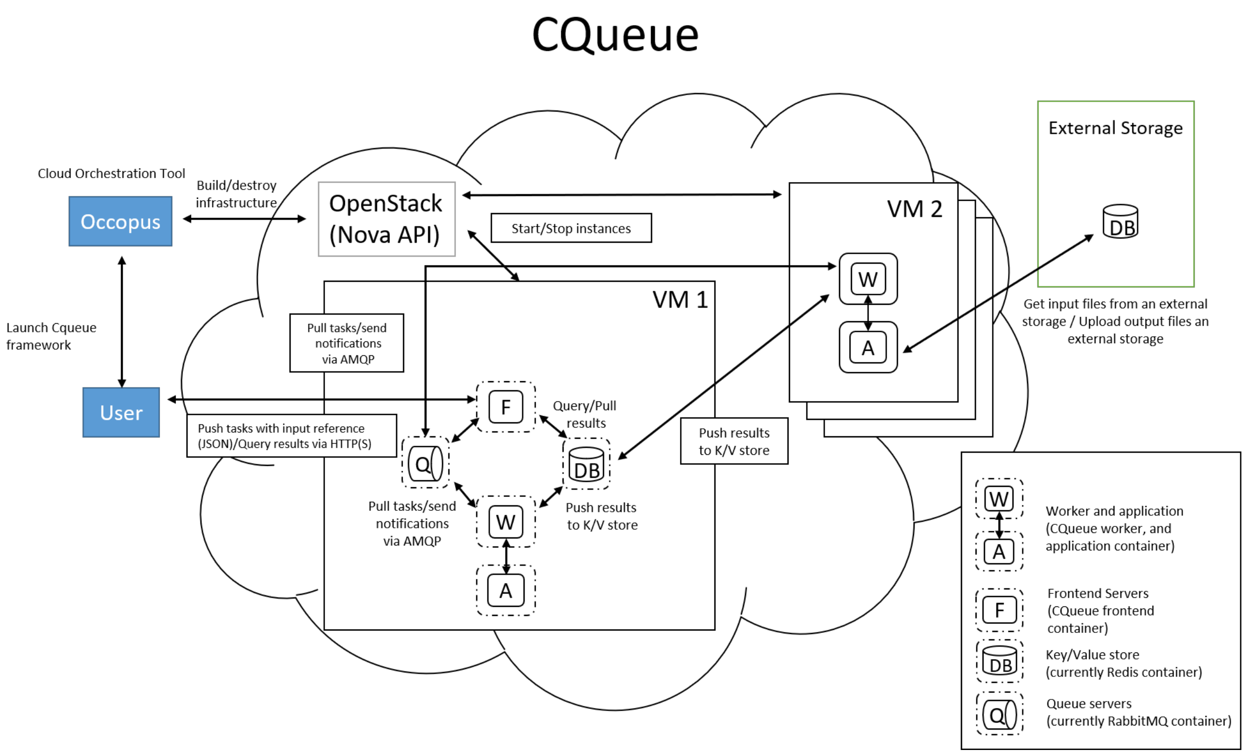


Fig. 2: Figure 1. CQueue cluster architecture

There are three use-cases identified for using CQueue.

Use-case 1 (Container executation)

The first use-case uses Container executor, i.e. the application container managed by the CQueue worker. After the application container (task) finished, the result saved on the result backend. (Redis)

```
curl -H 'Content-Type: application/json'␣
↪-X POST -d'{"image":"ubuntu", "cmd":["echo
↪", "test msg"]}' http://localhost:8080/task
```

Use-case 2 (Local executation)

The second use-case runs the task in the worker container. The container runs the given task, and after the execution, the worker container saves the result to the result backend.

```
curl -H 'Content-Type: application/json
↪' -X POST -d'{"type":"local", "cmd":["echo
↪", "test msg"]}' http://localhost:8080/task
```

Note: If you like to use this method, it is necessary to build the CQueue worker in the application container.

Use-case 3 (Batch executation)

In this use-case, the application runs in the worker container similarly to the second use-case, but it will define multiple tasks. In this mode, CQueue is capable of creating an iterable parameter in the application with the syntax of `{{.}}`. In this mode, it is necessary to define the start, and the stop parameter and CQueue will iterate over it. This execution mode can result in a very significant performance improvement when the tasks running times are short.

```
curl -H 'Content-Type: application/
↪json' -X POST -d'{"type":"batch", "start
↪":"1" , "stop":"10", "cmd":["echo", "run␣
↪{{.}}.cfg"]}' http://localhost:8080/task
```

Note: If you like to use this method, it is necessary to build the CQueue worker in the application container.

Note: To create a worker with batch capabilities, the worker must be started with `--batch=true` flag.

Features

- creating two types of nodes through contextualisation
- using the nova resource handler
- using parameters to scale up worker nodes

Prerequisites

- accessing an Occopus compatible interface
- target cloud contains an Ubuntu image with cloud-init support

Download

You can download the example as tutorial.examples.cqueue-cluster .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section).

However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	5672	AMQP
TCP	6379	Redis server
TCP	8080	CQueue frontend
TCP	15672	RabbitMQ management

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

4. Update the number of worker nodes if necessary. For this, edit the `infra-cqueue-cluster.yaml` file and modify the min and max parameter under the scaling keyword. Scaling is the interval, in which the number of nodes can change (min, max). Currently, the minimum is set to 1 (which will be the initial number at startup).

```
- &W
name: cqueue-worker
type: cqueue-worker_node
    scaling:
        min: 1
```

Important: Important: Keep in mind that Occopus has to start at least one node from each node type to work properly and scaling can be applied only for worker nodes in this example!

5. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

6. Start deploying the infrastructure.

```
occopus-build infra-cqueue-cluster.yaml
```

7. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
cqueue-worker:
    192.168.xxx.
↪xxx (34b07a23-a26a-4a42-a5f4-73966b8ed23f)
cqueue-master:
    192.168.xxx.
↪xxx (29b98290-c6f4-4ae7-95ca-b91a9baf2ea8)

db0f0047-f7e6-428e-a10d-3b8f7dbdb4d4
```

8. After a successful built, tasks can be sent to the CQueue master. The framework is built for executing Docker containers with their specic inputs. Also, environment variables and other input parameters can be specied for each container. The CQueue master receives the tasks via a REST API and the CQueue workers pull the tasks from the CQueue master and execute them. One worker process one task at a time.

   Push 'hello world' task (available parameters: image string, env []string, cmd []string, container_name string):

```
curl -H␣
↪'Content-Type: application/json' -X POST -
↪d'{"image":"ubuntu", "cmd":["echo", "hello␣
↪Docker"]}' http://<masterip>:8080/task

The result should be: ``{"id":"task_
↪324c5ec3-56b0-4ff3-ab5c-66e5e47c30e9"}``
```

Note: This id (task_324c5ec3-56b0-4ff3-ab5c-66e5e47c30e9) will be used later, in order to query its status and result.

9. The worker continuously updates the status (pending, received, started, retry, success, failure) of the task with the task's ID. After the task is completed, the workers send a notication to the CQueue master, and this task will be removed from the queue. The status of a task and the result can be queried from the key-value store through the CQueue master.

   Check the result of the push command by querying the `task_id` returned by the push command:

```
curl␣
↪-X GET http://<masterip>:8080/task/$task_id
```

The result should be: `{"status":"SUCCESS"}`

1. Fetch the result of the push command by querying the `task_id` returned by the push command:

```
curl -X GET␣
↪http://<masterip>:8080/task/$task_id/result
```

The result should be: `hello Docker`

2. Delete the task with the following command:

```
curl -X␣
↪DELETE http://<masterip>:8080/task/$task_id
```

3. For debugging, check the logs of the container at the CQueue worker node.

```
docker logs -f $(containerID)
```

4. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i db0f0047-f7e6-428e-a10d-3b8f7dbdb4d4
```

Note: The CQueue master and the worker components are written in golang, and they have a shared code-base. The open-source code is available at GitLab .

2.12 Autoscaling infrastructures

2.12.1 Autoscaling-DataAvenue

This tutorial aims to demonstrate the scaling capabilities of Occopus. With this solution applications can automatically scale without user intervention in a predefined scaling range to guarantee that the application always runs at the optimum level of resources.

The tutorial builds a scalable architecture framework with the help of Occopus and performs the automatic scaling of the application based on Occopus and Prometheus (a monitoring tool). The scalable architecture framework can be seen in Figure 1.

The scalable architecture framework consists of the following services:

1. Cloud orchestrator and manager: Occopus

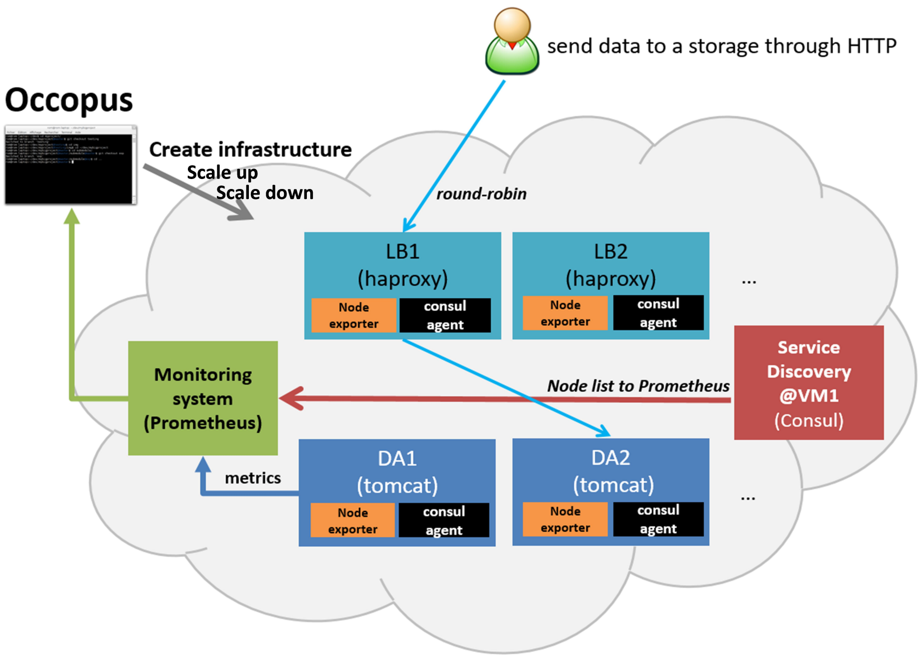


Fig. 3: Figure 1. Scalable architecture framework

2. Application node: Data Avanue (DA)
3. Service discovery: Consul
4. Load balancer: haproxy
5. Monitor: Prometheus

   In this infrastructure, nodes are discovered by Consul, which is a service discovery tool also providing DNS service and are monitored by Prometheus, a monitoring software. Prometheus supports alert definitions which later will help you write custom scaling events.

   In this autoscaling example we implemented a multi-layered traditional load-balancing schema. On the upper layer, there are load balancer nodes organised into a cluster and prepared for scaling. The load balancer cluster is handling the load generated by secured http transfer (https) between the client and the underlying application. The application is also organised inside a scalable cluster to distribute the load generated by serving the client requests. In this demonstration architecture, the Data Avenue (DA) service was selected to be the concrete application. Notice that other applications can easily replace the DA service and by changing the concrete application the scalable architecture template can support a large set of different applications. The DA service here implements data transfer between the client and a remote storage using various protocols (http, sftp, s3, …). For further details about the software, please visit the Data Avenue website . Finally, in the lowest layer there is a Database node (not shown in Figure 1) required by the instances of Data

Avenue to store and retrieve information (authentication, statistics) for their operation.

The monitor service Prometheus collects runtime information about the work of the DA services. If DA services are overloaded Prometheus instructs Occopus to scale up the number of DA services by deploying a new DA service in the cloud. The new DA service will be attached and configured the same way as it was done for the previously deployed DA services. If the DA services are underloaded Prometheus instructs Occopus to scale down the number of DA services. (In fact, the same scale up and down operations can be applied for the load balancer services, too.)

In case, this architecture fits to your need, you may replace the Data Avenue (with its Database node) with your own application. As a result, you will have a multi-level load-balancing infrastructure, where both load balancer nodes and application servers are dynamically scaled up and down depending on the load the corresponding cluster has.

Features

- using Prometheus to monitor nodes and create user-defined scaling events
- using load balancers to share system load between data nodes
- using Consul as a DNS service discovery agent
- using data nodes running the application

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g. EC2, Azure, Nova, etc.)
- target cloud contains a base 14.04 ubuntu OS image with cloud-init support (image id, instance type)
- start Occopus in Rest-API mode (occopus-rest-service)

Download

You can download the example as tutorial.examples.autoscaling-dataavenue .

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the EC2 plugin.

2. Optionally, edit the `infra_as_dataavenue.yaml` infrastructure descriptor file. Set the following attributes:

- `scaling` is the interval in which the number of nodes can change (min,max). You can change da and lb nodes or leave them as they are.

```
- &DA_cluster # Node Running your application
  name: da
  type: da
  scaling:
    min: 1
    max: 10
```

Important: Keep in mind that Occopus has to start at least one node from each node type to work properly!

3. Optionally, you can edit the `nodes/cloud_init_da.yaml` node descriptor file. If you wish, you can replace the actually implemented Grid Data Avenue webapplication with your own one. Be careful, when modifying this example!

   This autoscaling project scales the infrastructure over your application while you can run any application on it. You have to put your application code into the cloud_init_da.yaml file and make sure it starts automatically when the node boots up. This way every data node will run your application and load balancers will share the load between them. This solution fits to web applications serving high number of incoming http requests.

Note: For detailed explanation on cloud-init and its usage, please read the cloud-init documentation!

4. Optionally, edit the `nodes/cloud_init_prometheus.yaml` node descriptor file's "Prometheus rules" section in case you want to implement new scaling rules. The actually implemented rules are working well and can be seen below.

- `{infra_id}` is a built in Occopus variable and every alert has to implement it in their Labels!
- `node` should be set to da or lb depending on which type of node the alerts should work.

```
  lb_cpu_utilization␣
↪= 100 - (avg (rate(node_cpu{group=
↪"lb_cluster",mode="idle"}[60s])) * 100)
  da_cpu_utilization␣
↪= 100 - (avg (rate(node_cpu{group=
↪"da_cluster",mode="idle"}[60s])) * 100)

ALERT da_overloaded
```

(continues on next page)

(continued from previous page)

```
  IF da_cpu_utilization > 50
  FOR 1m
  LABELS␣
↪{alert="overloaded", cluster="da_cluster
↪", node="da", infra_id="{{infra_id}}"}
  ANNOTATIONS {
  summary = "DA cluster overloaded",
  description = "DA cluster average␣
↪CPU/RAM/HDD utilization is overloaded"}
ALERT da_underloaded
  IF da_cpu_utilization < 20
  FOR 2m
  LABELS␣
↪{alert="underloaded", cluster="da_cluster
↪", node="da", infra_id="{{infra_id}}"}
  ANNOTATIONS {
  summary = "DA cluster underloaded",
  description = "DA cluster average␣
↪CPU/RAM/HDD utilization is underloaded"}
```

Important: Autoscaling events (scale up, scale down) are based on Prometheus rules which act as thresholds, let's say scale up if cpu usage > 80%. In this example you can see the implementation of a cpu utilization in your da-lb cluster with some threshold values. Please, always use infra_id in you alerts as you can see below since Occopus will resolve this variable to your actual infrastructure id. If you are planning to write new alerts after you deployed your infrastructure, you can copy the same infrastructure id to the new one. Also make sure that the "node" property is set in the Labels subsection, too. For more information about Prometheus rules and alerts, please visit: https://prometheus.io/docs/alerting/rules/

5. Edit the "variables" section of the `infra_as_dataavenue.yaml` file. Set the following attributes:

- `occopus_restservice_ip` is the ip address of the host where you will start the occopus-rest-service
- `occopus_restservice_port` is the port you will bind the occopus-rest-service to

```
occopus_restservice_ip: "127.0.0.1"
occopus_restservice_port: "5000"
```

6. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8300	(Consul) TCP Server RPC. This is used by servers to handle incoming requests from other agents.
TCP and UDP	8301	(Consul) This is used to handle gossip in the LAN. Required by all agents.
TCP and UDP	8302	(Consul) This is used by servers to gossip over the WAN to other servers.
TCP	8400	(Consul) CLI RPC. This is used by all agents to handle RPC from the CLI.
TCP	8500	(Consul) HTTP API. This is used by clients to talk to the HTTP API.
TCP and UDP	8600	(Consul) DNS Interface. Used to resolve DNS queries.
TCP	8600	(Consul) DNS Interface. Used to resolve DNS queries.
TCP	9090	Prometheus
TCP	8080	Data Avenue
TCP	9093	Alertmanager

7. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

8. Load the node definitions into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

9. Start Occopus in REST service mode:

```
occopus-
↪rest-service --host [occopus_restservice_
↪ip] --port [occopus_restservice_port]
```

Use `ip` and `port` values as defined in the infrastructure description! Alternatively, use 0.0.0.0 for the host ip.

10. Start deploying the infrastructure through the Occopus service:

```
curl -X POST␣
↪http://[occopus_restservice_ip]:[occopus_
↪restservice_port]/infrastructures/
↪ --data-binary @infra_as_dataavenue.yaml
```

11. To test the down-scaling mechanism scale up manually the da nodes through the occopus REST interface and after a few minutes you can observe that the newly connected nodes will be automatically removed because the underloaded alert is

firing. You can also check the status of your alerts during the testing at [PrometheusIP]:9090/alerts.

```
curl -X POST␣
↪http://[occopus_restservice_ip]:[occopus_
↪restservice_port]/infrastructures/
↪[infrastructure_id]/scaleup/da
```

Important: Depending on the cloud you are using for you virtual machines it can take a few minutes to start a new node and connect it to your infrastructure. The connected nodes are present on prometheus's Targets page.

12. To test the up-scaling mechanism put some load on the data nodes with the command below. Just select one of your LB node and generate load on it with running the command below in a few copy. After a few minutes the cluster will be overloaded, the overloaded alerts will fire in Prometheus and a new da node will be started and connected to your cluster. Also, if you stop sending files for a while, the overloaded alerts will fire in Prometheus and one (or more) of the da nodes will be shut (scaled) down.

    To query the nodes and their ip addresses, use this command:

    ```
    curl -X GET http://[occopus_
    ↪restservice_ip]:[occopus_restservice_
    ↪port]/infrastructures/[infrastructure_id]
    ```

    Once, you have the ip of the selected LB node, generate load on it by transferring a 1GB file using the command below. Do not forget to update the placeholder!

    ```
    curl -k -o /dev/null -H "X-
    ↪Key: 1a7e159a-ffd8-49c8-8b40-549870c70e73"␣
    ↪-H "X-URI:https://autoscale.s3.lpds.sztaki.
    ↪hu/files_for_autoscale/1GB.dat" http:/
    ↪/[LB node ip address]/blacktop3/rest/file
    ```

    To check the status of alerts under Prometheus during the testing, keep watching the following url in your browser:

    ```
    http://[prometheus node ip]:9090/alerts
    ```

Important: Depending on the cloud you are using for you virtual machines it can take a few minutes to start a new node and connect it to your infrastructure. The connected nodes are present on prometheus's Targets page.

13. Finally, you may destroy the infrastructure using the infrastructure id.

```
curl -X DELETE http://[occopus_
↪restservice_ip]:[occopus_restservice_
↪port]/infrastructures/[infra id]
```

2.12.2 Autoscaling-Hadoop cluster

This tutorial aims to demonstrate the scaling capabilities of Occopus. With this solution applications can automatically scale without user intervention in a predefined scaling range to guarantee that the application always runs at the optimum level of resources.

The tutorial builds a scalable Apache Hadoop infrastructure with the help of Occopus and performs the automatic scaling of the application based on Occopus and Prometheus (a monitoring tool). It contains a Hadoop Master node and Hadoop Slave worker nodes, which can be scaled up or down. To register Hadoop Slave nodes Consul is used.

Features

- creating two types of nodes through contextualisation
- utilising health check against a predefined port
- using Prometheus to scale Hadoop Slaves automatically
- using Consul as a DNS service discovery agent

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g. EC2, Azure, Nova, etc.)
- target cloud contains a base 14.04 ubuntu OS image with cloud-init support (image id, instance type)
- generated ssh key-pair (or for testing purposes one is attached)
- start Occopus in Rest-API mode (occopus-rest-service)

  Download

  You can download the example as tutorial.examples.autoscaling-hadoop.

  Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*

- you may find a resource template for the plugin in the *resource plugin tutorials*

  The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attributes!

Important: Do not specify the server_name attribute for slaves so they are named automatically by Occopus to make sure node names are unique!

Note: If you want Occopus to monitor (health_check) your Hadoop Master and it is to be deployed in a different network, make sure you assign public (floating) IP to the Master node.

---

2. Optionally, edit the `nodes/cloud_init_hadoop_master.yaml` node descriptor file's "Prometheus rules" section in case you want to implement new scaling rules. The actually implemented rules are working well and can be seen below.

- `{infra_id}` is a built in Occopus variable and every alert has to implement it in their Labels!

```
hd_cpu_utilization␣
↪= 100 - (avg (rate(node_cpu{group=
↪"hd_cluster",mode="idle"}[60s])) * 100)
hd_ram_utilization = (sum(node_memory_
↪MemFree{job="hd_cluster"}) / sum(node_
↪memory_MemTotal{job="hd_cluster"})) * 100
hd_hdd_utilization␣
↪= sum(node_filesystem_free{job="hd_
↪cluster",mountpoint="/", device="rootfs"})␣
↪/ sum(node_filesystem_size{job="hd_cluster
↪",mountpoint="/", device="rootfs"}) *100

ALERT hd_overloaded
  IF hd_cpu_utilization > 80
  FOR 1m
  LABELS {alert=
↪"overloaded", cluster="hd_cluster", node=
↪"hadoop-slave", infra_id="{{infra_id}}"}
  ANNOTATIONS {
  summary = "HD cluster overloaded",
  description = "HD cluster␣
↪average CPU utilization is overloaded"}
ALERT hd_underloaded
  IF hd_cpu_utilization < 20
  FOR 2m
```

(continues on next page)

(continued from previous page)

```
  LABELS {alert=
↪"underloaded", cluster="hd_cluster", node=
↪"hadoop-slave", infra_id="{{infra_id}}"}
  ANNOTATIONS {
  summary = "HD cluster underloaded",
  description = "HD cluster␣
↪average CPU utilization is underloaded"}
```

Important: Autoscaling events (scale up, scale down) are based on Prometheus rules which act as thresholds, let's say scale up if cpu usage > 80%. In this example you can see the implementation of a cpu utilization in your Hadoop cluster with some threshold values. Please, always use infra_id in you alerts as you can see below since Occopus will resolve this variable to your actual infrastructure id. If you are planning to write new alerts after you deployed your infrastructure, you can copy the same infrastructure id to the new one. Also make sure that the "node" property is set in the Labels subsection, too. For more information about Prometheus rules and alerts, please visit: https://prometheus.io/docs/alerting/rules/

3. Edit the "variables" section of the `infra_as_hadoop.yaml` file. Set the following attributes:

- `occopus_restservice_ip` is the ip address of the host where you will start the occopus-rest-service
- `occopus_restservice_port` is the port you will bind the occopus-rest-service to

```
occopus_restservice_ip: "127.0.0.1"
occopus_restservice_port: "5000"
```

4. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8025	(Hadoop) Resource Manager
TCP	8042	(Hadoop) NodeManager
TCP	8080	…
TCP	8088	(Hadoop) Resource Manager WebUI
TCP	8300-8600	…
TCP	9000	…
TCP	9090	…
TCP	9093	…
TCP	50000-51000	…

5. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

6. Update the number of Hadoop Slave worker nodes if necessary. For this, edit the `infra-occopus-hadoop.yaml` file and modifiy the min and max parameter under the scaling keyword. Scaling is the interval in which the number of nodes can change (min, max). Currently, the minimum is set to 1 (which will be the initial number at startup), and the maximum is set to 10.

```
- &S
  name: hadoop-slave
  type: hadoop_slave_node
  scaling:
    min: 1
    max: 10
```

Important: Important: Keep in mind that Occopus has to start at least one node from each node type to work properly and scaling can be applied only for Hadoop Slave nodes in this example!

7. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

8. Start Occopus in REST service mode:

```
occopus-
↪rest-service --host [occopus_restservice_
↪ip] --port [occopus_restservice_port]
```

Use `ip` and `port` values as defined in the infrastructure description! Alternatively, use 0.0.0.0 for the host ip.

9. Start deploying the infrastructure through the Occopus service:

```
curl -X POST␣
↪http://[occopus_restservice_ip]:[occopus_
↪restservice_ip]/infrastructures/
↪ --data-binary @infra_as_hadoop.yaml
```

10. To test the down-scaling mechanism scale up manually the da nodes through the occopus REST interface and after a few minutes you can observe that the newly connected nodes will be automatically removed because the underloaded alert is firing. You can also check the status of your alerts during the testing at `[HaddopMasterIP]:9090/alerts`.

    ```
    curl -X POST
    ↪http://[occopus_restservice_ip]:[occopus_
    ↪restservice_ip]/infrastructures/
    ↪[infrastructure_id]/scaleup/hadoop-slave
    ```

    Important: Depending on the cloud you are using for you virtual machines it can take a few minutes to start a new node and connect it to your infrastructure. The connected nodes are present on prometheus's Targets page.

11. To test the up-scaling mechanism put some load on the Hadoop Slave nodes. After a few minutes the cluster will be overloaded, the overloaded alerts will fire in Prometheus and a new Hadoop Slave node will be started and connected to your cluster. Also, if you stop sending files for a while, the overloaded alerts will fire in Prometheus and one (or more) of the Hadoop Slave nodes will be shut (scaled) down.

    To query the nodes and their ip addresses, use this command:

    ```
    curl -X GET http://[occopus_
    ↪restservice_ip]:[occopus_restservice_
    ↪ip]/infrastructures/[infrastructure_id]
    ```

    Once, you have the ip of the Hadoop Master node, generate load on it by executing Hadoop MapRedcue jobs. To launch a Hadoop MapReduce job copy your input and executable files to the Hadoop Master node, and perform the submission described here . To login to the Hadoop Master node use the private key attached to the tutorial package:

    ```
    ssh -i builtin_
    ↪hadoop_private_key hduser@[HadoopMaster ip]
    ```

    To check the status of alerts under Prometheus during the testing, keep watching the following url in your browser:

    - `http://[HadoopMasterIP]:9090/alerts`

    Important: Depending on the cloud you are using for you virtual machines it can take a few minutes to start a new node and connect it to your infrastructure. The connected nodes are present on prometheus's Targets page.

12. You can check the health and statistics of the cluster through the following web pages:

- Health of nodes: `http://[HadoopMasterIP]:50070`
- Job statistics: `http://[HadoopMasterIP]:8088`

13. Finally, you may destroy the infrastructure using the infrastructure id.

```
curl -X DELETE␣
↪http://[occopus_restservice_ip]:[occopus_
↪restservice_ip]/infrastructures/[infra id]
```

2.13 Flowbster

2.13.1 Autodock vina

In this case we have used Flowbster to set up the infrastructure for processing the Vina workflow. The setup is as follows: one VM is acting as the Generator, 5 VMs are acting as Vina processing nodes, and finally one VM is acting as the Collector node.

The application used to execute the performance measurements was a workflow based on the AutoDock Vina application. The workflow consists of three nodes: a Generator, a set of Vina processing nodes, and a Collector. The input of the workflow includes the followings: a receptor molecule, a Vina configuration file, and a set of molecules to dock against the receptor molecule.

The task of the generator node is to split the set of molecules to dock into a number of parts. The task of the Vina nodes is to process this parts, iterating through each molecule in the given part, by performing the docking simulation. The result of the docking includes an energy level, finally the user is interested in the docking with the lowest energy level.

The task of the Collector node is to get the processing result of each molecule part from the Vina nodes, and select the best 5 energy levels.

For running the experiment, we selected a molecule set of 60 molecules. This set was split into 10 parts, so each part included 6 molecules to dock against the receptor molecule.

Features

- creating nodes through contextualisation
- using the ec2 resource handler
- utilising health check against a predefined port and url
- using parameters to scale up worker nodes

Prerequisites

- accessing an Occopus compatible interface

- target cloud contains an Ubuntu 14.04 image with cloud-init support

Download

You can download the example as tutorial.examples.flowbster-autodock-vina .

Steps

The following steps are suggested to be performed:

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the flowbster_node labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the ec2 plugin.

2. Make sure your authentication information is set correctly in your authentication file. You must set your email and password in the authentication file. Setting authentication information is described *here*.

3. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc.). Make sure you implement port opening in your cloud for the following port:

Protocol	Port(s)	Service
TCP	5000	This is used by nodes to handle incoming requests from other agents

4. Please note that in order to receive the results, you have to run a Gather service (part of Flowbster), which will finally gather the results (the docking simulations with the lowest energy levels) from the Collector (last node in the workflow). Start the Gather service using the following command:

```
scripts/flowbster-gather.sh -s
```

By default the Gather service is listening on port 5001.

Note: The scripts in the scripts directory need Python 2.7. Alternatively you can activate the Occopus virtualenv!

5. Edit the "variables" section of the infra-autodock-vina.yaml file. Set the following attributes:

- `gather_ip` is the ip address of the host where you have started the Gather service
- `gather_port` is the port of the Gather service is listening on

```
gather_ip: &gatherip "<External IP␣
↪of the host executing the Gather service>"
gather_port: &gatherport "5001"
```

6. Update the number of VINA nodes if necessary. For this, edit the `infra-autodock-vina.yaml` file and modify the `min` parameter under the `scaling` keyword. Currently, it is set to 5.

```
- &VINA
    name: VINA
    type: flowbster_node
    scaling:
            min: 5
```

7. Load the node definition for `flowbster_node` nodes into the database.

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition (file) changes!

```
occopus-import nodes/node_definitions.yaml
```

8. Start deploying the infrastructure. Make sure the proper virtualenv is activated!

```
occopus-build infra-autodock-vina.yaml
```

9. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
VINA:
  <ip-address>
↪ (2f7d3d7e-c90c-4f33-831d-91e987e8e8b2)
  <ip-address>
↪ (49bed8d2-94b0-4a7e-9672-744921dacac0)
  <ip-address>
↪ (10664026-0b31-4848-9f7a-98f880f98be7)
  <ip-address>
↪ (a0f5d091-aecc-488c-94f2-34e546f87832)
```

(continues on next page)

(continued from previous page)

```
  <ip-address>
↪ (285d7efd-84a7-4ed5-a6fa-73db47bc2e87)
COLLECTOR:
  <ip-address>
↪ (4ca11ad3-a6ec-411b-89e6-d516169df9c7)
GENERATOR:
  <ip-address>
↪ (9b8dc4f1-bed4-4d1c-ba9e-45c18ee2523d)
30bc1d09-8ed5-4b7e-9e51-24ed881fc166
```

10. Once the infrastructure is ready, the input files can be sent to the Generator node of the workflow (check the address of the node at the end of the output of the **occopus-build** command). Using the following command in the `flowbster-autodock-vina/inputs` directory:

```
../scripts/flowbster-feeder.sh -h <ip␣
↪of GENERATOR node> -i input-description-
↪for-vina.yaml -d input-ligands.zip␣
↪-d input-receptor.pdbqt -d vina-config.txt
```

The -h parameter is the Generator node's address, -i is the input description file and with -d we can define data file(s).

Note: The scripts in the scripts directory need Python 2.7. Alternatively you can activate the Occopus virtualenv!

Note: It may take a quite few minutes until the processes end. Please, be patient!

11. With step 10, the data processing was started. The whole processing time depends on the overall performance of the VINA nodes. VINA nodes process 10 molecule packages, which are collected by the Collector node. You can check the progress of processing on the Collector node by checking the number of files under `/var/flowbster/jobs/<id of workflow>/inputs` directory. When the number of files reaches 10, Collector node combines them and sends one package to Gather node which stores it under directory `/tmp/flowbster/results`.

12. Once you finished processing molecules, you may stop the Gather service:

```
scripts/flowbster-gather.sh -d
```

13. Finally, you can destroy the infrastructure using the infrastructure id returned by **occopus-build**

```
occopus-destroy␣
↪-i 30bc1d09-8ed5-4b7e-9e51-24ed881fc166
```

Note: You can run a bigger application, with more input files. This application will run for approximately 4 hours with 5 VINA nodes. Edit Generator node's variables section in the `infra-autodock-3node.yaml` file. Set the `jobflow/app/args` variable 10 to `240` and repeat the tutorial using the `input2` directory. For running this experiment, we selected a molecule set of 3840 molecules. This set will be splitted into 240 parts, so each part included 16 molecules to dock against the receptor molecule.

```
nodes:
    - &GENERATOR
        name: GENERATOR
        type: flowbster_node
        variables:
            flowbster:
                app:
                    exe:
                        filename: execute.bin
                        tgzurl:␣
↪https://github.com/occopus/flowbster/raw/
↪devel/examples/vina/bin/generator_exe.tgz
                    args: '240'
```

2.14 Big Data and AI applications

2.14.1 Apache Hadoop cluster

This tutorial sets up a complete Apache Hadoop (version **3.3.0**) infrastructure. It contains a Hadoop Master node and Hadoop Slave worker nodes, which can be scaled up or down. To register Hadoop Slave nodes Consul is used.

Features

- creating two types of nodes through contextualisation
- utilising health check against a predefined port
- using scaling parameters to limit the number of Hadoop Slave nodes
- manage cluster nodes with Consul

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains a base Ubuntu OS image with cloud-init support

Download

You can download the example as tutorial.examples.hadoop-cluster .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attributes!

Important: Do not specify the server_name attribute for slaves so they are named automatically by Occopus to make sure node names are unique!

Note: If you want Occopus to monitor (health_check) your Hadoop Master and it is to be deployed in a different network, make sure you assign public (floating) IP to the Master node.

2. Components in the infrastructure connect to each other, therefore several port ranges must be opened for the VMs executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8025	
TCP	8042	
TCP	8088	
TCP	8300-8600	
TCP	9000	
TCP	50000-51000	

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

4. Update the number of Hadoop Slave worker nodes if necessary. For this, edit the `infra-occopus-hadoop.yaml` file and modifiy the min and max parameter under the scaling keyword. Scaling is the interval in which the number of nodes can change (min, max). Currently, the minimum is set to 2 (which will be the initial number at startup), and the maximum is set to 10.

```
- &S
   name: hadoop-slave
   type: hadoop_slave_node
   scaling:
           min: 2
           max: 10
```

Important: Important: Keep in mind that Occopus has to start at least one node from each node type to work properly and scaling can be applied only for Hadoop Slave nodes in this example!

5. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

6. Start deploying the infrastructure.

```
occopus-build infra-hadoop-cluster.yaml
```

7. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
hadoop-master:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
hadoop-slave:
```

(continues on next page)

(continued from previous page)

```
    192.168.xxx.
↪xxx (23f13bd1-25e7-30a1-c1b4-39c3da15a456)
    192.168.xxx.
↪xxx (7b387348-b3a3-5556-83c3-26c43d498f39)

14032858-d628-40a2-b611-71381bd463fa
```

8. You can check the health and statistics of the cluster through the following web pages:

- Health of nodes: `http://[HadoopMasterIP]:9870`
- Job statistics: `http://[HadoopMasterIP]:8088`

9. To launch a Hadoop MapReduce job copy your input and executable files to the Hadoop Master node, and perform the submission described here.

10. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.14.2 Apache Spark cluster with RStudio Stack

This tutorial sets up a complete Apache Spark (version **3.0.1**) infrastructure with HDFS (Hadoop Distributed File System) (version **3.3.0**) and RStudio server. Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. For more information visit the official Apache Spark page .

Apache Spark cluster together with HDFS (Hadoop Distributed File System) represents one of the most important tool for Big Data and machine learning applications, enabling the parallel processing of large data sets on many virtual machines, which are running Spark workers. On the other hand, setting up a Spark cluster with HDFS on clouds is not straightforward, requiring deep knowledge of both cloud and Apache Spark architecture. To save this hard work for scientists we have created and made public the required infrastructure descriptors by which Occopus can automatically deploy Spark clusters with the number of workers specified by the user. One of the most typical application area of Big Data technology is the statistical data processing that is usually done by the programming language R. In order to facilitate the work of statisticians using Spark on cloud, we have created an extended version of the Spark infrastructure descriptors placing the sparklyr library on Spark workers, too. Finally, we have also integrated the user-friendly RStudio user interface into the Spark system. As a result, researchers using the statistical R package can easily and quickly deploy a complete R-oriented Spark cluster on clouds containing the following components: RStudio, R, sparklyr, Spark and HDFS.

This tutorial sets up a complete Apache Spark infrastructure integrated with HDFS, R, RStudio and sparklyr. It contains a Spark Master node and Spark Worker nodes, which can be scaled up or down.

Features

- creating two types of nodes through contextualisation
- utilising health check against a predefined port
- using scaling parameters to limit the number of Spark Worker nodes

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains a base Ubuntu OS image with cloud-init support

Download

You can download the example as tutorial.examples.spark-cluster-with-r .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an Occopus compatible resource plugin
- you can find and specify the relevant list of attributes for the plugin
- you may follow the help on collecting the values of the attributes for the plugin
- you may find a resource template for the plugin in the resource plugin tutorials

The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attributes!

Important: Do not specify the server_name attribute for workers so they are named automatically by Occopus to make sure node names are unique!

Note: If you want Occopus to monitor (health_check) your Spark Master and it is to be deployed in a different network, make sure you assign public (floating) IP to the Master node.

2. Generally speaking, a Spark cluster and its services are not deployed on the public internet. They are generally private services, and should only be accessible within the network of the organization that deploys Spark. Access to the hosts and ports used by Spark services should be limited to origin hosts that need to access the services. This means that you need to create a firewall rule to allow **all traffic between Spark nodes** and the **required ports** [web UI and job submission port(s)] should be allowed **only from your IP address**.

   Main UI port list:

Port	Description
4040	Application port (active only if a Spark application is running)
6066	Submit job to cluster via REST API
7077	Submit job to cluster/Join to the cluster
8080	Master UI
8081	Worker UI
9870	HDFS NameNode UI

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.
4. Update the number of Spark Worker nodes if necessary. For this, edit the `infra-occopus-spark.yaml` file and modifiy the min and max parameter under the scaling keyword. Scaling is the interval in which the number of nodes can change (min, max). Currently, the minimum is set to 2 (which will be the initial number at startup), and the maximum is set to 10.

```
- &W
   name: spark-worker
   type: spark_worker_node
   scaling:
           min: 2
           max: 10
```

Important: Important: Keep in mind that Occopus has to start at least one node from each node type to work properly and scaling can be applied only for Spark Worker nodes in this example!

5. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

6. Start deploying the infrastructure.

```
occopus-build infra-spark-cluster.yaml
```

7. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
spark-master:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
spark-worker:
    192.168.xxx.
↪xxx (23f13bd1-25e7-30a1-c1b4-39c3da15a456)
    192.168.xxx.
↪xxx (7b387348-b3a3-5556-83c3-26c43d498f39)

14032858-d628-40a2-b611-71381bd463fa
```

8. You can check the health and statistics of the cluster through the following web pages:

- HDFS NameNode UI: `http://<SparkMasterIP>:9870`
- Spark UI: `http://<SparkMasterIP>:8080`
- Spark Application UI: `http://<SparkMasterIP>:4040` (active only if a Spark application is running)

Note: The webUIs are protected, the access needs a login. The default username/password is spark/lpds, which can be changed before deployment.

9. Testing RStudio

The RStudio's web interface can be access via `http://<SparkMasterIP>:8787`, logging with the `sparkuser/lpds` username/password pair.

9.1. Testing R package

```
install.packages('txtplot')
library('txtplot')
txtplot(cars[,1], cars[,
↪2], xlab = "speed", ylab = "distance")
```

In this test, we download an R package, called "txtplot" from CRAN , load it to R and then draw an XY plot.

9.2. Testing R with Spark on local mode

```
install.packages("sparklyr")
library(sparklyr)
Sys.setenv(SPARK_
↪HOME = '/home/sparkuser/spark')
sc <- spark_connect(master = "local")
sdf_len(sc, 5, repartition = 1) %>%
spark_apply(function(e) I(e))
spark_disconnect_all()
```

In this test, we download the "sparklyr" package for Spark, load it into R, enter the path to our Spark directory, and create the Spark Context to run the code. When the Spark Context is created, our application is also displayed on the Application UI interface under Running Applications, available at http: // <SparkMasterIP>: 4040. An active Spark Context session can also be found on the interface of RStudio, in the upper right corner, under the "Connections" tab, the Spark logo appears with the configurations of Spark Context.

Note: Downloading new packages may take a few minutes.

The result of the test are numbers listed from 1 to 5. This test shows that the Spark Master ran with Spark R. The last line closes the application, otherwise Spark Context will run forever and a new application would not get new resources. (see Figure 1.)

9.3. Testing R with Spark on cluster mode

```
install.packages("sparklyr")
library(sparklyr)
Sys.setenv(SPARK_
↪HOME = '/home/sparkuser/spark')
sc <- spark_connect(master␣
↪= "spark://<SparkMasterIP>:7077")
sdf_len(sc, 5, repartition = 1) %>%
```

(continues on next page)

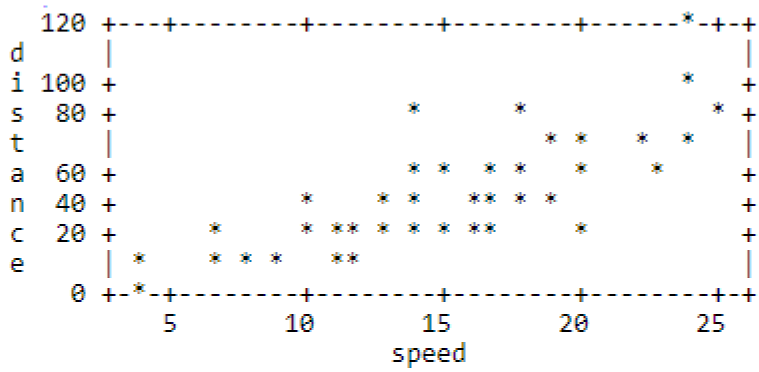


Fig. 4: Figure 1. Result of the first test

(continued from previous page)

```
spark_apply(function(e) I(e))
spark_disconnect_all()
```

The first three rows are the same as those of the second test, but we have repeated them for the sake of completeness. In this test, we download the "sparklyr" package required to use Spark, load it into R, enter the path of our Spark directory and create the Spark Context to run the code.

Note: Downloading new packages may take a few minutes.

Important: Do not forget to update placeholders.

When the Spark Context is created, the application is also displayed on the Application UI interface under Running Applications available at http: // <SparkMasterIP>: 4040.

An active Spark Context session can also be seen on the RStudio interface, in the upper right corner, under the "Connections" tab, the Spark logo appears with the configurations of Spark Context, now with the Spark Master IP address.

The test results are the same, numbers listed 1 through 5 (see Figure 1). This test shows that in the Spark cluster, the task was run in parallel, distributed along with R. The last line closes the application, otherwise Spark Context will run indefinitely, so the new application will not get new resources.

Note: For more example visit spark.rstudio.com .

10. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

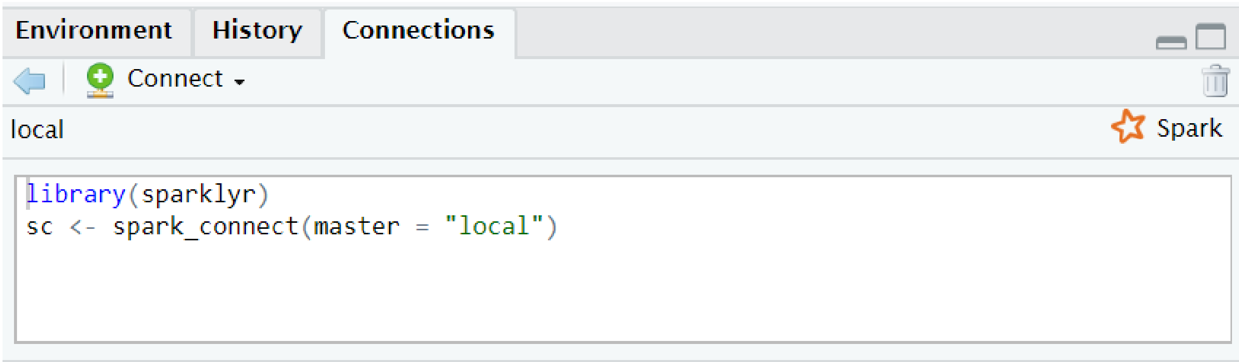


Fig. 5: Figure 2. Spark Context session on RStudio UI

```
occopus-destroy -i 14032858-d628-40a2-b611-71381bd463fa
```

2.14.3 Apache Spark cluster with Jupyter notebook and PySpark

This tutorial sets up a complete Apache Spark (version **3.0.1**) infrastructure with HDFS (Hadoop Distributed File System) (version **3.3.0**) and PySpark. Apache Spark is a fast and general-purpose cluster computing system. It provides high-level APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming. For more information visit the official Apache Spark page .

Apache Spark cluster together with HDFS (Hadoop Distributed File System) represents one of the most important tool for Big Data and machine learning applications, enabling the parallel processing of large data sets on many virtual machines, which are running Spark workers. On the other hand, setting up a Spark cluster with HDFS on clouds is not straightforward, requiring deep knowledge of both cloud and Apache Spark architecture. To save this hard work for scientists we have created and made public the required infrastructure descriptors by which Occopus can automatically deploy Spark clusters with the number of workers specified by the user. Spark also provides a special library called "Spark MLlib" for supporting machine learning applications. Similarly, to the R-oriented Spark environment, we have developed the infrastructure descriptors for the creation of a machine learning environment in the cloud. Here, the programming language is Python and the user programming environment is Jupyter. The complete machine learning environment consists of the following components: Jupyter, Python, Spark and HDFS. Deploying this machine learning environment is also automatically done by Occopus and the number of Spark workers can be defined by the user.

This tutorial sets up a complete Apache Spark infrastructure integrated with HDFS, Python and Jupyter Notebook. It contains a Spark Master node and Spark Worker nodes, which can be scaled up or down.

Features

- creating two types of nodes through contextualisation
- utilising health check against a predefined port
- using scaling parameters to limit the number of Spark Worker nodes

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains a base Ubuntu OS image with cloud-init support

Download

You can download the example as tutorial.examples.spark-cluster-with-python .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attributes!

Important: Do not specify the server_name attribute for workers so they are named automatically by Occopus to make sure node names are unique!

Note: If you want Occopus to monitor (health_check) your Spark Master and it is to be deployed in a different network, make sure you assign public (floating) IP to the Master node.

2. Generally speaking, a Spark cluster and its services are not deployed on the public internet. They are generally private services, and should only be accessible within the network of the organization that deploys Spark. Access to the hosts and ports used by Spark services should be limited to origin hosts that need to access the services.

   This means that you need to create a firewall rule to allow **all traffic between Spark nodes** and the **required ports** [web UI and job submission port(s)] should be allowed **only from your IP address**.

   Main UI port list:

Port	Description
4040	Application port (active only if a Spark application is running)
6066	Submit job to cluster via REST API
7077	Submit job to cluster/Join to the cluster
8080	Master UI
8081	Worker UI
9870	HDFS NameNode UI

1. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.
2. Update the number of Spark Worker nodes if necessary. For this, edit the `infra-occopus-spark.yaml` file and modifiy the min and max parameter under the scaling keyword. Scaling is the interval in which the number of nodes can change (min, max). Currently, the minimum is set to 2 (which will be the initial number at startup), and the maximum is set to 10.

```
- &W
   name: spark-worker
   type: spark_worker_node
   scaling:
           min: 2
           max: 10
```

Important: Important: Keep in mind that Occopus has to start at least one node from each node type to work properly and scaling can be applied only for Spark Worker nodes in this example!

3. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure.

```
occopus-build infra-spark-cluster.yaml
```

5. After successful finish, the nodes with `ip address` and `node id` are listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
spark-master:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)
spark-worker:
    192.168.xxx.
↪xxx (23f13bd1-25e7-30a1-c1b4-39c3da15a456)
    192.168.xxx.
↪xxx (7b387348-b3a3-5556-83c3-26c43d498f39)

14032858-d628-40a2-b611-71381bd463fa
```

Note: After Occopus finished the infrastructure, the Worker instance takes some time to finish the deployment process via cloud-init.

6. You can check the health and statistics of the cluster through the following web pages:

- HDFS NameNode UI: `http://<SparkMasterIP>:9870`
- Spark UI: `http://<SparkMasterIP>:8080`
- Spark Application UI: `http://<SparkMasterIP>:4040` (active only if a Spark application is running)

Note: The webUIs are protected, the access needs a login. The default username/password is spark/lpds, which can be changed before deployment.

7. Testing with Jupyter Notebook

   The Jupyter notebook's web interface can be access via `http://<SparkMasterIP>:8888`. Here, you can upload and run Jupyter notebooks and try out the prepared demo notebook.

Note: The webUIs are protected, the access needs a login. The default password is "lpds", which can be changed before deployment.

8. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.14.4 TensorFlow and Keras with Jupyter Notebook Stack

Note: This Occopus-based version is now deprecated in favor of the TensorFlow-JupyterLab Reference Architecture based on Terraform and Ansible.

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 on November 9, 2015. For more information visit the official TensorFlow page .

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras contains numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier. In addition to standard neural networks, Keras has support for convolutional and recurrent neural networks. It supports other common utility layers like dropout, batch normalization, and pooling. For more information visit the official Keras page .

The complete machine learning environment consists of the following components: Jupyter, Keras (version 2.2.4) and TensorFlow (version 1.13.1).

Features

- creating a node through contextualisation
- utilising health check against a predefined port

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains a base Ubuntu OS image with cloud-init support

Download

You can download the example as tutorial.examples.tensorflow-keras-jupyter .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attribute!

Note: If you want Occopus to monitor (health_check) your initiated virtual machine and it is to be deployed in a different network, make sure you assign public (floating) IP to the node.

2. Services on the virtual machine should be available from outside, therefore some port numbers must be opened for the VM executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8888	Jupyter Notebook

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.
4. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

5. Start deploying the infrastructure.

```
occopus-build infra-jupyter-server.yaml
```

6. After successful finish, the node with `ip address` and `node id` is listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
jupyter-server:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)

14032858-d628-40a2-b611-71381bd463fa
```

7. You can start using the TensorFlow/Keras stack through the Jupyter notebook using your web browser at the following URL:

- Jupyter notebook: `http://<JupyterServerIP>:8888`

Note: The webUIs are protected, the access needs a login. The default password is "lpds", which can be changed before deployment.

8. Run a demo ML application. Select tensorflow-demo/TensorFlowDemoWithPictures.ipynb file within the Jupyter notebook interface, and select Cells/Run All to run all of the commands below, or use shift+enter within a cell to run the cells one-by-one.
9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

(continues on next page)

(continued from previous page)

2.14.5 TensorFlow 2 with JupyterLab Stack using NVIDIA GPU card

Note: This Occopus-based version is now deprecated in favor of the TensorFlow-JupyterLab Reference Architecture based on Terraform and Ansible.

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. TensorFlow was developed by the Google Brain team for internal Google use. It was released under the Apache License 2.0 on November 9, 2015. For more information visit the official TensorFlow page .

The complete machine learning environment consists of the following components: JupyterLab and TensorFlow 2 utilizing the power of a GPU card.

Important: If you want to use this tutorial, your virtual machine should have an attached NVIDIA GPU card. If you would like to alter the CUDA driver, feel free to personalize the install-cuda.sh script within nodes/cloud_init_jupyter_server_gpu.yaml file.

Features

- creating a node through contextualisation
- utilising health check against a predefined port

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains a base Ubuntu 20.04 OS image with cloud-init support and Docker CE

Download

You can download the example as tutorial.examples.tensorflow-jupyter-gpu .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

  The downloadable package for this example contains a resource template for the Nova plugin.

Important: Do not modify the values of the contextualisation and the health_check section's attribute!

Note: Make sure you assign public (floating) IP to the node.

2. Services on the virtual machine should be available from outside, therefore some port numbers must be opened for the VM executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8888	Jupyter Notebook

3. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.
4. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

5. Start deploying the infrastructure.

```
occopus-build infra-tensorflow.yaml
```

6. After successful finish, the node with `ip address` and `node id` is listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store

the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/ip addresses:
jupyter-server-gpu:
    192.168.xxx.
↪xxx (3116eaf5-89e7-405f-ab94-9550ba1d0a7c)

14032858-d628-40a2-b611-71381bd463fa
```

7. You can start using the TensorFlow/Keras stack through the Jupyter notebook using your web browster at the following URL:

- Jupyter notebook: `` `http://<JupyterServerIP>:8888` ``

Note: The webUIs are protected, the access needs a login. The default password is "tensorflow", which can be changed before deployment.

8. Run a demo TensorFlow notebook .

   Select beginner.ipynb file (see Figure 1) within the JupyterLab interface, and select Cells/Run All to run all of the commands below, or use shift+enter within a cell to run the cells one-by-one.

9. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy␣
↪-i 14032858-d628-40a2-b611-71381bd463fa
```

2.14.6 JupyterLab

Note: This Occopus-based version is now deprecated in favor of the JupyterLab Reference Architecture based on Terraform and Ansible.

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. The notebook extends the console-based approach to interactive computing in a qualitatively new direction, providing a web-based application suitable for

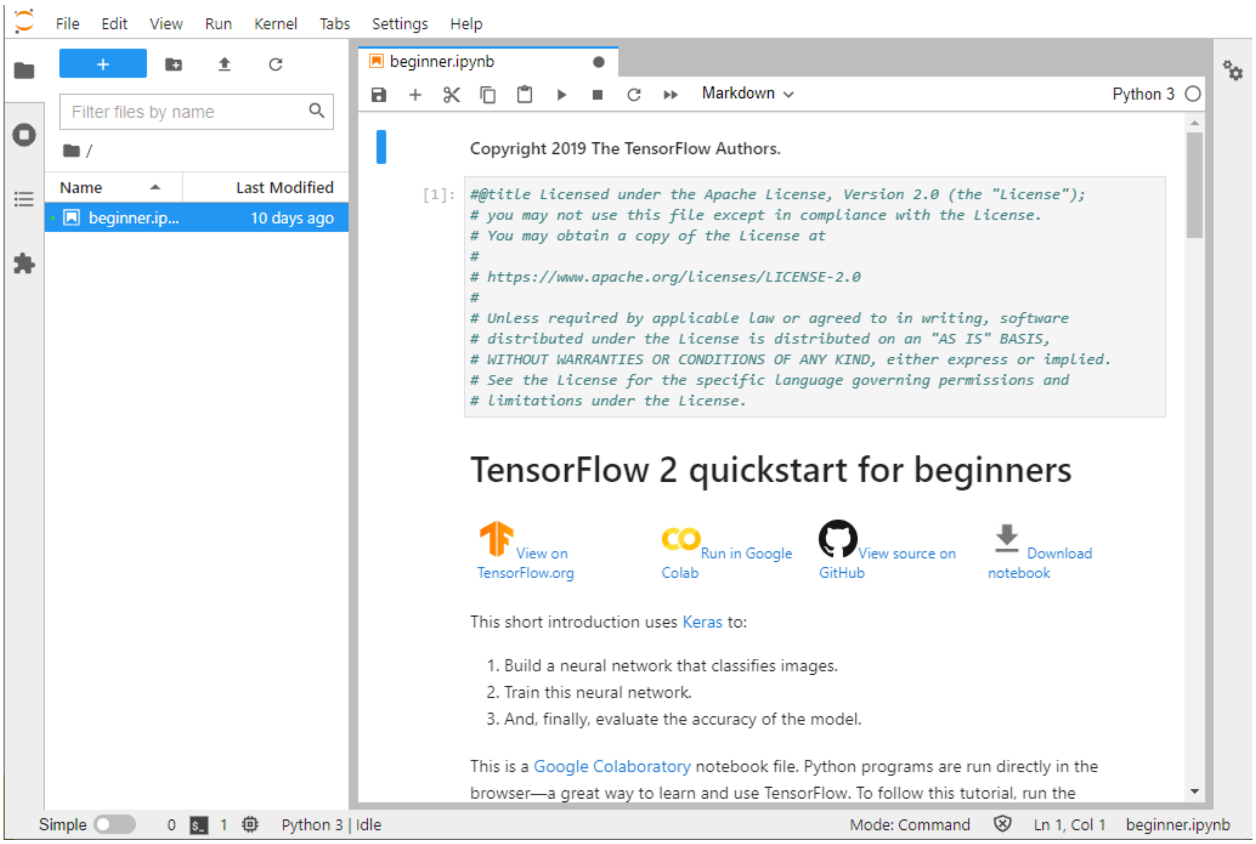


Fig. 6: Figure 1: Jupyter Notebook for testing TensorFlow 2 environment with GPU

capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

The Jupyter Notebook combines two components:

- A web application: a browser-based tool for interactive authoring of documents which combine explanatory text, mathematics, computations and their rich media output.
- Notebook documents: a representation of all content visible in the web application, including inputs and outputs of the computations, explanatory text, mathematics, images, and rich media representations of objects.

For more information on Jupyter Notebooks, visit the official documentation of Jupyter Notebook.

JupyterLab is the next-generation web-based user interface for Project Jupyter, it's a web-based interactive development environment for Jupyter notebooks, code, and data. JupyterLab is flexible: configure and arrange the user interface to support a wide range of workflows in data science, scientific computing, and machine learning. JupyterLab is extensible and modular: write plugins that add new components and integrate with existing ones.

Compared to the classical web user interface where users can manage Jupyter Notebooks (available at `http://<JupyterLabIP>:8888/tree`) JupyterLab (available at `http://<JupyterLabIP>:8888/lab`) provides a more modern user interface where users can install extensions to satisfy their needs and improve their productivity using the Extension Manager.

For more information on how to use the JupyterLab web-based user interface, visit the official documentation of JupyterLab.

Features

- creating a node through contextualisation
- utilising health check against a predefined port

Prerequisites

- accessing a cloud through an Occopus-compatible interface (e.g EC2, Nova, Azure, etc.)
- target cloud contains an Ubuntu 18.04 image with cloud-init support

Download

You can download the example as tutorials.examples.jupyterlab .

Note: In this tutorial, we will use nova cloud resources (based on our nova tutorials in the basic tutorial section). However, feel free to use any Occopus-compatible cloud resource for the nodes, but we suggest to instantiate all nodes in the same cloud.

Steps

1. Open the file `nodes/node_definitions.yaml` and edit the resource section of the nodes labelled by `node_def:`.

- you must select an *Occopus compatible resource plugin*
- you can find and specify the relevant *list of attributes for the plugin*
- you may follow the help on *collecting the values of the attributes for the plugin*
- you may find a resource template for the plugin in the *resource plugin tutorials*

The downloadable package for this example contains a resource template for the Nova plugin.

Important: For the JupyterLab extensions to work properly, the recommended resources are `VCPU:2`, `RAM:4GB`

Important: Do not modify the values of the contextualisation and the health_check section's attribute!

Note: If you want Occopus to monitor (health_check) your initiated virtual machine and it is to be deployed in a different network, make sure you assign public (floating) IP to the node.

2. Open the file `nodes/infra-jupyterlab.yaml` and edit the variables section labelled by `variables`. The default username is "jovyan" and the default password is "lpds". Change the value of `pwd_jupyterlab` to a safe password!

Important: Make sure the default password is changed, because the JupyterLab environment is exposed publicly on the Internet and anyone with access to the password could execute arbitrary code on the underlying virtual machine with root privileges!

1. Services on the virtual machine should be available from outside, therefore some port numbers must be opened for the VM executing the components. Clouds implement port opening various way (e.g. security groups for OpenStack, etc). Make sure you implement port opening in your cloud for the following port ranges:

Protocol	Port(s)	Service
TCP	22	SSH
TCP	8888	Jupyter Notebook

2. Make sure your authentication information is set correctly in your authentication file. You must set your authentication data for the `resource` you would like to use. Setting authentication information is described *here*.

3. Load the node definitions into the database. Make sure the proper virtualenv is activated!

Important: Occopus takes node definitions from its database when builds up the infrastructure, so importing is necessary whenever the node definition or any imported (e.g. contextualisation) file changes!

```
occopus-import nodes/node_definitions.yaml
```

4. Start deploying the infrastructure.

```
occopus-build infra-jupyterlab.yaml
```

5. After successful finish, the node with `ip address` and `node id` is listed at the end of the logging messages and the identifier of the newly built infrastructure is printed. You can store the identifier of the infrastructure to perform further operations on your infra or alternatively you can query the identifier using the **occopus-maintain** command.

```
List of nodes/instances/addresses:
jupyterlab:
    3116eaf5-89e7-405f-ab94-9550ba1d0a7c
      192.168.xxx.xxx

14032858-d628-40a2-b611-71381bd463fa
```

6. You can start using JupyterLab using your web browser at the following URL:

- JupyterLab: `http://<JupyterLabIP>:8888`

Note: The JupyterLab web user interface is password protected, enter the password that was set in `nodes/infra-jupyterlab.yaml`

---

7. Finally, you may destroy the infrastructure using the infrastructure id returned by `occopus-build`

```
occopus-destroy -i 14032858-d628-40a2-b611-71381bd463fa
```

Always use `virtualenv` for any kind of deployment (testing, building, production, … everything). This ensures there will be no dependency

issues: deployment collisions, missing dependencies in releases, etc. See the virtualenv site for details.

2.15 Build environment

Important: We primarily support **Ubuntu** operating system. The following instruction steps were tested on **Ubuntu 20.04** version.

There are only a few system-wide packages needed:

Git submodules can be used to clone and manage all repositories at once:

Most scripts included in these components rely on **this exact directory structure** (especially testing and documentation dependencies).

There is a Vagrantfile to bootstrap the Occopus environment. After checkout just simply execute `vagrant up` and the virtual machine (created by VirtualBox) should be correctly set up on your machine.

One should work on an Occopus component in a virtualenv. The following shows how to setup the `api` repo. By doing this the `occopus-` commands will appear and work correctly.

```
cd github-occopus
cd api
./reset-env.sh
source env/occopus/bin/activate
```

To try the `occopus-` commands, go to the Tutorial section of the Users' Guide and follow the instructions. There you will find examples prepared for different cloud backends and you can have proper configuration very fast. Users' Guide can be found at the Occopus Website. Alternatively, you can go to the `docs` reporisory and find examples under the `tutorial` directory.

Virtualenvs should be placed in the `env/` directory, so they don't linger in the working tree. `git` will ignore the contents of the `env/` directory so virtualenvs will not be commited accidentally.

2.16 Packaging and deployment

Occopus is split into several Python packages. The packages can be made available on the LPDS internal PyPI server (or *package index*) as Python wheels.

The **internal PyPI server** at the time of writing is on `192.168.155.11`. It is accessible through an Apache proxy using the `pip3.lpds.sztaki.hu` hostname.

Pip can use the following switches to use this package index:

```
pip --trusted-host pip3.lpds.sztaki.hu --find-links http://pip3.lpds.sztaki.hu/packages -
↪-no-index
```

The packages must be **versioned** according to the Semantic Versioning standard.

Development should be done using locally checked out Occopus packages instead of using package dependencies. The `requirements_test.txt` files rely on local dependencies (`pip install -e ...`) to encourage this. This is to avoid uploading too many useless package versions to the package index.

In each repository there is a `package.sh` which generates the wheels to be published. `upload.sh` will upload the output package to the `pip3.lpds.sztaki.hu`, provided the uploader has root access to it.

2.16.1 Managing the internal PyPI server

All dependencies can be found in this index. *Future dependencies* can be added to the index thus:

```
ssh ubuntu@192.168.155.11
cd /opt/packages/
pip download pymongo==2.8        # For example
```

This will download the new dependency from the community servers and installs (caches) it on the internal PyPI server. Locally mirroring and maintaining all used packages in an organization is a common practive anyway.

2.16.2 Dependency Manifests

There are three dependency manifests to be maintained in each package.

`setup.py`

Used by `pip`, this module contains package information, including dependencies.

The dependencies declared here are abstract (versionless) dependencies, declaring only the *relations* among packages.

`requirements.txt`

Used for deployment, this text contains the *real dependencies* of the package, including version constraints.

This file will be used by the users of Occopus, so it must contain package names as references and no source information (cf. `requirements_test.txt`).

This file should contain strict kinds of version specifications (== or possibly ~>), specifying the dependencies against which the package has been tested and verified.

`requirements_test.txt`

This file specifies the packages needed to *test* the package. This includes nosetests, and the current package itself (as a modifiable reference: `-e .`).

Unlike `requirements.txt`, this file references other Occopus packages as local, modifiable repositories (e.g. `-e ../util`). This helps the coding-testing cycle as modifications to other packages will be immediately "visible", without reinstallation.

This file contains the source of the packages (LPDS internal PyPI server) hard-coded.

This file must contain == type version specifications so the testing results are deterministic and reliable.

2.16.3 Creating Packages

The packages can be generated with the `package.sh` script in each package's directory. This script creates and prepares an empty virtualenv and uses `pip wheel` to generate wheels. While building the new wheel, it gathers all its dependencies too, so the resulting `wheelhouse` directory will be a self-contained set of packages that can be vendored. This script relies on the internal PyPI server to gather the dependencies.

2.16.4 Vendoring Packages

The generated wheel packages can be uploaded to the internal PyPI server using the `upload.sh` script in each package's directory. It uploads everything found in the `wheelhouse` directory generated by `package.sh`. This is redundant, as the dependencies already exist on the server, but this makes the upload script dead simple.

When a package is uploaded, its version should be bumped unless it is otherwise justified.

2.16.5 Packages (in *a* topological order)

This is one possible topological ordering of the packages; i.e., they can be built/tested/deployed in this order.

Only interdependencies are annotated here, dependencies on external packages are omitted.

Table 1: **OCCO-Util**

Depends	–
Repository	https://github.com/occopus/util.git
Description	Generic utility functions, configuration, communication, etc. See: `occo.util`.
Testing	The virtualenv must be bootstrapped by executing `occo_test/bootstrap_tests.sh`.

Table 2: **OCCO-Compiler**

Depends	OCCO-Util
Repository	https://github.com/occopus/compiler.git
Description	Compiler module for OCCO. See: `occo.compiler`.

Table 3: **OCCO-InfoBroker**

Depends	OCCO-Util
Repository	https://github.com/occopus/info-broker.git
Description	Information broker for the OCCO system. See: `occo.infobroker`.

Table 4: **OCCO-Enactor**

Depends	OCCO-Util, OCCO-Compiler, OCCO-InfoBroker
Repository	https://github.com/occopus/enactor.git
Description	Active component of the OCCO infrastructure maintenance system. See: `occo.enactor`.

Table 5: **OCCO-InfraProcessor**

Depends	OCCO-Util, OCCO-InfoBroker
Repository	https://github.com/occopus/infra-processor.git
Description	Central processor and synchronizer of the OCCO system. See: `occo.infraprocessor`.

Table 6: **OCCO-ResourceHandler**

Depends	OCCO-Util, OCCO-InfoBroker
Repository	https://github.com/occopus/resource-handler.git
Description	Backend component of the OCCO system, responsible for handling specific kinds of resources. See `occo.resourcehandler`.

Table 7: **OCCO-ConfigManager**

Depends	OCCO-Util, OCCO-InfoBroker
Repository	https://github.com/occopus/config-manager.git
Description	Responsible for provisioning, setting up, configuring, etc. the nodes instantiated by the resource handler.

Table 8: **OCCO-API**

Depends	all OCCO packages
Repository	https://github.com/occopus/api.git
Description	This package combines the primitives provided by other occo packages into higher level services and features. This package is intended to be the top-level package of the Occopus system upon which use-cases, user interfaces can be built.

2.17 API

2.17.1 Basic features for Occopus-based applications

Common functions of a generic Occopus app.

This module can be used to implement OCCO-based applications in a unified way. The module provides features for command-line and file based configuration of an Occopus application, and other generic features.

There are two ways to build an Occopus application.

1. The components provided by Occopus can be used as simple librares: they can be imported and glued together with specialized code, a script.
2. The other way is to use this module as the core of such an application. This module can build an Occopus architecture based on the contents of a YAML config file. (Utilizing the highly dynamic nature of YAML compared to other markup languages.)

The setup function expects a config file either through its cfg_path parameter, or it will try to get the path from the command line, or it will try some default paths (see occo.util.config.config for specifics). See the documentation of setup for details.

data occo.api.occoapp.args = None

Arguments parsed by argparse or an occo.util.config class.

data occo.api.occoapp.configuration = None

Configuration data loaded from the file(s) specified with `--cfg`.

data occo.api.occoapp.infrastructure = None

The OCCO infrastructure defined in the configuration.

func occo.api.occoapp.setup(setup_args=None, cfg_path=None, auth_data_path=None)

Build an Occopus application from configuration.

Parameters:

- `setup_args (function)` – A function that accepts an argparse.ArgumentParser object. This function can set up the argument parser as needed (mainly: add command line arguments).
- `cfg_path (str)` – Optional. The path of the configuration file. If unspecified, other sources will be used (see occo.util.config.config for details).

2.17.1.1 Occopus Configuration

Occopus uses YAML as a configuration language, mainly for its dynamic properties, and its human readability. The parsed configuration is a dictionary, containing both static parameters and objects already instantiated (or executed, sometimes!) by the YAML parser.

The configuration must contain the following items.

logging The logging configuration dictionary that will be used with logging.config.dictConfig to setup logging.

components The components of the Occopus architecture that's need to be built.

resourcehandler The ResourceHandler instance (singleton) to be used by other components (e.g. the InfraProcessor. Multiple backends can be supported by using a basic occo.resourcehandler.ResourceHandler instance here configured with multiple backend clouds/resources.

configmanager The ConfigManager instance (singleton) to be used by other components (e.g. the InfraProcessor. Multiple backends can be supported by using a basic occo.resourcehandler.ConfigManager instance here configured with multiple backend service composers *(This feature is not yet implemented at the time of writing.)*.

uds The storage used by this Occopus application.

2.17.2 Infrastructure Manager

Occopus Infrastructure Manager

class

```
occo.api.manager.InfrastructureManager(process_strategy='sequential')
```

Manages a set of infrastructures. Each submitted infrastructure is assigned an InfrastructureMaintenanceProcess that maintains it. Compiling + storing the infrastructure is decoupled from starting provisioning. This enables the manager to attach to existing, but not provisioned infrastructures. I.e., if the manager fails, it can be restarted and reattached to previously submitted infrastructures.

Parameters: process_strategy (str) – The identifier of the processing strategy for Infrastructure Processor

method

add(infra_desc) Compile, store, and start provisioning the given infrastructure. A simple composition of submit_infrastructure and start_provisioning. **Parameters:** infra_desc – An infrastructure description.

attach(infra_id) Start provisioning an existing infrastructure.

Parameters: infra_id (str) – The identifier of the infrastructure. The infrastructure must be already compiled and stored in the UDS.

detach(infra_id) Stop provisioning an existing infrastructure.

Parameters: infra_id (str) – The identifier of the infrastructure. The infrastructure must be already compiled and stored in the UDS.

get(infra_id) Get the managing process of the given infrastructure.

Parameters: infra_id (str) – The identifier of the infrastructure. **Raises InfrastructureIDNotFoundException:** if the infrastructure is not managed.

start_provisioning(infra_id) Start provisioning the given infrastructure. An InfrastructureMaintenanceProcess is created for the given infrastructure. This process is then stored in a process table so it can be managed. This method can be used to attach the manager to infrastructures already started and having a state in the database.

Parameters: infra_id (str) – The identifier of the infrastructure. The infrastructure must be already compiled and stored in the UDS. **Raises InfrastructureIDTakenException:** when the infrastructure specified is already being managed.

stop_provisioning(infra_id, wait_timeout=60) Stop provisioning the given infrastructure. The managing process of the infrastructure is terminated gracefully, so the infrastructure stops being maintained; the manager is detached from the infrastructure. The infrastructure itself will not be torn down.

Parameters: infra_id (str) – The identifier of the infrastructure. **Raises InfrastructureIDNotFoundException:** if the infrastructure is not managed.

submit_infrastructure(infra_desc) Compile the given infrastructure and stores it in the UDS.

Parameters: infra_desc – An infrastructure description.

tear_down(infra_id) Tear down an infrastructure. This method tears down a running, but unmanaged infrastructure. For this purpose, an Infrastructure Processor is created, so this method does not rely on the Enactor's ability (non-existent at the time of writing) to tear down an infrastructure. If the infrastructure is being provisioned (the manager is attached), this method will fail, and not call stop_provisioning implicitly.

Parameters: infra_id (str) – The identifier of the infrastructure. **Raises ValueError:** if the infrastructure is being maintained by this manager. Call stop_provisioning first, explicitly.

```
occo.api.manager.InfrastructureMaintenanceProcess(infra_id, enactor_interval=10,␣
↪process_strategy='sequential')
```

A process maintaining a single infrastructure. This process consists of an Enactor, and the corresponding Infrastructure Processor. The Enactor is instructed to make a pass at given intervals.

Parameters: `infra_id (str)` – The identifier of the already submitted infrastructure. `enactor_interval (float)` – The number of seconds to elapse between Enactor passes. `process_strategy (str)` – The identifier of the processing strategy for Infrastructure Processor

2.18 Develop documentation

This guide aims to help you get familiar with the Occopus documentation part.

2.18.1 Creating documentation environment locally

To set up a documentation environment you need to have a `Python3` installation with installed `Spinx` and `sphinx_rtd_theme` package.

The documentation tested with the following versions:

- Sphinx - `3.0.3`
- sphinx_rtd_theme - `0.4.3`

To create a local documentation environment just follow the steps (Debian-based OS):

```
sudo apt update && sudo apt install -y python3-pip virtualenv
virtualenv -p python3 ./venv/docs
source venv/docs/bin/activate
git clone https://github.com/occopus/docs.git -b devel
pip install -r docs/sphinx/requirements.txt
cd docs/sphinx/
make html
```

Note: It is recommended to use virtual environment however you can continue without it.

Now you can easily build your own documentation with `make html` command under `docs/sphinx/` path. After the process finished, you can find the built documentation under `docs/sphinx/build`.

2.18.2 Visualize local build

For **testing purposes** you can install nginx and host your documentation. The following steps will help you to do that:

```
sudo apt update && sudo apt install -y nginx
sudo sed -i "s/^        root/        root \/home\/ubuntu\/docs\/sphinx\/build\/html\;/g"␣
↪\
/etc/nginx/sites-available/default
# the sed part could be different in different OS. If it does not work, just replace the␣
↪root
# line with your docs location
sudo service nginx restart
```

After these steps, you can look at the documentation under: `http://[Your_IP_Address]/`

> **Danger:** Nginx config is **not** a valid production ready config! Use **only** for **testing** purposes! If you are able to do that do not expose it to the public (use local network if it is possible).

2.18.3 Helper scripts

Under the documentation repository, there is helperScripts folder with two different helper scripts. It provides quick automation of different tasks.

2.18.3.1 createTarFileFromTutorials.sh

This script creates a tar.gz file from every directory from `docs/tutorials`. It is important to run this script when you modify the description in the tutorials folder. The tar.gz file requires to make tutorials downloadable through hosted documentation (Read the Docs) via raw GitHub URL.

2.18.3.2 updateAbsoluteGithubLinksToChangeBranch.sh

This script aids to help change the GitHub branch absolute path easily through a semi-automated way. The script requires two arguments. First argument is the **current** branch and the second argument is the **target** branch.

The script looks through the following path, and modify the branch if needed:

- `/sphinx/source/*.rst`
- `/tutorials/`

Usually, there are two common usages but you can modify as you wish:

This way the script change every `master` branch reference to the `devel` branch.

```
$ ./updateAbsoluteGithubLinksToChangeBranch.sh master devel
```

The other way does the opposite. The script change every `devel` branch reference to the `master` branch.

```
$ ./updateAbsoluteGithubLinksToChangeBranch.sh devel master
```

2.18.4 Read the Docs build

Every tag creates a new version for the Occopus documentation site. Occopus documentation is hosted by Read the Docs (RTD) at the URL: https://occopus.readthedocs.io.

The `master` branch defines the lastest tag in RTD which is considered as the stable version of the documentation. Each releases of the master branch is compiled and shown by RTD as versions.

Actual version of the `devel` branch is also continuesly refreshed by RTD and shown under a hidden (/devel) URL. Optionally, it can be built privately on your local machine as described in sections *Creating documentation environment locally* and *Visualize local build*.

If there is a new tag or commit in master or devel branch in Occopus Docs repository RTD will rebuild the whole documentation. After a while the documentation will be available with the changes through the documentation URL.