ONEedge.io

A Software-defined Edge Computing Solution

D3.3. Software Report - c

Software Report

Version 1.0

3 November 2021

Abstract

This report summarizes the design of the technology components that have been implemented as part of the Third Innovation Cycle (M17-M23), as well as the full details of each of the software requirements that are being addressed as part of the development of such components. For each software requirement, this document provides a full description, a list of detailed requirements and specifications, a description of its architecture and components, the data model, and relevant changes applied to the API and Interfaces.

Copyright © 2021 OpenNebula Systems SL. All rights reserved.

This project has received funding from the European Union's Horizon 2020 Research and Innovation Programme under Grant Agreement No 880412.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Deliverable Metadata

Project Title:	A Software-defined Edge Computing Solution
Project Acronym:	ONEedge
Call:	H2020-SMEInst-2018-2020-2
Grant Agreement:	880412
WP number and Title:	WP3. Product Innovation
Nature:	R: Report
Dissemination Level:	PU: Public
Version:	1.0
Contractual Date of Delivery:	30/9/2021
Actual Date of Delivery:	3/11/2021
Lead Authors:	Vlastimil Holer, Rubén S. Montero and Constantino Vázquez
Authors:	Sergio Betanzos, Pavel Czerny, Ricardo Díaz, Jim Freeman, Christian González, Alejandro Huertas and Jorge M. Lobo
Status:	Submitted

Document History

Version	Issue Date	Status[1]	Content and changes
1.0	3/11/2021	Submitted	First final version of the D3.3 report

[1] A deliverable can be in one of these stages: Draft, Peer-Reviewed, Submitted and Approved.

Executive Summary

The purpose of deliverable D3.3 is to offer a summary of the design of the technology components that have been implemented in the Third Innovation Cycle (M17-M23), as well as to provide the full details of each of the software requirements that are being addressed as part of the development of such components. For each software requirement, this document provides a full description, a list of detailed requirements and specifications, a description of its architecture and components, the data model, and relevant changes applied to the API and Interfaces.

During the Third Innovation Cycle (M17-M23), the project mostly focused on those software requirements needed to achieve the third milestone in M23, which is the base functionality needed to meet networking & storage integration, and mostly its release as a standalone managed service (On-demand Edge Cloud Service).

The work carried out during this Third Innovation Cycle involved software requirements from components CPNT1, CPNT3, CPNT4 and CPNT5, with a special focus on the completion and integration of all components to release a first version of the On-demand Edge Cloud Platform service (CPNT1) and the deployment and provision of edge infrastructures (CPNT4). During the Third Innovation Cycle, we have developed campaigns that are specific to the ONEedge hosted service (standalone commercial solution) and not to be incorporated into OpenNebula. These are some of the main new features that have been implemented as part of this process:

- First version of the On-demand Edge Cloud Service (ONEedge “Edge as a Service” hosted instances) with definition of basic services and security requirements, key performance indicators, 24/7 health monitoring and alerts.
- Automatic customer environment deployment, configuration and bootstrapping, and complete life-cycle management of the ONEedge instances.
- ONEedge hosted framework implemented following a GitOps paradigm and leveraging Github WebUI portal and tools to simplify request, monitoring and basic management, even for non-technical operators.
- Ability to dynamically load providers into OneProvision and extension of the OneProvision GUI, which features the Edge Provider Catalog Service, to scan and load new drivers without the need to update to a new release or modify any lines of code.
- Development of guides to create new providers that can be made dynamically available in the Edge Provider Catalog Service.
- Enhancements to provision and components to support transparent secure connection among geographically distributed edge locations.
- Development of new drivers for Google Compute Engine, Vultr (bare metal and virtual instances), and Digital Ocean.
- Development of new drivers for on-prem far-edge provisions.
- Support for ARM devices at edge locations.
- Addition of MetalLB load balancer to K8s appliance for better networking in Kubernetes clusters deployed at cloud and edge locations.
- New Sunstone GUI beta built using React/Redux and delivered by the FireEdge server.

Table of Contents

1. Edge Instance Manager (CPNT1) 5

[SR1.1] Simple Product Deployment 5

[SR1.3] Instance Management 7

[SR1.4] Subscription Management 9

[SR1.5] Web Control Interface (GUI) 11

2. Edge Workload Orchestration and Management (CPNT2) 14

3. Edge Provider Selection (CPNT3) 15

[SR3.1] Edge Provider Catalog Service 15

[SR3.4] Driver Maintenance Process 18

4. Edge Infrastructure Provision and Deployment (CPNT4) 21

[SR4.4] Inter-edge Networking Deployment Scenario 21

[SR4.5] Drivers for Host Provision 24

[SR4.9] Support on-Premises far-Edge for Resource Provisioning 32

[SR4.10] Support ARM for Resource Provisioning 34

5. Edge Apps Marketplace (CPNT5) 36

[SR5.2] Built-in Management of Application Containers Engine 36

[SR5.5] Edge Market GUI Developments 39

1. Edge Instance Manager (CPNT1)

[SR1.1] Simple Product Deployment

Description

Deployment based on application containers has evolved into the ONEedge hosted deployment. This change allows a better life-cycle management as well as improving deployment automation. Moreover, it is the base of a new business model.

A ONEedge hosted instance is a standalone installation of the OpenNebula frontend, FireEdge and OneProvision services. The instance is available on the Internet via a hostname in the opennebula.cloud domain, public IP address and a valid HTTPS certificate. All components are preconfigured to be ready to use, and FireEdge with OneProvision are prepared to quickly provision new HW resources from a given provider list.

Requirements and Specifications

- The deployment is fully automated and it is periodically tested.
- GitOps approach to the deployment and management of the hosted instances. The state of the instances is represented with git objects.
- Simple operations: if you want to create a new OpenNebula hosted deploy or update an existing one, you only need to update the repository—the automated process handles everything from that point.

Architecture and Components

Terraform Templates
Current version of the hosted OpenNebula deployments includes Terraform templates for hosted instances in AWS cloud. The following resources need to be pre-created in a given AWS region to support the instances:

- VPC, CIDR: 10.0.0.0/16
- AWS Internet gateway
- AWS default route in given VPC

New providers or AWS regions can be added easily because other components are fully independent.

Ansible Roles and Playbooks
Once the compute resources are allocated, Ansible is used to configure them and install OpenNebula and the associated ONEedge components. In the following table we detail the list of ansible roles used for each instance, their purpose, and associated services.

Ansible Role	Purpose	Service
`opennebula-repository`	Enable enterprise ONE repository	`n/a`
`opennebula-server`	Install ONE packages, install and bootstrap database, configures ONE	`opennebula, opennebula-scheduler, opennebula-flow, opennebula-gate, opennebula-hem, opennebula-novnc, opennebula-sunstone`
`opennebula-provision`	Installs opennebula-oneprovison	`n/a`

opennebula-fireedge	Installs opennbula-fireedge	opennebula-fireedge
opennebula-passenger	Installs passenger, httpd server	httpd
opennebula-prometheus	Installs and configures prometheus	prometheus
docker	Installs docker for dockerhub images	docker
certbot	Retrieves and maintains Let's Encrypt SSL certificate	n/a

Tools and Tests

To put the GitOps together with the Terraform and Ansible, we developed a tool which allows the deployments to be fully automatic. The deployment is operated in the same way that it is also regularly tested.

Data Model

The attributes for the deployment are saved in a Git repository in a YAML file `options.yaml`. Any changes to the deployment are made by updating this file, which triggers automatic reconfiguration of the instance.

This is an example of options.yaml:

```
---
:name: deployment name
:company: company name
:mail: deployment contact email
:one_mail: OpenNebula contact email
:subdomain: name of the subdomain
:type: single/ha
:aws_region: eu-central-1
:aws_size: t2.large
:provider: aws
```

Similarly, the state of the deployment is held in a state.yaml file.

```
---
:state: running
:signature: af27d42bc6b6b384ceeef186c1f658d2
:oneadmin_password: ---
```

[SR1.3] Instance Management

Description

ONEedge instances are managed using a GitOps approach, where the state is represented as git objects. The management of the life-cycle is controlled by performing updates on these objects, which ultimately triggers actions that perform the necessary changes to match the new state.

Requirements and Specifications

The main requirement for interfacing with the ONEedge hosted instances is that it should be simple so that it can be used by community and pre-sales teams to set up demonstrations or Proof-of-Concept (PoC) environments for potential customers.

Architecture and Components

The following picture represents the life-cycle and states of a hosted instance, as well as the associated actions that trigger the instance changes.

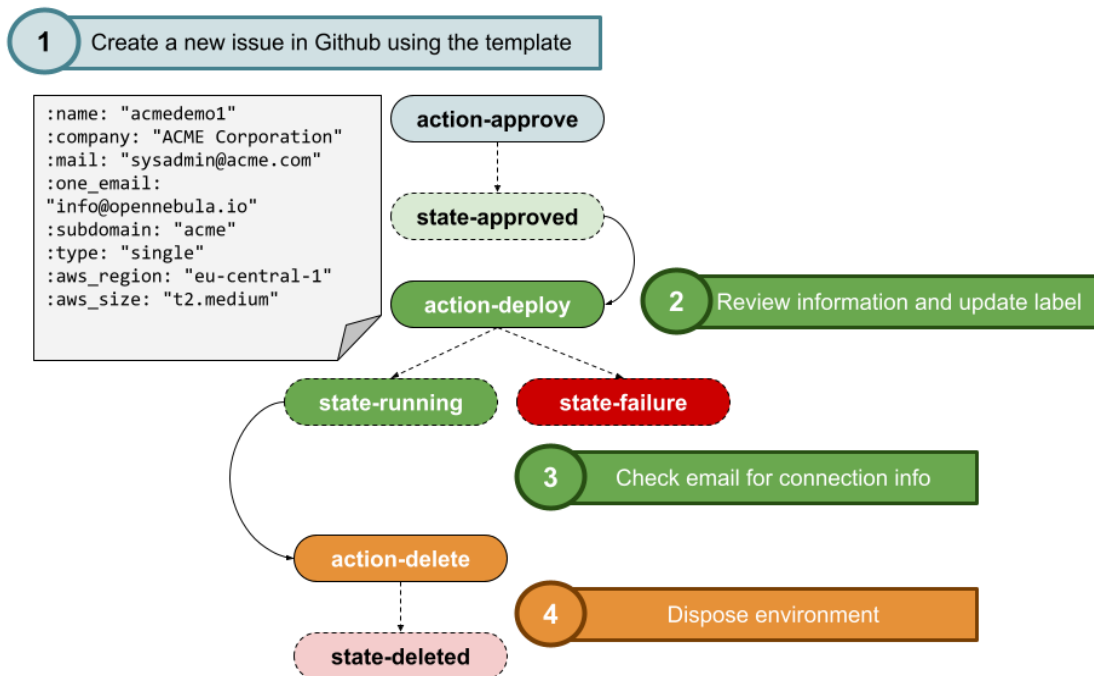


Figure 1.3.1: ONEedge instance states and life-cycle.

The ONEedge hosted instance is in one of the following states:

- new: the instance is created by filling a new GitHub issue
- running: the instance is running and fully configured
- failure: the instance could not be deployed because of an error during the creation or configuration phases
- terminated: the instance is terminated
- purged: configuration and state files are deleted

Changing of the instance state is done by following actions, that are implemented by adding the corresponding label to the issue associated with the ONEedge hosted instance:

- approve
- deploy
- update
- delete
- purge

Data Model

The state of the instance is stored in state.yaml as described above. Another part of the definition of a running deployment instance is the terraform template and state file. Having all the files persisted in the Git allows the deployment to be easily updated later.

API and Interfaces

Not applicable.

[SR1.4] Subscription Management

Description

ONEedge hosted environments have been integrated with Zendesk. We use the Zendesk portal to provide commercial support to our customers. Additionally, the ONEedge hosted environment can be used to showcase our technology, so this integration is optional; a hosted environment can be created without creating any user in Zendesk.

Requirements and Specifications

The main requirement is to be able to automatically create a new customer organization in Zendesk and to add users to this organization. Once the users are added, they receive a welcome email with all the information to use the support portal, their new hosted environment, and how to open new support requests.

The converse operation is also supported: once the hosted environment is terminated, the organization and the users can be optionally removed from Zendesk.

The organization also has various tags to identify the services they have contracted. These tags can be automatically added to the user if they are included in the ONEedge hosted environment creation request. There are some defaults added to identify the ONEedge hosted product: `active`, `evaluation`, `standard_technical`, `1_zone`.

Architecture and Components

The integration with Zendesk uses some information included in the creation request for the hosted environment. By default, user creation in Zendesk is not activated, and needs to be explicitly set to true, for example:

```
…
:use_zendesk: true
:zendesk_organization:
   :name: testing
   :tags:
      - europe
:zendesk_users:
   - :name: Testing User
     :email testing@opennebula.io
```

Figure 1.4.1: Activate Zendesk integration

Once the integration has been activated, more information needs to be provided:

- `zendesk_organization` contains the `name` and `tags` that are added (together with the default ones) to each user.
- `zendesk_users`, a list of users to add to this organization, for each of them the name and the email must be provided:

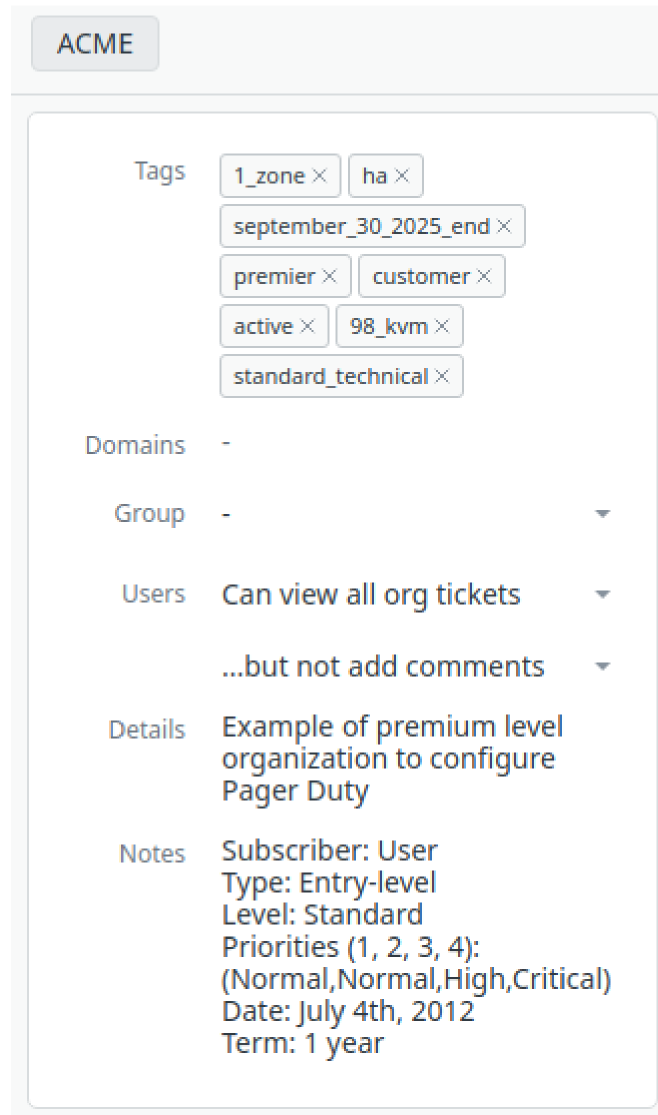


Figure 1.4.2: Zendesk organization

Data Model

There is no special data model. All the integration is done via GitHub issue as explained earlier in this document

API and Interfaces

The integration is done using the official Zendesk API for Ruby[2].

[2] https://github.com/zendesk/zendesk_api_client_rb/wiki

[SR1.5] Web Control Interface (GUI)

Description

GitOps is a DevOps practice that uses a git repository to represent the state of the infrastructure. Changes to the infrastructure are implemented by commits to the repository or changes to git objects including: files, Github issues, labels and/or actions. This approach allows us to use the Github interface so the user doesn't need to run any Git commands directly.

Requirements and Specifications

Using the Github web interface allows community and pre-sales teams to operate the deployments without the need to interact with lowevel Git commands or infrastructure.

Architecture and Components

To create a new hosted deployment instance, the user first needs to create a Github issue and store the deployment definition there. The figure below shows a screenshot of this process:

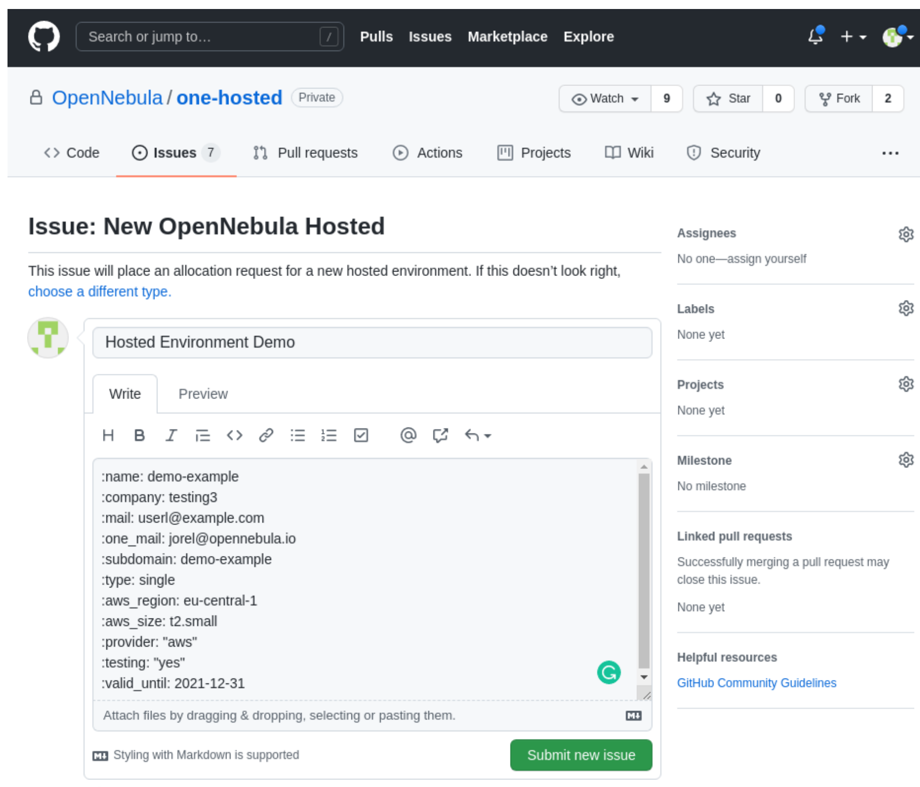


Figure 1.5.1: Creation interface for a new ONEedge hosted environment

Actions are triggered by adding labels to the issue created to represent the hosted environment. For example, for approving the deployment and storing its definition into the Git repository, the user needs to add a `[action-approve]` label to the issue. See image below:

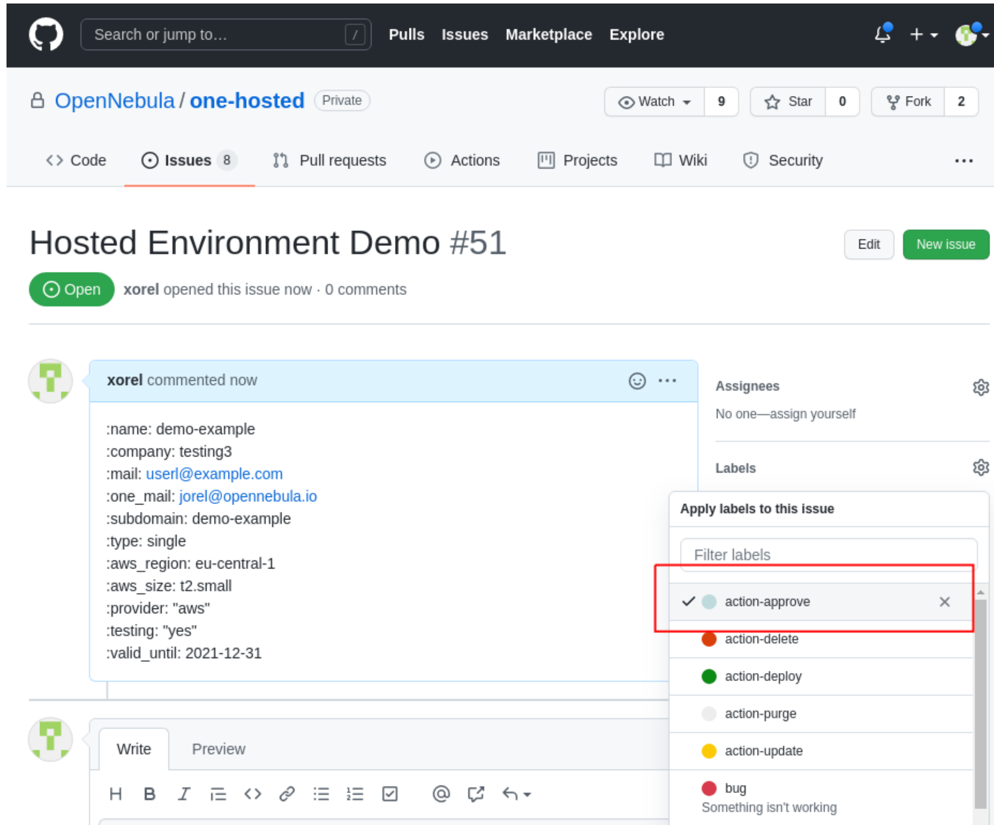


Figure 1.5.2: Approving the deployment of ONEedge hosted environment

Then, a Github action is triggered and once it finishes it removes the action label and adds a state label, `[state-approved]` in this case. Similarly, to run the deployment we add `[action-deploy]` label and once it's running, the issue will have `[state-running]` label attached to it. In the action details we will find the deployment report which is also sent by email to the user. Should the deployment fail for some reason, the issue will be tagged with a `[state-failure]` label and the action logs can be inspected to debug the problem. The figure below shows the logs of a deployment action.

Data Model

States:

- state-approved
- state-running
- state-failure
- state-deleted
- state-purged

Triggers:

- action-approve
- action-deploy
- action-update
- action-delete
- action-purge

one-hosted-deploy
succeeded 24 minutes ago in 22m 42s

Search logs

Run ./tools/one-hosted.rb deploy|update|delete|purge 22m 32s

```
1297  create mode 100644 deployments/demo-example/site.tf
1298  create mode 100644 deployments/demo-example/site.yml
1299  create mode 100644 deployments/demo-example/state.yml
1300  create mode 100644 deployments/demo-example/terraform.tfstate
1301 To https://github.com/OpenNebula/one-hosted
1302    d305c26..692f253  HEAD -> master
1303 OneHosted report action-deploy
1304
1305 provider: aws
1306 region: eu-central-1
1307 size: t2.small
1308
1309 sunstone: https://demo-example.opennebula.cloud
1310 fireedge: https://demo-example.opennebula.cloud/fireedge
1311 public_ip: 3.121.195.0
1312 instance_id: i-081a847ce36f82b6b
1313 oneadmin_password: aCnzED7Z
1314
1315 login with 'ssh -l ubuntu 3.121.195.0'
1316
1317 Tests results:
1318
1319 One-hosted test
1320   response http
1321   response on https
1322
1323 Finished in 6.5 seconds (files took 0.20542 seconds to load)
1324 2 examples, 0 failures
1325
1326
1327 Removing label state-approved from issue 51
1328 Adding label state-running
1329 Removing label action-deploy from issue 51
```

Figure 1.5.3: Logs for the deployment action of a ONEedge action.

API and Interfaces

The actions and ONEedge interface are based on the Github interface and Github API.[3]

[3] https://docs.github.com/en/rest

2. Edge Workload Orchestration and Management (CPNT2)

No activity done during the cycle.

3. Edge Provider Selection (CPNT3)

[SR3.1] Edge Provider Catalog Service

Description

The goal of this software requirement is to provide a Catalog Service, currently implemented in the OneProvision GUI. In this cycle, and to terminate the issue, we added the ability to load new drivers. In this way, the OneProvision component is able to dynamically load the drivers on execution time. This means it is easier to add a new provider, as OneProvision is able to detect them without having to make any changes in the distribution code.

Although the providers are exposed in the GUI, they are also available in the CLI, to be used in the same way. It is important to note that the functionality remains the same; this only changes the method of adding new providers into OpenNebula.

Requirements and Specifications

The following changes have been made to OneProvision so admins do not need to touch the distribution code to support new providers:

- Avoid having references to the specific provider names in the code. Currently there is no distinction between them as they are generalized.
- Avoid having references to the directory where the information is stored. Each provider uses different ERB files to generate the Terraform templates, stored in a folder that is scanned automatically.
- Add a method to load the existing providers located on */usr/lib/one/oneprovision/lib/terraform/providers.*
- Create an example class that can be copied to add the new provider, as follows:

```
require 'terraform/terraform'

# Module OneProvision
module OneProvision

    # <<PROVIDER NAME>> Terraform Provider
    class <<PROVIDER CLASS>> < Terraform

        NAME = Terraform.append_provider(__FILE__, name)

        # OpenNebula - Terraform equivalence
        TYPES = {
            :cluster   => '<<TERRAFORM RESOURCE>>',
            :datastore => '<<TERRAFORM RESOURCE>>',
            :host      => '<<TERRAFORM RESOURCE>>',
            :network   => '<<TERRAFORM RESOURCE>>'
        }

        KEYS = %w[<<PROPVIDER CONNECTION INFO>>]

        # Class constructor
        #
        # @param provider [Provider]
        # @param state    [String] Terraform state in base64
        # @param conf     [String] Terraform config state in base64
        def initialize(provider, state, conf)
            # If credentials are stored into a file, set this variable to true
            # If not, leave it as it is
            @file_credentials = false
```

```
            super
        end

        # Get user data to add into the VM
        #
        # @param ssh_key [String] SSH keys to add
        def user_data(ssh_key)
            <<IMPLEMENT THIS METHOD IF NEEDED, IF NOT YOU CAN DELETE IT>>
        end

    end

end
```

Figure 3.1.1: Example class

Architecture and Components

The main changes are located on Terraform classes that are in charge of deploying the servers into the remote providers. All the distinctions between the providers have been removed, so now Terraform is able to work with any provider. To achieve this, a new attribute NAME has been added to the class. This allows Terraform to know which provider is being used:

```
NAME = Terraform.append_provider(__FILE__, name)
```

Figure 3.1.2: Name attribute

The method to load the providers has been added in all the necessary places. This is across all the OneProvision code, as the providers are used in many places. This is basically a static method that reads all the available files and loads the provider written on them:

```
Terraform.p_load
```

Figure 3.1.3: Load providers methods

Some changes have been added to OneProvision GUI so that it is able to read the logos of the new provider.

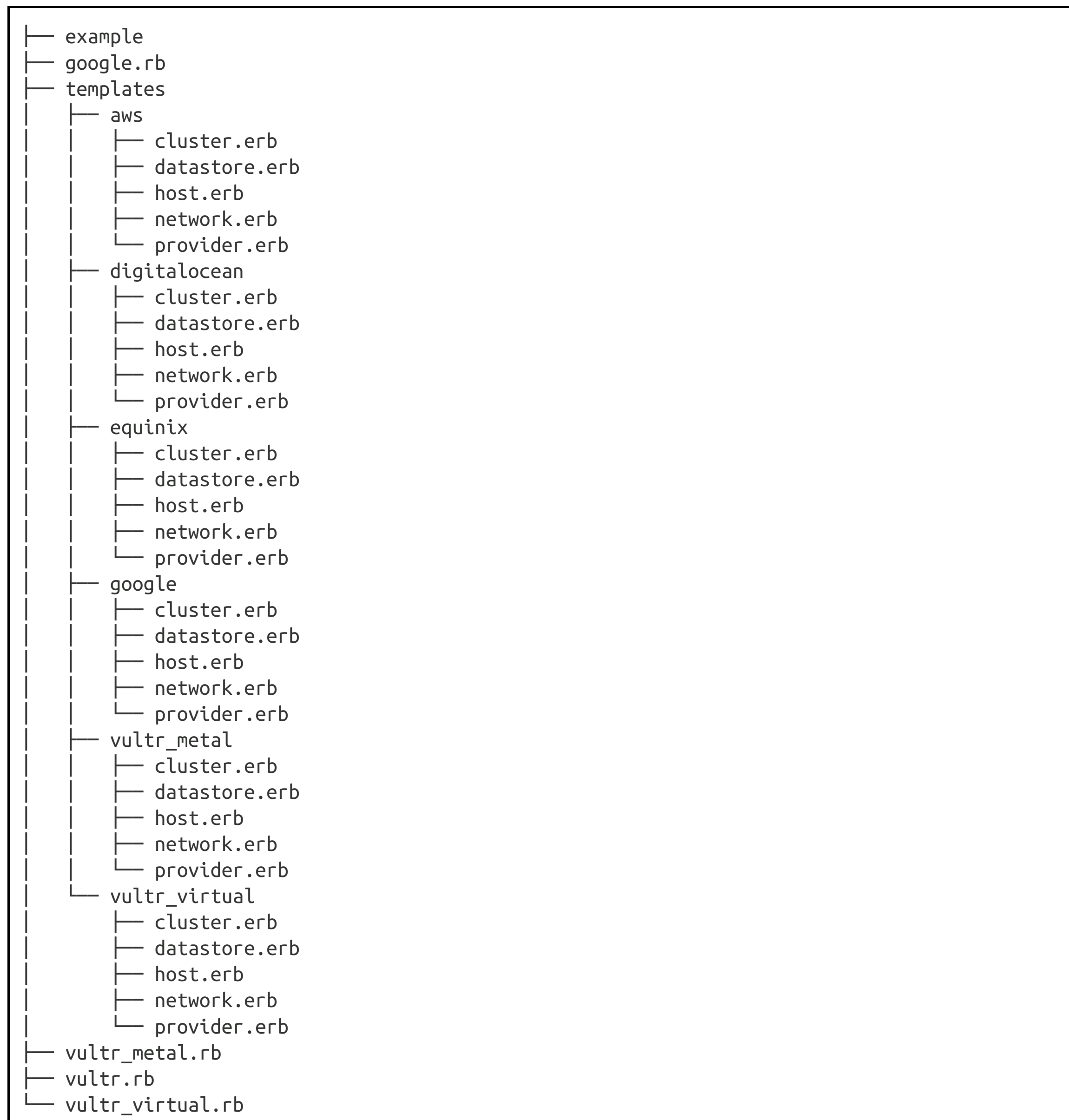
Data Model

The key part here is the structure of the directory where the providers are stored. Although the directory itself has not changed, the way to read it has. The directory is located in */usr/lib/one/oneprovision/lib/terraform/providers* and contains:

- The Provider class: these are the files with .rb extensions, it is very important to note that only the files with this extension are loaded.
- The Provider templates: this directory contains the ERB for each provider.

The name of the provider file class should be the same (without .rb) as the folder inside the templates directory. For example, if the provider is called *digitalocean.rb,* the folder inside templates should be named *digitalocean*. The current state of the folder, showing the supported providers, is shown in the following figure.

```
providers
├── aws.rb
├── digitalocean.rb
├── dummy.rb
├── equinix.rb
```



```
├── example
├── google.rb
├── templates
│   ├── aws
│   │   ├── cluster.erb
│   │   ├── datastore.erb
│   │   ├── host.erb
│   │   ├── network.erb
│   │   └── provider.erb
│   ├── digitalocean
│   │   ├── cluster.erb
│   │   ├── datastore.erb
│   │   ├── host.erb
│   │   ├── network.erb
│   │   └── provider.erb
│   ├── equinix
│   │   ├── cluster.erb
│   │   ├── datastore.erb
│   │   ├── host.erb
│   │   ├── network.erb
│   │   └── provider.erb
│   ├── google
│   │   ├── cluster.erb
│   │   ├── datastore.erb
│   │   ├── host.erb
│   │   ├── network.erb
│   │   └── provider.erb
│   ├── vultr_metal
│   │   ├── cluster.erb
│   │   ├── datastore.erb
│   │   ├── host.erb
│   │   ├── network.erb
│   │   └── provider.erb
│   └── vultr_virtual
│       ├── cluster.erb
│       ├── datastore.erb
│       ├── host.erb
│       ├── network.erb
│       └── provider.erb
├── vultr_metal.rb
├── vultr.rb
└── vultr_virtual.rb
```

Figure 3.1.4: Current filesystem structure

API and Interfaces

There are no changes in the API, but rather in the structure and semantics of the providers stored in the front-end filesystem.

The admin interface to add a new provider uses typical linux shell tools:

1. Copy the file */usr/lib/one/oneprovision/lib/terraform/providers/example* to */usr/lib/one/oneprovision/lib/terraform/providers/<provider>.rb*
2. Fill all the information inside of it.
3. Create the ERB templates inside */usr/lib/one/oneprovision/lib/terraform/providers/templates/<provider>*
4. Test the new provider.

[SR3.4] Driver Maintenance Process

Description

The goal of this software requirement is to provide a process to add new infrastructure providers into OpenNebula with full support to be used in OneProvision. For this purpose, a guide has been created, with all the steps from the beginning until the Ansible recipes to configure the servers.

The process has multiple steps:

- Add the provider class.
- Add Terraform ERB files.
- Add Ansible playbook.
- Add provision templates.

Requirements and Specifications

Each driver requires the following elements:

- Terraform support: it needs a Terraform provider to be able to deploy and destroy the servers and all the resources, such as networks, VPC, etc. If the provider is not supported by Terraform, it cannot be added into OpenNebula.
- Network model: networking is consistently the most specific part of a provider. In some cases it needs to use the OpenNebula IP address management (IPAM) submodule, so an IPAM driver needs to be implemented which includes all the scripts that manage the IPAM functionality, with developer guides available. Alternatively, a different way to give access to networking to virtual machines can be used: in this case, it needs to be implemented as a network driver into OpenNebula. Both of the methods would require API calls to the remote provider and this can be done using an external API or just directly using HTTP requests.

Architecture and Components

Each driver is a combination of multiple components and behind each of them is the idea to place the driver in combination with FireEdge, so both can work together without any issue. they are the following:

- **Run-time class**: implements the driver functionality. Classes are created dynamically (check section SR3.1 for more information), so the user can implement their own drivers in an easy way. The class must follow these rules:
  - The hash TYPES contains the relationship between the OpenNebula object and the Terraform name, e.g.:

```
TYPES = {
   :cluster   => 'digitalocean_vpc',
   :datastore => 'digitalocean_volume',
   :host      => 'digitalocean_droplet',
   :network   => ''
}
```

Figure 3.4.1: Driver TYPES example

  - The array KEYS contains the name of the keys that are used to authenticate with the remote provider, e.g.:

```
KEYS = %w[token region]
```

Figure 3.4.2: Driver KEYS example

- The method user_data is used to add the SSH keys to the server in order to access it, e.g.:

```
def user_data(ssh_key)
    user_data = "#cloud-config\n"

    user_data << "users:\n"
    user_data << " - name: install\n"
    user_data << "   groups: sudo\n"
    user_data << "   shell: /bin/bash\n"
    user_data << "   sudo: ['ALL=(ALL) NOPASSWD:ALL']\n"
    user_data << "   ssh_authorized_keys:\n"

    ssh_key.split("\n").each {|key| user_data << "- #{key}\n" }

    user_data = user_data.gsub("\n", '\\n')
end
```

Figure 3.4.3: Function user_data example

- **Terraform templates**: they are written in ERB format and are used to create the YAML file that Terraform needs to deploy the resources. There is a template for each resource, e.g.:

```
digitalocean
├── cluster.erb   # Common resources for the cluster
├── datastore.erb # Datastore representation
├── host.erb      # Host representation
├── network.erb   # Network representation
└── provider.erb  # Terraform provider definition
```

Figure 3.4.4: Terraform ERB files

- **Provision template**: YAML file that represents the provision. This is all the information that is needed to deploy it:
    - Provision name.
    - Ansible playbook used to configure the hosts.
    - Some provision defaults like the image to deploy, the instance type, etc.
    - OpenNebula cluster definition.
    - Common information for all the provisions.

```
name: 'digitalocean-cluster'

extends:
    - common.d/defaults.yml
    - common.d/resources.yml
    - common.d/hosts.yml
    - digitalocean.d/datastores.yml
    - digitalocean.d/fireedge.yml
    - digitalocean.d/inputs.yml
    - digitalocean.d/networks.yml

playbook:
  - digitalocean

defaults:
  provision:
    provider_name: 'digitalocean'
    image: "${input.digitalocean_image}"
    size: "${input.digitalocean_size}"
  connection:
```

```
    remote_user: 'install'

cluster:
  name: "${provision}"
  description: 'Digitalocean virtual edge cluster'
  reserved_cpu: '0'
  reserved_mem: '0'
  datastores:
    - 1
    - 2
```

Figure 3.4..5: Digitalocean provision YAML

- **Provider template**: YAML file that represents the provider. This is the information that is needed to interact with it and the location in the world in which to deploy the servers.

Data Model

There are two key parts of the procedure to create a new provider driver: the provision template that defines the provision that is going to be deployed, and the Ansible roles that are going to configure the provision:

- The **provision template** is a YAML-formatted file with parameters specifying the new physical resources to be provisioned with the following attributes:
    - Header (name, configuration playbook)
    - Global default parameters for
        - remote connection (SSH)
        - host provision driver
        - host configuration tunables
    - OpenNebula infrastructure objects (cluster, hosts, datastores, virtual networks) to deploy with overrides to the global defaults above
    - OpenNebula virtual objects (images, templates, vnet templates, marketplace apps, service templates)
- **Ansible** is used to configure the provision.
    - It needs to implement the following parts:
        - task: a single configuration step
        - role: a set of related tasks
        - playbook: a set of roles/tasks to configure several components at once
    - The configuration phase can be parameterized to slightly change the configuration process. These custom parameters are specified in the configuration section of the provision template. In most cases, the general defaults should meet requirements. All code for Ansible (tasks, roles, playbooks) is installed in */usr/share/one/oneprovision/ansible/*.

4. Edge Infrastructure Provision and Deployment (CPNT4)

[SR4.4] Inter-edge Networking Deployment Scenario

Description

In general, connections with services are performed using a TCP/IP endpoint (UDP in case of connectionless communication) identified by the tuple: `(SOURCE_IP:SOURCE_PORT, DESTINATION_IP:DESTINATION_PORT)`. However, in some situations, due to the IP address being a limited resource, one or several IP addresses have to be shared among different services. The *Port Forwarding* technique (also known as *NodePort* by other systems like Kubernetes) consists of still identifying the service by using an IP address and a TCP/UDP port for external connections, and forwarding that connection to an internal IP address (usually in a private network) and a UDP/TCP port.

Requirements and Specifications

We assume the following requirements for the network connection of the applications:

- A public endpoint composed of a public IP address and a UDP/TCP port.
- A private endpoint composed of a private IP address and a UDP/TCP port.
- A forwarding mechanism to send the traffic from/to the private and public endpoints.

Therefore, it can be considered that when using Port Forwarding, services are identified by the TCP/UDP port (layer 4, L4) listening on the public IP.

For example, let's assume two virtual machines with private IP addresses *IP1* and *IP2* are hosted by a machine (host) with a public IP address *IP3*. Both VMs have the SSH service listening on port TCP/22. The external (public) SSH service can be identified by *IP3:9022* endpoint for the first VM and by *IP3:9122* endpoint for the second. Using Port Forwarding, connections to *IP3:9022* will be forwarded to *IP1:22,* and connections to *IP3:9122* will be forwarded to *IP2:22*.

Architecture and Components

The following picture shows the network connectivity architecture for the Port Forwarding technique:

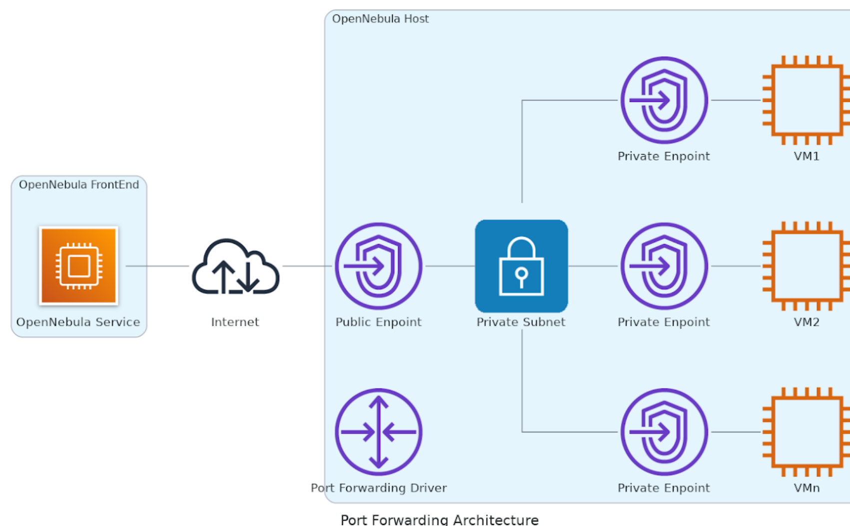


Figure 4.4.1: Port Forwarding technique to expose VM ports.

The *nodeport* driver (`nodeport.rb`) in OpenNebula configures the host routing tables to forward the traffic from/to the public endpoint to the target private endpoint using *iptables*. When a virtual machine is created, the following *iptables* rules are added:

```
iptables -t nat -I PREROUTING -p tcp --dport <EXTERNAL_PORT_RANGE>  -j DNAT --to-destination
<INTERNAL_IP>:<INTERNAL_PORT_RANGE>

iptables -t nat -A POSTROUTING -s <INTERNAL_IP> -j MASQUERADE
```

Figure 4.4.2: SNAT/DNAT `iptables` rules to set up port forwarding

The first rule below is in charge of changing the destination public (external) IP address and port of the incoming packets by the private (internal) IP address and port of the virtual machine; the second rule is in charge of restoring the external IP address and port of packet responses when they are about to go out to the Internet.

Data Model

Virtual networks should be able to allocate and manage the port ranges associated with each IP lease. In the figure below, we show the definition of a *nodeport* virtual network:

```
NAME=<NAME>
BRIDGE=<BRIDGE>
VN_MAD="nodeport"
AR = [
  IP=<INTERNAL_IP>,
  PORT_START=<EXTERNAL_PORT_START>,
  PORT_SIZE=<EXTERNAL_PORT_RANGE_SIZE>,
  TYPE="IP4"
]
```

Figure 4.4.3: Network Template that defines a nodeport association

Each lease in an Address Range (AR) is identified by an ID and the corresponding port ranges (e.g. 4000:4099, 4100:4199, 4200:4299…) following this formula:

$$\text{RANGE}(i) = [P_{START} + i - P_{START} + i + P_{SIZE}]$$

For example, the following figure shows a virtual network template that defines a port range of 100 ports mapped for each lease from port 9000:

```
NAME=”mynodeportvnet”
BRIDGE="br0"
BRIDGE_TYPE="linux"
VN_MAD="nodeport"
AR = [
  GATEWAY="192.168.23.1",
  IP="192.168.23.2",
  PORT_SIZE="100",
  PORT_START="9000",
  PROVISION_ID="1",
  SIZE="250",
  TYPE="IP4",
  IP_END="192.168.23.251"
]
```

Figure 4.4.4: Network Template example for port ranges of size 100 from port 9000

When a VM requests a lease from the network it also gets the associated port range as shown below:

```
ADDRESS RANGE POOL
AR 0
SIZE           : 250
LEASES         : 2

RANGE                                  FIRST                               LAST
MAC                        02:00:c0:a8:17:02                  02:00:c0:a8:17:fb
IP                              192.168.23.2                     192.168.23.251


LEASES
AR  OWNER                    MAC           IP         PORT_FORWARD   IP6
0   V:0        02:00:c0:a8:17:02 192.168.23.2 [9001:9100]:[1-100]     -
```

Figure 4.4.5: Sample allocation of IP and port ranges for a VM

API and Interfaces

The *nodeport* driver uses the OpenNebula network interfaces and APIs to implement the forwarding driver. The virtual network drivers are loaded from the `/var/lib/one/remotes/vnm/nodeport/` folder and they implement the actions:

```
/var/lib/one/remotes/vnm/nodeport/
├── nodeport.rb   # nodeport driver implementation
├── pre           # before the NIC attach
├── post          # after the NIC attach (installs iptables rules)
├── clean         # on VM power off or destroy (deletes iptables rules)
└── update_sg     # Security Groups update
```

Figure 4.4.6: Driver files

[SR4.5] Drivers for Host Provision

Description

A new set of provision drivers (used to communicate with edge/cloud providers) has been added to the main ONEedge distribution. A provision driver includes the functionality needed to interact with the edge/cloud provider to manage the associated resources (e.g. creating or updating them). The new drivers included are the following:

- DigitalOcean
- Google Cloud Compute Engine
- Vultr

Requirements and Specifications

Each driver is a combination of multiple components, each one designed to be integrated with the FireEdge UI. The main components of a driver are:

- **Run-time class**: implements the driver functionality. The classes are created dynamically (see sections SR3.1 and SR3.4 for more information), so the user can implement their own drivers in an easy way.

- **Terraform templates**: they are written in ERB format and are used to create the YAML file that Terraform needs to deploy the resources. There is a template for each resource, for example:

```
digitalocean
├── cluster.erb   # Common resources for the cluster
├── datastore.erb # Datastore representation
├── host.erb      # Host representation
├── network.erb   # Network representation
└── provider.erb  # Terraform provider definition
```

Figure 4.5.1: Terraform ERB files

- **Provision template**: YAML file that represents the provision, with all the information needed to deploy it such as the name, the ansible playbook to configure it, or the associated OpenNebula resources.

- **Provider template**: YAML file that represents the provider. This is the information that is needed to interact with it and the location in the world in which to deploy the servers.

Architecture and Components

DigitalOcean[4]

Interacts with Digitalocean provider to deploy virtual servers as base nodes for the provision. This type of provisions can be used to deploy application containers given their lightweight profile.

<u>Provision Architecture</u>

The servers are deployed using Terraform and each server is a virtual machine, not a physical server. It uses a VPC to isolate the networking in the cluster in combination with a firewall to

[4] https://www.digitalocean.com/

allow only the desired traffic. This can be further customized using OpenNebula security groups. Check section SR 4.4 to see the network model in detail.

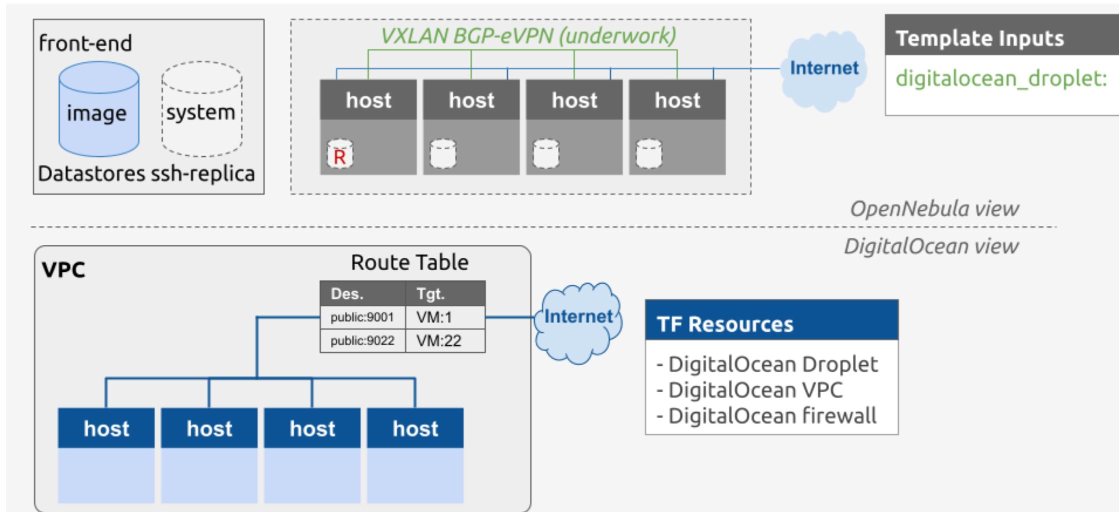


Figure 4.5.2: Digital Ocean Architecture

Terraform Templates

The following terraform objects are created:

- digitalocean_vpc

```
resource "digitalocean_vpc" "device_<%= obj['ID'] %>" {
  name     = "vpc-one-<%= obj['ID'] %>"
  region   = "<%= provision['REGION'] %>"
  <%
     net_id = obj['ID'].to_i
     id_h   = (( net_id & 3840) >> 8) + 16
     id_l   = net_id & 255
  %>
  ip_range = "172.<%= id_h %>.<%= id_l %>.0/24"
}
```

Figure 4.5.3: Digitalocean Terraform VPC

- digitalocean_droplet

```
resource "digitalocean_droplet" "device_<%= obj['ID'] %>" {
    image     = "<%= provision['IMAGE'] %>"
    name      = "<%= provision['HOSTNAME'] %>"
    region    = "<%= provision['REGION'] %>"
    size      = "<%= provision['SIZE'] %>"
    user_data = "<%= obj['user_data'] %>"
    vpc_uuid = digitalocean_vpc.device_<%= c['ID'] %>.id
}
```

Figure 4.5.4: Digitalocean Terraform droplet

- digitalocean_firewall

```
resource "digitalocean_firewall" "device_<%= obj['ID'] %>" {
    name = "vnc-device-<%= obj['ID'] %>"

    droplet_ids = [digitalocean_droplet.device_<%= obj['ID'] %>.id]
```

```
    inbound_rule {
        protocol         = "tcp"
        port_range       = "22"
        source_addresses = ["0.0.0.0/0", "::/0"]
    }

    # BGP traffic from VPC droplets. IP range MUST be consistent with cluster.erb
    inbound_rule {
        protocol         = "tcp"
        port_range       = "179"
        source_addresses = [digitalocean_vpc.device_<%= c['ID'] %>.ip_range]
    }

    # VXLAN traffic from VPC droplets. IP range MUST be consistent with cluster.erb
    inbound_rule {
        protocol         = "udp"
        port_range       = "8472"
        source_addresses = [digitalocean_vpc.device_<%= c['ID'] %>.ip_range]
    }

    # Client Ports for VMs. Port range MUST be consistent with VNET definition
    inbound_rule {
        protocol         = "tcp"
        port_range       = "9000-65535"
        source_addresses = ["0.0.0.0/0"]
    }

    outbound_rule {
        protocol              = "tcp"
        port_range            = "1-65535"
        destination_addresses = ["0.0.0.0/0", "::/0"]
    }

    outbound_rule {
        protocol              = "udp"
        port_range            = "1-65535"
        destination_addresses = ["0.0.0.0/0", "::/0"]
    }

    outbound_rule {
        protocol              = "icmp"
        destination_addresses = ["0.0.0.0/0", "::/0"]
    }
}
```

Figure 4.5.5: Digitalocean Terraform firewall

Provision Template

The following elements are created in OpenNebula:

- A cluster that contains all the resources, including the default datastores.
- LXC hosts, one host for each droplet.
- System and image datastores using replica OpenNebula method.
- Public network to access VMs running on hosts.
- Vnet template that can be instantiated to create private networks for the VMs.

Provider Template

There are five pre-created provider templates, each one for a different DigitalOcean location. A DigitalOcean template includes:

- Token: used to authenticate against DigitalOcean.

- Region: location in the world in which to deploy all the resources.

```
name: 'digitalocean-ams3'

description: 'Virtual Edge Cluster in DigitalOcean datacenter in Amsterdam (AMS3)'
provider: 'digitalocean'

connection:
  token: 'DigitalOcean token'
  region: 'ams3
```

Figure 4.5.6: Digitalocean Amsterdam provider YAML

Google Cloud Compute Engine[5]

Interacts with Google Cloud Compute Engine provider to deploy virtual servers as provision nodes.

Provision Architecture

The servers are deployed using Terraform and each server is a virtual machine, not a physical server. The schema is similar to that described for DigitalOcean above.

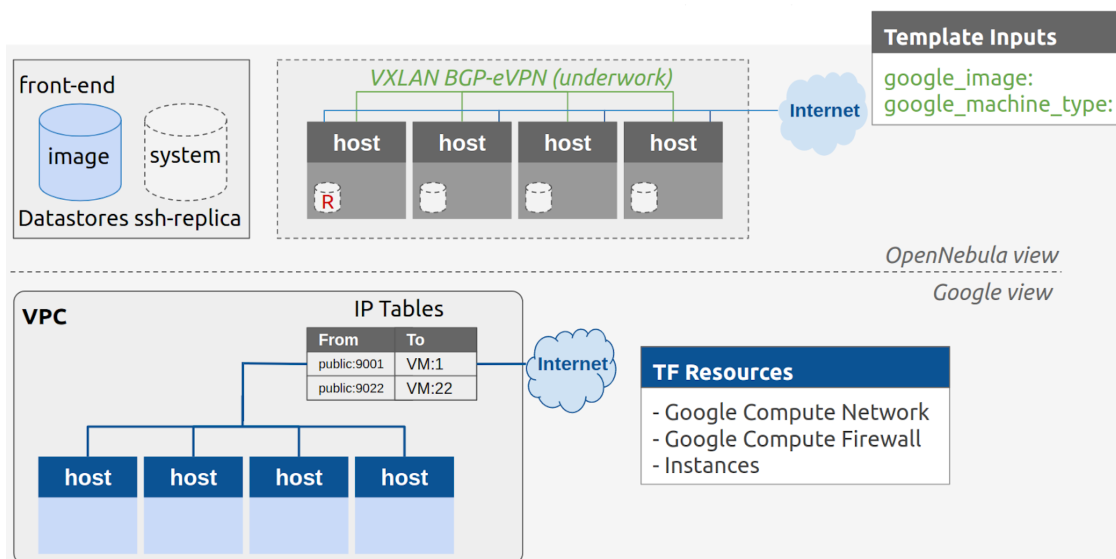


Figure 4.5.7: Google Compute Engine Architecture

Terraform Templates

The following terraform objects are created:

- google_compute_network

```
resource "google_compute_network" "device_<%= obj['ID'] %>" {
    name = "<%= obj['NAME'] %>-vpc"
}
```

Figure 4.5.8: Google Terraform network

- google_compute_instance

[5] https://cloud.google.com/compute

```
resource "google_compute_instance" "device_<%= obj['ID'] %>" {
    name         = "<%= provision['HOSTNAME'] %>"
    machine_type = "<%= provision['MACHINETYPE'] %>"

    boot_disk {
        initialize_params {
            image = "<%= provision['IMAGE'] %>"
        }
    }

    network_interface {
        network = google_compute_network.device_<%= c['ID'] %>.name

        access_config {
            // Ephemeral IP
        }
    }

    metadata = {
        ssh-keys = "<%= obj['user_data'] %>"
    }
}
```

Figure 4.5.9: Google Terraform instance

- google_compute_firewall

```
resource "google_compute_firewall" "device_<%= obj['ID'] %>" {
    name    = "<%= obj['NAME'] %>-firewall"
    network = google_compute_network.device_<%= obj['ID'] %>.name

    allow {
        protocol = "icmp"
    }

    # Client Ports for VMs. Port range MUST be consistent with VNET definition
    allow {
        protocol = "tcp"
        ports    = ["22", "179", "5900-6000", "9000-65535"]
    }

    allow {
        protocol = "udp"
        ports    = ["8472"]
    }
}
```

Figure 4.5.10: Google Terraform firewall

Provision Template

The following elements are created in OpenNebula:

- A cluster that contains all the resources, including the default datastores.
- LXC hosts.
- System and image datastores using replica OpenNebula method.
- Public network to access VMs running on hosts.
- Vnet template that can be instantiated to create private networks for the VMs.

Provider Template

There are four pre-created provider templates, each of them is for a different GCE. A GCE template includes:

- Credentials: JSON file used to authenticate against Google.

- Project: project ID where the instance is going to be deployed.
- Region and zone: both together identify the location in the world where the instance is going to be deployed.

```
name: 'google-belgium'

description: 'Virtual Edge Cluster in Google Belgium (europe-west1-b)'
provider: 'google'

connection:
  credentials: 'JSON credentials file'
  project: 'Google Cloud Platform project ID'
  region: 'europe-west1'
  zone: 'europe-west1-b'

inputs:
   - name: 'google_image'
     type: 'list'
     options:
       - 'centos-8-v20210316'
   - name: 'google_machine_type'
     type: 'list'
     options:
       - 'e2-standard-2'
       - 'e2-standard-4'
       - 'e2-standard-8'
```

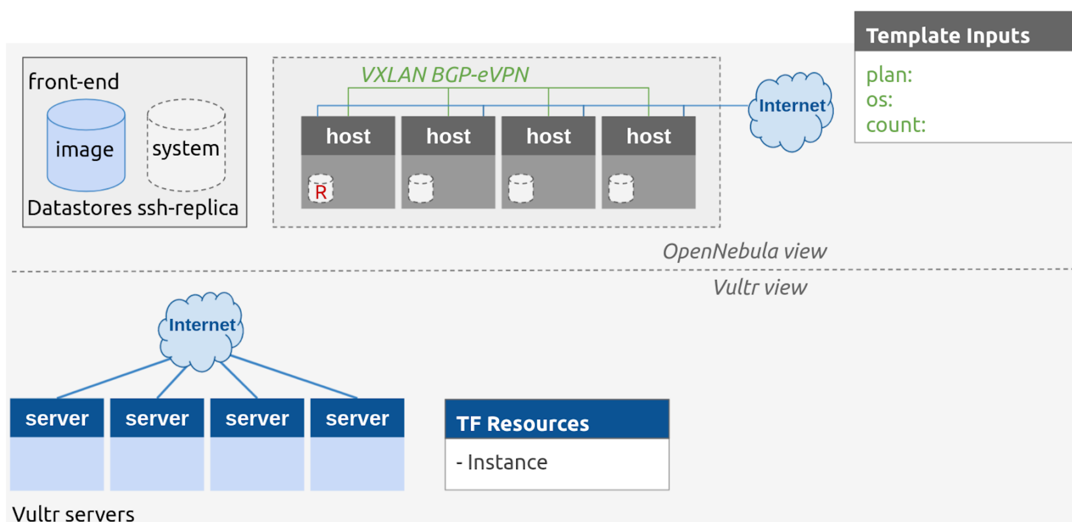
Figure 4.5.11: Google Belgium provider YAML

Vultr[6]

Interacts with Vultr provider to deploy two variants of provision hosts virtual and bare metal servers.

Provision Architecture

The servers are deployed using Terraform and each server is a virtual machine or physical server. It requests an IP address from the provider to add public connectivity to the VMs running on the hosts. This is used in combination with the OpenNebula IPAM driver to request or release the IPs from the VMs.



[6] https://www.vultr.com/

Figure 4.5.12: Google Compute Engine Architecture

Terraform Templates

The following terraform objects are created:

- vultr_bare_metal_server

```
resource "vultr_bare_metal_server" "device_<%= obj['ID'] %>" {
    hostname         = "<%= provision['HOSTNAME'] %>"
    plan             = "<%= provision['PLAN'] %>"
    region           = "<%= provision['REGION'] %>"
    os_id            = "<%= provision['OS'] %>"
    script_id        = vultr_startup_script.device_<%= obj['ID'] %>.id
    tag              = "OpenNebula - ONE_ID=<%= obj['ID'] %>"
    activation_email = false
}
```

Figure 4.5.13: Vultr Terraform server

Provision Template

The following elements are created in OpenNebula:

- A cluster that contains all the resources, including the default datastores.
- KVM or LXC hosts.
- System and image datastores using replica OpenNebula method.
- Public network to access VMs running on hosts.

Provider Template

There are four provider templates. Each of them is for a different location in the world and they contain:

- Key: used to authenticate against Vultr.
- Region: location in the world in which to deploy all the resources.

```
name: 'vultr-amsterdam'

description: 'Edge cluster in Vultr Amsterdam'
provider: 'vultr_metal'

connection:
  key: 'Vultr key'
  region: 'ams'

inputs:
   - name: 'vultr_os'
     type: 'list'
     options:
       - '362'
   - name: 'vultr_plan'
     type: 'list'
     options:
       - 'vbm-8c-132gb'
```

Figure 4.5.14: Vultr Amsterdam provider YAML

Data Model

The data is stored in the OpenNebula database using XML documents through the *oneprovision* and *oneprovider* commands. The user can use the templates described in the previous section

to create a provider using the new drivers via `oneprovider` and create a new provision via the `oneprovision` command.

Once the provision is deployed using Terraform, the state file is also stored in the database using a provision document. This state is managed internally by OpenNebula in the same way as is done by the other drivers.

API and Interfaces

No changes were made in the API nor in the CLI. The idea of this feature was to add new providers to the existing one, maintaining the data structure that was previously developed.

[SR4.9] Support on-Premises far-Edge for Resource Provisioning

Description

In order to allow the configuration of on-premises nodes, the `oneprovision` tool has been extended to be able to configure existing bare metal resources, and thus simplify and automate the process of adding new resources to the cloud. The main goals of this requirement are:

- **Standardization**: ONEedge instances only support the configurations that better fit the far-edge deployment. The use of well defined configurations will force standardization across the on-premises and edge clusters, easing both maintenance and operation.
- **Speed up**: the automation on the configuration of the on-premises cluster infrastructure allows us to speed up the provisioning of the hosts.
- **Reduce costs**: thanks to the standardization and the speed-up of the provisioning process.

Requirements and Specifications

The on-premises provisioning is fully compatible with the existing `oneprovision` tool, hence it has the same requirements defined in section *SR4.5 Drivers for Host Provision*. Also, it must be treated like any other driver in order to keep all the oneprovision's interfaces as simple as possible.

Architecture and Components

The on-premises cluster includes the following components:

- Nodes with a CentOS8 installation.
- Networking consisting of: (i) management interface to connect the nodes to the front-end; (ii) a public network; and (iii) a private network for VM interconnection using VXLAN.
- Storage area in the nodes to hold the VM disk images.

As part of the cluster, the provision drivers create the associated elements in OpenNebula including the hosts, datastores, and private and public networks.

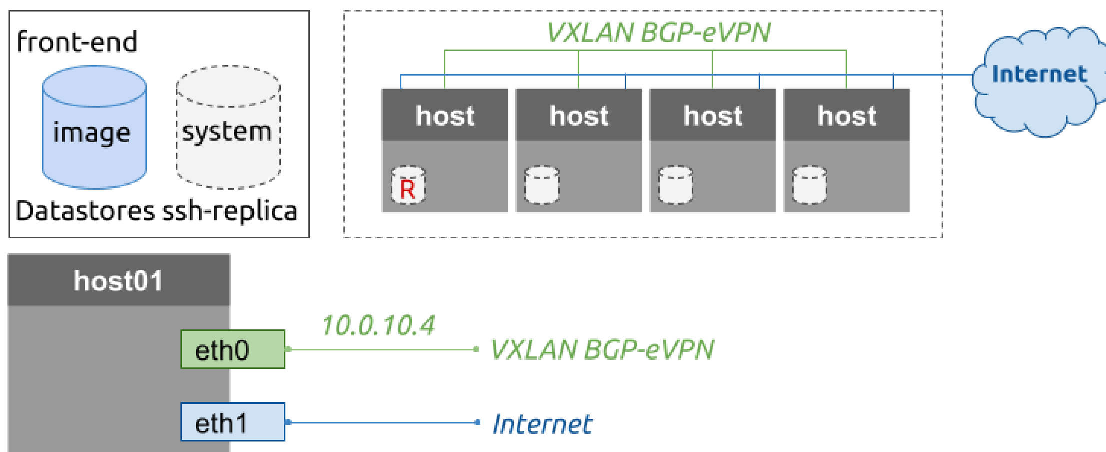


Figure 4.9.1: Architecture and components for the default on-premises cluster

The on-premises clusters are defined by:

- A cluster template that specifies the cluster elements described above.
- A set of ansible playbooks that includes the installation of the software dependencies and configures the network interfaces and the storage volumes.

It is interesting to note that these elements of the on-premises driver are customizable in order to support ad-hoc scenarios and/or install additional components that better integrate the cluster with other existing services in the data center.

Data Model

The data model is defined in Section *SR4.5 Drivers for Host Provision.* The on-premises drivers have been designed to be fully compatible with the `oneprovision` runtime and so leverage all the associated tools.

API and Interfaces

Not applicable.

[SR4.10] Support ARM for Resource Provisioning

Description

In order to be able to provision ARM resources, the building tooling was extended to support *aarch64* along with x86_64. To allow greater flexibility, we added support for cross-compilation of binary packages, and so compile ARM packages on Intel servers.

Requirements and Specifications

Not applicable.

Architecture and Components

The same package structure is preserved for ARM builds. Note that some of these packages are architecture independent and are the same for every platform. In the following table we detail the new ARM packages for RPM and deb-based distributions, as well as architecture independent packages:

deb packages (aarch64/arm64)	RPM packages (aarch64/arm64)
docker-machine-opennebula_6.2.0-2_arm64.deb	docker-machine-opennebula-6.2.0-2.el8.aarch64.rpm
opennebula_6.2.0-2_arm64.deb	opennebula-6.2.0-2.el8.aarch64.rpm
opennebula-fireedge_6.2.0-2_arm64.deb	opennebula-fireedge-6.2.0-2.el8.aarch64.rpm
opennebula-guacd_6.2.0-2_arm64.deb	opennebula-guacd-6.2.0-1.2.0+2.el8.aarch64.rpm
opennebula-node-firecracker_6.2.0-2_arm64.deb	opennebula-node-lxc-6.2.0-2.el8.aarch64.rpm
opennebula-node-lxc_6.2.0-2_arm64.deb	opennebula-rubygems-6.2.0-2.el8.aarch64.rpm
opennebula-node-lxd_6.2.0-2_arm64.deb	**RPM packages (no arch)**
opennebula-rubygems_6.2.0-2_arm64.deb	opennebula-common-6.2.0-2.el8.noarch.rpm
deb packages (all)	opennebula-common-onecfg-6.2.0-2.el8.noarch.rpm
libopennebula-java_6.2.0-2_all.deb	opennebula-flow-6.2.0-2.el8.noarch.rpm
libopennebula-java-doc_6.2.0-2_all.deb	opennebula-gate-6.2.0-2.el8.noarch.rpm
opennebula-common_6.2.0-2_all.deb	opennebula-java-6.2.0-2.el8.noarch.rpm
opennebula-common-onecfg_6.2.0-2_all.deb	opennebula-libs-6.2.0-2.el8.noarch.rpm
opennebula-flow_6.2.0-2_all.deb	opennebula-migration-6.2.0-2.el8.noarch.rpm
opennebula-gate_6.2.0-2_all.deb	opennebula-migration-community-6.2.0-2.el8.noarch.rpm
opennebula-libs_6.2.0-2_all.deb	opennebula-node-firecracker-6.2.0-2.el8.aarch64.rpm
opennebula-migration_6.2.0-2_all.deb	opennebula-node-kvm-6.2.0-2.el8.noarch.rpm
opennebula-migration-community_6.2.0-2_all.deb	opennebula-provision-6.2.0-2.el8.noarch.rpm
opennebula-node-kvm_6.2.0-2_all.deb	opennebula-provision-data-6.2.0-2.el8.noarch.rpm
opennebula-provision_6.2.0-2_all.deb	opennebula-sunstone-6.2.0-2.el8.noarch.rpm
opennebula-provision-data_6.2.0-2_all.deb	opennebula-tools-6.2.0-2.el8.noarch.rpm
opennebula-sunstone_6.2.0-2_all.deb	python3-pyone-6.2.0-2.el8.noarch.rpm

opennebula-tools_6.2.0-2_all.deb
python3-pyone_6.2.0-2_all.deb

Data Model

The number and name of packages are the same for Intel and ARM architectures.

API and Interfaces

Not applicable.

5. Edge Apps Marketplace (CPNT5)

[SR5.2] Built-in Management of Application Containers Engine

Description

In order to deploy containerized applications on cloud and/or edge resources provisioned with the tools/SRs described in Section 4, a container orchestration system and application containers engine must be provided. The main aim of this software requirement is the delivery and management of Kubernetes clusters as a service to deploy, manage, and scale application containers.

Kubernetes clusters can be created and provisioned using one of the two appliances:

- K8s appliance based on the standard Kubernetes distribution (https://kubernetes.io/)
- K3s appliance based on the lightweight K3s distribution (https://k3s.io/), which is more suitable for edge environments

In this cycle, and in order to access applications deployed on the Kubernetes clusters, the K8s appliance has been enhanced by integrating the **Load Balancer service type**, beside the already available NodePort and External IP service types.

Requirements and Specifications

Kubernetes clusters are defined as virtual appliances in OpenNebula Marketplace that are ready to run and can be instantiated in any OpenNebula environment. In order to expose applications running in Kubernetes pods, the K8s appliance is enhanced with the Load Balancer service type, based on MetalLB.

Architecture and Components

The “Kubernetes” Appliances (coming in both K8s and K3s flavors) are composed of:

- an image that already contains all the installed packages that are needed to configure and bootstrap a Kubernetes cluster
- the Virtual Machine template that allows the deployment and configuration of a single node of the cluster (both the master or the worker)
- One Flow Service template that allows the deployment and configuration of a multi-node Kubernetes cluster

The VM and the OneFlow Service templates are the same for both the Kubernetes distributions.

Single Node Deployment

In this type of deployment, the user doesn’t need to set up anything in advance. The appliance will bootstrap a fully functional single node Kubernetes cluster which can then be extended with other worker nodes at any time. In cases where the default behavior is not sufficient, the user can contextualize the appliance with few parameters that control deployment customization. New nodes can join an already running Kubernetes cluster, if the user provides the contextualization parameters that specify the master node IP address and the secret token and hash.

Multi-Node Deployment

An automatically managed multi-node Kubernetes cluster can be created by instantiating the OneFlow Service template. The Service defines two roles: the master and the worker. The user

can customize the K8s cluster deployment by providing a list of specific contextualization variables parameters.

In this cycle, and for both kinds of deployments, the appliance has been extended so the user can configure a Load Balancer service type by configuring a context parameter for the IP range (ONEAPP_K8S_LOADBALANCER_RANGE, see the Data Model section for a more detailed description of the permitted context parameters). The implementation is based on the integration of the baremetal load balancer provider, called MetalLB, that by default is configured as ARP/Layer2 LoadBalancer (i.e. the exposed LoadBalancer IP must be routed to one of the Kubernetes nodes by means outside of the scope of the appliance itself). MetalLB also supports BGP/Layer3 load balancing. If the user is capable of setting up the network for this dynamic routing protocol then the user can provide the appliance with the proper configuration via the contextualization parameter ONEAPP_K8S_LOADBALANCER_CONFIG (Base64 encoded).

Data Model

A Kubernetes cluster can be created in any OpenNebula cluster by importing the corresponding Marketplace appliance in the cluster datastores and then instantiating the VM or the OneFlow Service template, according to the chosen deployment model (single node or multi-node).

The instantiation can be customized by the user through different configuration parameters reported in Table 5.2.1.

Parameter	Default	Default
ONEAPP_K8S_ADDRESS	routable IP	Master node address or network (in CIDR format)
ONEAPP_K8S_NODENAME	hostname	Master Node Name
ONEAPP_K8S_PORT	6443	Kubernetes API port on which nodes communicate
ONEAPP_K8S_TAINTED_MASTER	no	Master node acts as control-plane only (you will need to add worker nodes)
ONEAPP_K8S_PODS_NETWORK	10.244.0.0/16	Kubernetes pod network - pods will have IP from this range
ONEAPP_K8S_ADMIN_USERNAME	admin-user	UI dashboard admin account - K8s secret's token is prefixed with this name

Table 5.2.1: Attributes for K8s controllers

New worker nodes can join the already running Kubernetes cluster if they are provided with the contextualization parameters stated in Table 5.2.2.

Parameter	Default
ONEAPP_K8S_ADDRESS	Master node IP address
ONEAPP_K8S_TOKEN	Secret token - to add worker node to the cluster

ONEAPP_K8S_HASH	Secret hash - to add worker node to the cluster

Table 5.2.2: Attributes for K8s nodes

In order to correctly parameterize the Load Balancer MetalLB available in the K8s appliance, two configuration parameters have been added for the Load Balancer service type:

- ONEAPP_K8S_LOADBALANCER_RANGE[0-9] is used to configure the LoadBalancer type of services. The value must be a range (or multiple ranges by adding more parameters with numbered suffix), or LoadBalancer IP range (can be used multiple times with numbered suffix)
- ONEAPP_K8S_LOADBALANCER_CONFIG: if the user is capable of setting up the network for this dynamic routing protocol then the user can provide the appliance with the proper configuration via the contextualization parameter

As an example, let's see how the OpenNebula contextualization parameters can be used to generate a configMap for ARP load balancing in the K8s appliance.

```
ONEAPP_K8S_LOADBALANCER_RANGE1=192.168.10.100-192.168.10.200
ONEAPP_K8S_LOADBALANCER_RANGE2=192.168.20.100-192.168.20.200
ONEAPP_K8S_LOADBALANCER_RANGE3=192.168.30.100-192.168.30.200
```

Figure 5.2.3: LoadBalancer parameters defining external IPs

Starting from the attribute values in Figure 5.2.2, the contextualization scripts made available in this cycle within the K8s appliance will render the configMap depicted in Figure 5.2.3 based on the ONEAPP_K8S_LOADBALANCER_RANGE attribute value.

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.10.100-192.168.10.200
      - 192.168.20.100-192.168.20.200
      - 192.168.30.100-192.168.30.200
```

Figure 5.2.4: ConfigMap needed to configure K8s MetalLB

[SR5.5] Edge Market GUI Developments

Description

In order to meet all the requirements to consume edge and cloud resources from the ONEedge web interface, it has become necessary to extend the current OpenNebula GUI. This has proven difficult due to the technical debt of a web application over a decade old, which in turn has motivated a rewrite of the interface from scratch using a modern web development framework such as React/Redux, taking the opportunity to bring new multi-cloud and edge functionality to the interface.

Requirements and Specifications

The new Sunstone interface—which is in Beta state in OpenNebula 6.2—aims to replace the decade-old Sunstone web interface based on Ruby Backend technologies and JQuery Frontend technologies. Requirements can be succinctly expressed as follows:

- Cover **ALL** functionality offered by the current Sunstone ruby web interface
- Use a modern web development framework that ensures future sustainability and extensibility to adapt to the dynamic edge landscape
- Backend in Node.js and Frontend in React/Redux
- Reuse FireEdge server, enable multi-app Backend delivery for OneProvision GUI and new Sunstone
- Enable the deployment of workloads over different OpenNebula clusters and edge clusters

Backend Architecture and Components

Figure 5.5.1 shows the architecture of the ONEedge web interface server.

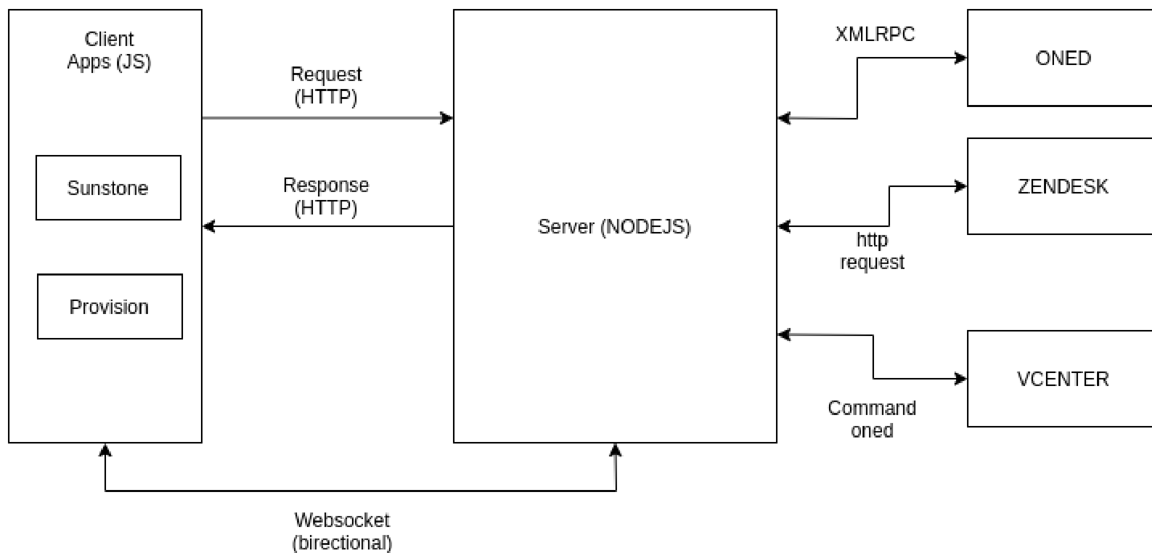


Figure 5.5.1: Example of FireEdge architecture

The server Backend is composed of multiple calls to different services (XMLRPC, HTTP request, etc.) that depend on them. The information flow is depicted in Figure 5.5.2.

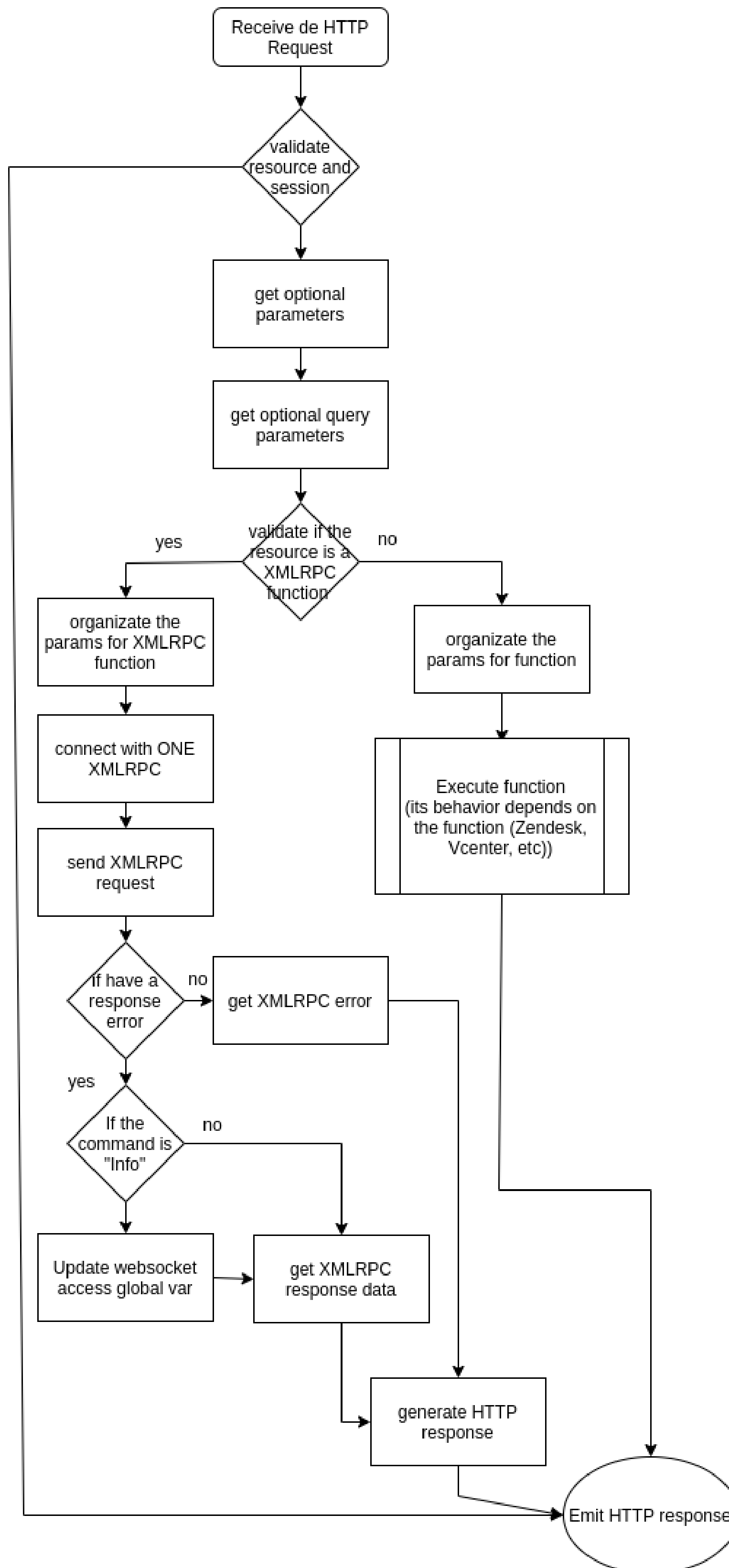


Figure 5.5.2: FireEdge information flow

The behavior changes depending on the resource given in the request: a) if it is an existing ONE XMLRPC command—i.e., it is served by OpenNebula core daemon, oned—, the logic flow

depicted in Figure 5.5.2 will be executed; b) on the other hand, if the resource is from Zendesk, vCenter, provision, Sunstone, etc., it depends on the behavior defined in the function.

XMLRPC Commands

XMLRPC commands are defined in the folder *fireedge/server/utils/constants/commands*. Inside each of the files (listed in Figure 5.5.3) there is a JSON file describing each of the XMLRPC commands implemented by the OpenNebula core.

```
fireedge/server/utils/constants/commands
├── acl.js
├── cluster.js
├── datastore.js
├── documents.js
├── groups.js
├── hooks.js
├── host.js
├── image.js
├── index.js
├── market.js
├── marketapp.js
├── secgroup.js
├── system.js
├── template.js
├── user.js
├── vdc.js
├── vm.js
├── vmgroup.js
├── vn.js
├── vntemplate.js
├── vrouter.js
└── zone.js
```

Figure 5.5.3: FireEdge command path

An example of the model followed to create a datastore resource can be found in Figure 5.5.4.

```
const DATASTORE_ALLOCATE = 'datastore.allocate'

const Actions = {
  DATASTORE_ALLOCATE
}

module.exports = {
  Actions,
  Commands: {
    [DATASTORE_ALLOCATE]: {
      // inspected
      httpMethod: POST,
      params: {
        template: {
          from: postBody,
          default: ''
        },
        cluster: {
          from: postBody,
          default: -1
        }
      }
    }
  }
}
```

Figure 5.5.4: ONE command example

Function commands

Function commands are defined within *fireedge/src/server/routes/api* location, which contains folders (listed in Figure 5.5.5) implementing all the logic needed to define the Backend routes.

```
fireedge/src/server/routes/api
├── 2fa
 |         ├── index.js
 |         ├── string-route.js
 |         ├── tfa-functions.js
 |         └── tfa.js
├── auth
 |         ├── auth-functions.js
 |         ├── auth.js
 |         ├── functions.js
 |         ├── string-routes.js
 |         └── index.js
├── files
 |         ├── files.js
 |         ├── functions.js
 |         ├── string-routes.js
 |         └── index.js
├── oneflow
 |         ├── functions.js
 |         ├── schemas.js
 |         ├── service_template-functions.js
 |         ├── service-functions.js
 |         ├── service_template.js
 |         ├── service.js
 |         ├── string-routes.js
 |         └── index.js
├── provision
 |         ├── functions.js
 |         ├── provider-functions.js
 |         ├── provider.js
 |         ├── provision_template-functions.js
 |         ├── provision_template.js
 |         ├── provision-functions.js
 |         ├── provision.js
 |         ├── schemas.js
 |         ├── string-routes.js
 |         └── index.js
├── sunstone
 |         ├── sunstone.js
 |         ├── string-route.js
 |         ├── sunstone-functions.js
 |         └── index.js
├── support
 |         └── index.js
├── vcenter
 |         ├── command-flags.js
 |         ├── functions.js
 |         ├── string-routes.js
 |         ├── vcenter.js
 |         └── index.js
├── zendesk
 |         ├── zendesk-functions.js
 |         ├── zendesk.js
 |         ├── string-routes.js
 |         ├── functions.js
 |         └── index.js
├── vm
 |         ├── functions.js
 |         ├── string-routes.js
 |         ├── vm.js
```

Figure 5.5.5: FireEdge functions command path

The structure of each of the folders is as follows:

- **Index.js**: joins the logic of each of the routes. It must return an object with the functions that will be public (they do not require user validation) as well as the private ones. An example is given in Figure 5.5.6.

```
Const { addFunctionAsRoute, setFunctionRoute } = require('server/utils/server')
const { routes: vmRoutes } = require('./vm')
const { VM } = require('./string-routes')

const privateRoutes = []
const publicRoutes = []

/**
 * Set private routes.
 *
 * @param {object} routes - object of routes
 * @param {string} path - principal route
 * @param {Function} action - function of route
 */
const setPrivateRoutes = (routes = {}, path = '', action = () => undefined) => {
  if (Object.keys(routes).length > 0 && routes.constructor === Object) {
    Object.keys(routes).forEach((route) => {
      privateRoutes.push(
        setFunctionRoute(route, path,
          (req, res, next, connection, userId, user) => {
            action(req, res, next, routes[route], user, connection)
          }
        )
      )
    })
  }
}

/**
 * Add routes.
 *
 * @returns {Array} routes
 */
const generatePrivateRoutes = () => {
  setPrivateRoutes(vmRoutes, VM, addFunctionAsRoute)
  return privateRoutes
}

const functionRoutes = {
  private: generatePrivateRoutes(),
  public: publicRoutes
}

module.exports = functionRoutes
```

Figure 5.5.6: index.js example

- **[command].js**: Defines the URL and the parameters of the function. As per the example in Figure 5.5.7, the command to perform a Save As Template over a running VM can be requested through the following URL: http://<FIREEDGE_URL>:2616/fireedge/api/vm/save/<ID>

```
const { httpMethod, from: fromData } = require('server/utils/constants/defaults')
const { saveAsTemplate } = require('./functions')
const { POST } = httpMethod

const routes = {
  [POST]: {
    save: {
      action: saveAsTemplate,
      params: {
        id: {
          from: fromData.resource,
          name: 'id'
        },
        name: {
          from: fromData.postBody,
          name: 'name'
        }
      }
    }
  }
}

const authApi = {
  routes
}
module.exports = authApi
```

Figure 5.5.7: [COMMAND].js example

- **String-routes.js**: defines the url of a particular resource (Figure 5.5.8 defines a route to access a VM template contents through http: //<FIREEDGE_URL>:2616/fireedge/api/vm/save/<ID>)

```
const VM = 'vm'

const Actions = {
 VM
}

module.exports = Actions
```

Figure 5.5.8: string-routes.js example

- **Functions.js:** defines each of the functions of each route, which receives the following parameters (Save As Template function example is given in Figure 5.5.9):
  - Res: the HTTP request
  - Next: the express stepper function
  - Params: It is an object with the parameters required for the function; they are defined in the file [COMMAND].js
  - User: it is the user data

```
const saveAsTemplate = (res = {}, next = defaultEmptyFunction, params = {}, userData
= {}) => {
 let rtn = httpBadRequest
 if (params && params.id && params.name) {
   const paramsCommand = ['save', `${params.id}`, `${params.name}`]

   const executedCommand = executeCommand(defaultCommandVM, paramsCommand,
prependCommand)

   const response = executedCommand.success ? ok : internalServerError
   let message = ''
   if (executedCommand.data) {
```

```
    message = executedCommand.data.replace(regexpSplitLine, '')
  }
  rtn = httpResponse(response, message)
 }
 res.locals.httpCode = rtn
 next()
}

const functionRoutes = {
 saveAsTemplate
}
module.exports = functionRoutes
```

Figure 5.5.9: functions.js example

Authentication

User authentication is done via XMLRPC using the OpenNebula authorization module. If the username and password matches with the serveradmin data, the user's request will be granted, the session data will be saved in a global variable (cache-nodejs), and a JWT (JSON Web Token) will be generated that must be sent in each call that requires authentication.

For the creation of the JWT, the data of the user ID, User, and ONE's token are used, and are protected by a key that is generated randomly when the FireEdge is executed for the first time. This key can be found in */var/lib/one/.one/fireedge_key*. The JWT lifetime can be configured in the fireedge_server.conf configuration file.

Configuration files

FireEdge Backend is made up of multiple calls to different services. These are divided into applications (currently OneProvision and Sunstone), each one managing its own configuration file. The general structure of FireEdge configuration files as modified in this cycle can be found in Figure 5.5.10.

```
/etc/one
├── fireedge-server.conf
├── provision
 |         ├── provider.d
 |         └── provision-server.conf
└── sunstone
          ├── admin
          ├── user
          ├── sunstone-server.conf
          └── sunstone-views.yaml
```

Figure 5.5.10: FireEdge configuration path

Specific configuration for the Sunstone Beta appliance served by FireEdge is listed in Figure 5.5.11.

```
# Prepend for oneprovision command
vcenter_prepend_command: ''

# Prepend for sunstone commands
sunstone_prepend: ''

# Support
support_url: ''
support_token: ''
```

```
# this display button and clock icon in table of vm
leases:
 suspend:
   time: "+1209600"
   color: "#000000"
   warning:
     time: "-86400"
     color: "#085aef"
 terminate:
   time: "+1209600"
   color: "#e1ef08"
   warning:
     time: "-86400"
     color: "#ef2808"
```

Figure 5.5.11: Sunstone-server.conf example

Frontend Architecture and Components

An important part of managing OpenNebula through an interface is the use of forms and lists of resources. For this reason, we decided to extract some of this logic in configuration files. We differentiate between the view files located in */etc/one/fireedge/sunstone/<view_name>* and the master file located in */etc/one/fireedge/sunstone/sunstone-view.yaml* that orchestrates the views according to the primary group the user belongs to.

These view files, with yaml extension, describe the behavior of each of the resources within the application. Each file will contain a series of sections within it:

- Name of the resource
- Actions available regarding the resources
- List of criteria to filter the resource list
- Information tabs available to show the detailed information of a resource
- Sections that will be shown in a dialog

An example of the definition of a particular resource can be found in Figure 5.5.12.

```
resource_name: “VM_TEMPLATE”

actions:
  create_dialog: true
  delete: false

filters:
  label: true

info-tabs:
  info:
    enabled: true
    information_panel:
      enabled: true
      actions:
        rename: true
    permission_panel:
     enabled: true
      actions:
       chmod: false

dialogs:
  create_dialog:
    information: true
```

```
    capacity: true
    vcenter:
      enabled: true
      not_on:
        - kvm
        - lxc
        - firecracker
    network: true
    storage: false
```

Figure 5.5.12: Example of VM Template view yaml

Using the view files as a starting point, the interface generates the available routes and defines them in a menu.

To draw the resource lists, we have developed a component (*fireedge/src/client/components/Tables/Enhanced*) that allows us to define their context: actions over one or a group of elements, filtering, sorting, paging. Unlike the current, ruby-based Sunstone, it's the behavior of requests in parallel which allows the use of the interface with greater flexibility and fluidity.

Actions on resources permitted to users can proceed to a common form or a stepper form. Forms will be located in *fireedge/src/client/components/Forms*.

Both forms can be created with the functions located in *fireedge/src/client/utils/schema.js*, *createForm* and *createSteps*. For the creation of these forms we use a schema builder to parse and validate the values. An example can be found in figure 5.5.13.

```
const NAME = {
  name: 'NAME',
  label: 'New Image name',
  type: 'text'
  tooltip: 'Name for the new Image when the disk will be saved.'
}

const FIELDS = [NAME]
const SCHEMA = object({ NAME: string().trim().required() })

const SaveAsDiskForm = createForm(SCHEMA, FIELDS)
```

Figure 5.5.13: Example of common web form used in Sunstone

Data Model

Figure 5.5.14 shows how Sunstone renders a list of resources. FireEdge server checks that the user from Sunstone is authenticated, then sends the request to OpenNebula. The information that OpenNebula returns is in XML format, so we need to transform it into JSON to facilitate the manipulation from the Frontend.

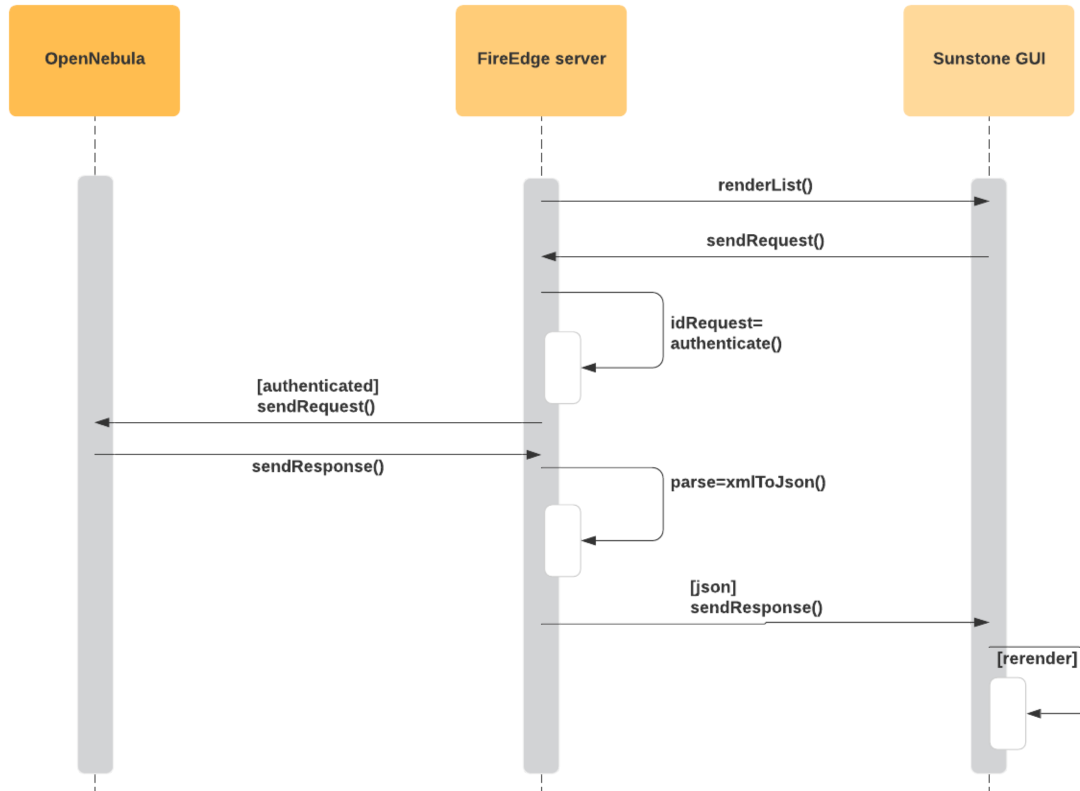


Figure 5.5.14: Sequence diagram of Sunstone GUI render process

Queries to get the pool resource from OpenNebula are greatly optimized, which ensures a swift response of the interface. If a large amount of certain types of resources are present (for example VMs or Hosts), a performance strategy that consists of making queries with intervals is implemented. Thus, the representation of the first interval list of resources is faster and the rest of the queries are kept in the background.

API and Interfaces

Sunstone routes exposed by the FireEdge Backend are divided into two categories:

- OpenNebula commands, described in the official documentation
- Functions, which are routes whose result depends on the function they execute. See Figure 5.5.15 for a list of available functions in specific FireEdge routes. Only Sunstone-relevant routes are shown (i.e., no OneProvision routes are shown)

Auth		
Method	**Route**	**Description**
POST	`/fireedge/api/auth`	Authenticate User
TFA		
POST	`/fireedge/api/tfa`	Configure 2fa to user
GET	`/fireedge/api/tfa`	Get QR resource

DEL	/fireedge/api/tfa	Delete 2fa to user
File		
POST	/fireedge/api/files	Upload file
GET	/fireedge/api/files/list/<ID>	Get file uploaded (the ID is optional)
PUT	/fireedge/api/files/update/<ID>	Update file
DEL	/fireedge/api/files/delete/<ID>	Delete file
OneFlow		
GET	/fireedge/api/service_template/list/<ID>	Get the service template (the ID is optional)
POST	/fireedge/api/service_template/create	Create a service template
POST	/fireedge/api/service_template/action/<ID>	Add action
PUT	/fireedge/api/service_template/update/<ID>	Update service template
DELETE	/fireedge/api/service_template/delete/<ID>	Delete Service template
GET	/fireedge/api/service/list/<ID>	Get the service (the ID is optional)
POST	/fireedge/api/service/action/<ID>	Add action to service
POST	/fireedge/api/service/scale/<ID>	Add scale to service
POST	/fireedge/api/service/role-action/<ROLE_ID>/ID	Add role to service
POST	/fireedge/api/service/sched_action/<ID>	Add Schedule action to service
PUT	/fireedge/api/service/sched_action/<ID>/<ID_SCHED_ACTION>	Update Schedule action to service
DELETE	/fireedge/api/service/delete/<ID>	Delete service
DELETE	/fireedge/api/service/sched_action/<ID>/<ID_SCHED_ACTION>	Delete Schedule action to service
Sunstone		
GET	/fireedge/api/sunstone/views	Get sunstone views
GET	/fireedge/api/sunstone/config	Get sunstone config
vCenter		
POST	/fireedge/api/vcenter/import/<vObject>	Import vObject
POST	/fireedge/api/vcenter/cleartags/<ID>	Clear tags
POST	/fireedge/api/vcenter/hosts/<vCenter>	Import hosts
GET	/fireedge/api/vcenter/list/<ID>	Get list vCenter
GET	/fireedge/api/vcenter/listall/<ID>	Get list_all vCenter

VM		
POST	/fireedge/api/vm/save/<ID>	Save VM to VM template
Zendesk		
POST	/fireedge/api/zendesk/login	Login to zendesk
POST	fireedge/api/zendesk/create	Create ticket
PUT	fireedge/api/zendesk/update	Update ticket
GET	fireedge/api/zendesk/list/<ID>	Get tickets (the ID is optional)
GET	fireedge/api/zendesk/comments/<ID>	Get comments of ticket

Table 5.5.15: FireEdge function routes