Technical company internship

Internship at Cisco Systems

Report

Development of an ACI device package for nginx as a load-balancer

---

Fabrice Servais - s111093

Master of science in computer science and engineering

Academic year 2015 - 2016

Industrial sponsor:
Hugues De Pra

Jury:
Bernard Boigelot
Guy Leduc
Laurent Mathy

January 8, 2016

Acknowledgements

I consider that having an internship at Cisco is a great honour and is a huge chance for learning as well as for professional and personal development. I am grateful to have had the opportunity to be a part of it and also for the wonderful experience it has been, full of talented and motivated people.

I would first like to thank *Pr. Guy Leduc*, the first gearwheel of this machinery, for his efforts that gave me the possibility to do an internship at Cisco.

I express also my deepest thanks to *Hugues De Pra*, Manager of the Data Center sales team at Cisco Belux, also my industrial sponsor, for the preparation of my internship, for being concerned about it, as well as keeping me always motivated.

A special thanks is also needed for the whole Data Center team for their welcoming and for considering me as a part of the team and never as an intern. Amongst them, *Jan Van Den Broeck*, Systems Engineer at Cisco, also my coach, has my deepest gratitude for his support and help, week by week, even when he was extremely busy and, when something went wrong, for putting me in its priorities.

I would like to acknowledge the huge contribution of *Borys Berlog*, Customer Support Engineer, for all the times when I could bother him with my questions on ACI and for his technical answers that were irrefutably vital. I thank also *Juan Lage*, Principal Engineer at the Insieme Business Unit, Cisco, obviously for the great subject of intership that he gave me, but also for its help and for reframing the scope of my internship, as well as the work I did.

Last but not least, I am grateful to the Global Virtual Systems Engineer team in Belgium (*Pieter Lewyllie* and *Michiel Vanthoor*, Virtual Systems Engineers, as well as *Cedric Distelmans* and *Lionel Hercot*, Associate Systems Engineers) for their huge kindness, for integrating me into their team and work life, and for the role they played in my understanding of Cisco.

Contents

Part I

Cisco - General presentation

Chapter 1

What is Cisco ?

Cisco is an IT company founded in 1984 in San Francisco and now based in San Jose, which sells networking-related products and services. It was first known for the first multiple-protocol routers and it sells now a wide range of solutions in data transmission: Routing and Switching, Security, Storage, Enterprise Networking, Datacenter, VoIP, Telepresence, and others.

Figure 1.1: Cisco's logo

To give some numbers, Cisco counts more than 71500 employees, around 35% work in engineering, 26% in sales and 20% in services. The company has also a revenue of more than 49 billions of US$ and around 13% of the revenue is spent in R&D [4].

The company is organized in several departments :

- Sales/Pre-sales: This important part of the company is assigned with selling the solutions. Systems engineers help the vendors for the technical parts by answering the questions, designing a solution, and so on. They are divided by type of solution: data center, enterprise networking, collaboration, service provider, security, etc.
- Engineering: The products are designed and created in this department. They are organized in Business Units that take care of one kind of solution.
- Post-sales: The post-sales department is also known as TAC (Technical Assistance Center), they provide help to customers by solving their problems and potentially warning the BU of the found bug. Cisco Belgium has one of the few biggest Cisco TAC lab in the world where the problems are recreated in an environment as close as possible to the real one.
- R&D: Research and Development part, the only one of Cisco is located at Kortrijk. They work in the video and image processing field.
- Services: Products and systems made available for Cisco employees internally.
- Other functions (business, support,...).

Part II

Objective

Chapter 2

Mission

The mission was to :

Develop and deploy an ACI device package to insert NGINX *for implementing a Load Balancing as a Service solution.*

In order to better understand the objective, I will briefly explain each part of this mission. Before that, an important note is that the work aims at being a proof-of-concept. Also, the device package is an open-source project hosted on my personal Github account:

- https://github.com/FServais/NGINX-Device-Package
- https://github.com/FServais/NGINX-Agent

2.1 ACI description

Some parts of this chapter are taken or inspired from [7] and [3].

2.1.1 Introduction

ACI is the Cisco' SDN solution, it stands for "**Application Centric Infrastructure**". In this context, an **application** is a set of networking components that provides connectivity between some workloads. These relationships are represented with **Application Network Profiles (ANPs)**.

2.1.2 Physical topology

On the physical side, a fabric consists of components operating both as switches and routers, these are for now the Cisco Nexus 9k series. Those switches can operate in 2 modes: normal switching mode or ACI mode. These components are structured in a spine-leaf topology, as in FIGURE 2.1. All the devices (e.g. servers, load-balancer, firewall, controllers,...) are connected on any leaf of the topology, but nothing is connected to the spines (except the leaves). The controller is called the **APIC (Application Policy Infrastructure Controller)**.

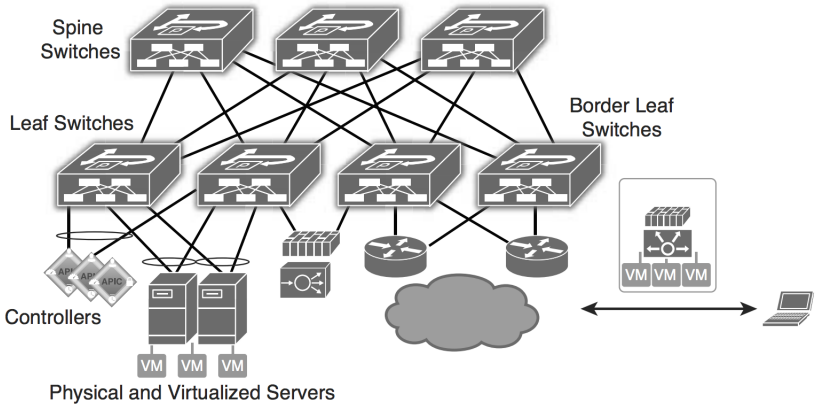

Figure 2.1: ACI fabric structure

2.1.3 Logical networking

In the logical view, a **tenant** is a logical container for application policies. It can contain Layer 3 networks represented as **context**. Each network can contain **bridge domains** for Layer 2 connectivity. It is similar to VLAN without being tied to a VLAN ID. Furthermore, each bridge domain can have several **subnets**, Layer 3 addresses. These terms are illustrated in Figure 2.2.

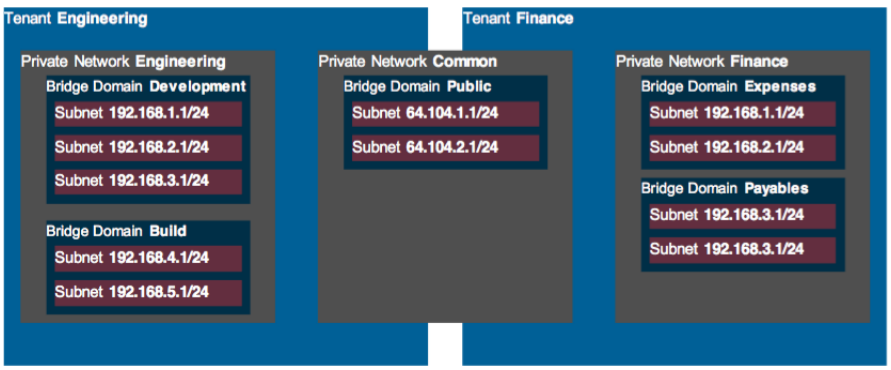

Figure 2.2: Logical networking

2.1.4 Application Network Profile

An **EPG (Endpoint Group)** is a collection of similar **endpoints**, which are identifiers to represent application components: NIC, vNIC,... These EPGs are linked together using **contracts** that defines the connectivity between them, it is the policy. The communication between 2 EPGs follows the white-list model: everything is denied except what is allowed. Contracts contains **filters** that define what is allowed, similarly to ACL. It's important to note that a policy is not only a set of ACLs but also have QoS, marking rules, redirection rules and, as I will explain later, Layers 4-7 service graphs. With the concept of contract comes the concepts of **provider** and **consumer**. Indeed, a contract is both *provided* (an EPG provides the services defined in a contract) and *consumed* (an EPG uses the services defined in a contract). An example of ANP is depicted on Figure 2.3.

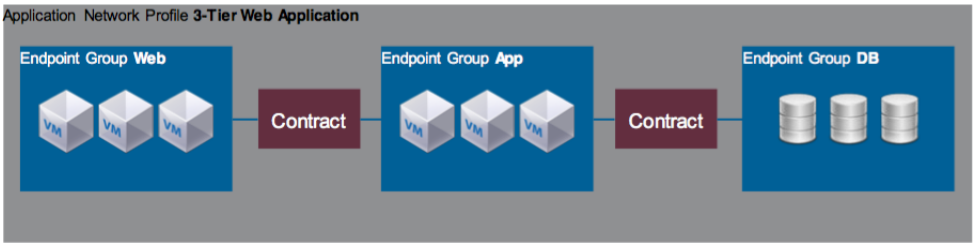

Figure 2.3: Application Network Profile

2.1.5 Service insertion

ACI has the capability to add Layer 4 through Layer 7 devices in the path between endpoints. EPGs are connected through a **service graph**, it is a variation of a contract. It defines a set of services (named **functions**) needed by an application between 2 EPGs. Those services are automatically provisioned based on the application's requirements. Actually, the user creates a graph containing **abstract nodes** that are metadevices. The APIC will translate it into a sequence of concrete devices.

To let the controller know about such function, one needs to import a **device package** containing the information about the function and the device to insert. The details concerning device package will be explained later. A scheme representing a service graph is shown in FIGURE 2.4.

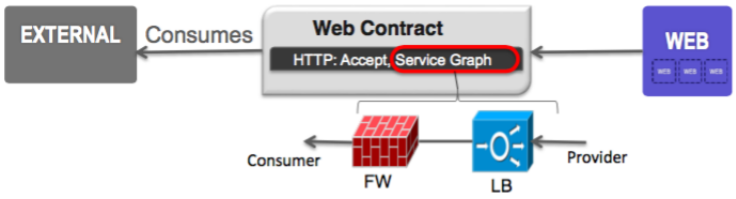

Figure 2.4: Service insertion

2.2 NGINX

Figure 2.5: NGINX's logo

NGINX is well known in the web server field for being a HA (High-Availability) web server that excel at handling a huge number of concurrent connections. With Apache, they are responsible for serving over 50% of the Internet traffic [1].

NGINX can also serve as a load-balancer [8], one just has to configure it as such, as I will explain it later. The scope of the internship includes the free version of *nginx* but not the paid one (NGINX Plus).

Part III

Methodology

Chapter 3

Introduction

The methodology of developing a device package, its composition and others are explained in the documentation [2].

A device package is a `zip` file containing:

- `DeviceModel.xml`: It is the device specification, it provides a hierarchical description of the device and the function(s) provided.
- `DeviceScript.py`: It defines the APIs that the controller will call to interact with the device. It converts call from a generic API to device-related calls.
- Image of the vendor.
- Additional files.

The Figure 3.1 shows the relationship between the device package and the controller.

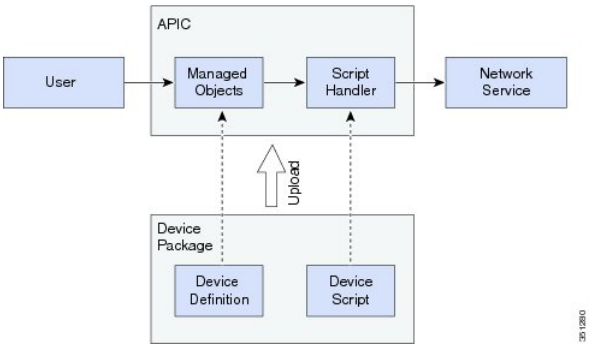

Figure 3.1: Relationship between the device package and the controller.

Chapter 4

Device specification

The device specification is represented in a XML format where each node contains a parameter or a set of parameters. Some of them are specific and used "as such" by the APIC, whereas some of them are generic and are used to represent the function or device. It is more specifically composed of:

- Generic part, containing all the generic information,
- Cluster and device configuration part, which defines the cluster(s) and device(s) (`vnsClusterCfg`, `vnsDevCfg`),
- Functional part, which defines the service function,
- Function profiles (`vnsAbsFuncProf`).

4.1 Generic part

The generic part includes:

- Information about the device package (`vnsDevScript`),
- Interfaces (`vnsDevProf`, `vnsMIfLbl`),
- Credentials (`vnsMCred`),
- Parameter validation (`vnsComparison`, `vnsComposite`),
- Errors with their description (`vnsMDfcts`),

4.2 Functional part

The parameters can be defined in one of the following categories:

- Function (`vnsMFunc`): identify a function on a device. It has to contain connector objects linked to interfaces. In this category, the configurations are links (a.k.a. "relations") to configurations defined in the other categories.
- Group (`vnsGrpCfg`): the parameters and folders can be shared across different functions in a single graph. It is not used in the case of the NGINX device package.
- Global function device (`vnsMDevCfg`): the parameters and folders can be shared across different functions across multiple graphs.

An example of how `vnsMFunc` refers to `vnsMDevCfg` is shown in FIGURE 4.1.

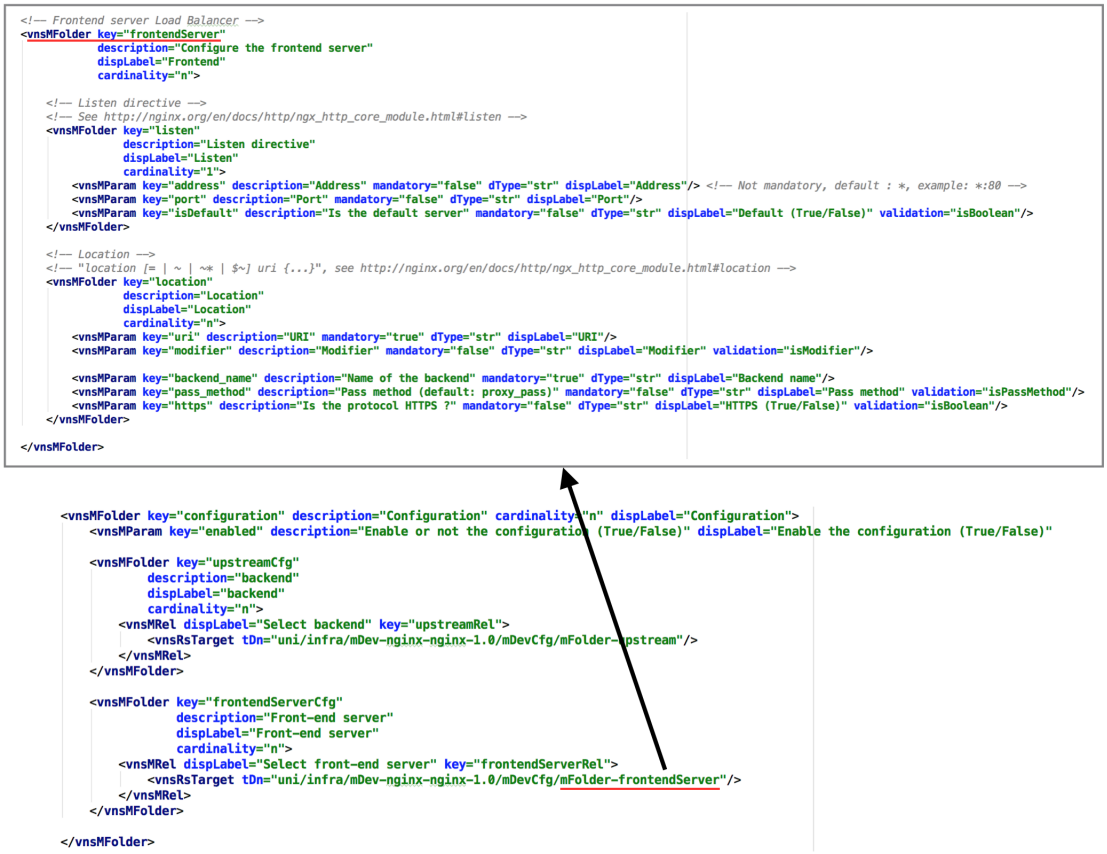

Figure 4.1: Example of linkage between Function and Global function device

4.3 Parameters

The parameters are represented by `vnsMParam`, they represent a configuration object related to the service (e.g. the name of a server, its address, a port, ...).

Parameters can be grouped in a folder represented by `vnsMFolder`. A folder can also contain other folders.

Each `vnsMParam` and `vnsMFolder` are identified by a unique `key` and contains a `description` that is displayed on the APIC GUI when more information are needed. Other attributes are not mandatory, amongst which:

- `dType`: data type of the parameter (string, integer, real),
- `validation`: id of the validation expression (see 'Validation expression'),
- `cardinality`: number of occurrences of the parameter.

4.4 Validation expression

One can define parameter validation objects used to add constraint to the parameter value or format. Such a constraint is represented by `vnsComparison`, using operations such as: equal, less (or greater) than (or equal to), or using a regular expression.

Several comparators can be grouped into `vnsComposite`. They are aggregated following one of the policy: all match, any match or exactly one match.

4.5 Faults

One can define faults that can occur. They are composed of:

- `code`: identifier,
- `descr`: description of the error,
- `recAct`: (potential) solution,
- `vnsRsDfctToCat`: link to the severity of the fault (warning, minor, major, critical)

They will be usable later in the script using their `code`.

4.6 Function profile

A function profile is a collection of pre-set values for the functional part, they basically give a value to the `vnsMParam` and `vmsMFolder` nodes (via `vnsAbsParam` and `vnsAbsFolder`), previously specified in section 4.2.

A diagram representing the different nodes in a device model is shown in appendix, on Figure A.3.

Chapter 5

Device script

The purpose of the device script is to be the intermediary between the controller and the device, as shown in FIGURE 5.1.

Figure 5.1: Device script model

5.1 Device script APIs

The functions called are classified into 4 categories:

- Device
- Cluster
- Service
- Endpoint and network event

5.1.1 Device APIs

- `def deviceValidate( device, version )`: validates whether the device version, is compatible with the APIC.
- `def deviceModify( device, interfaces, configuration)`: called when the configuration of the device is changed.
- `def deviceAudit( device, interfaces, configuration )`: clears global device configuration.
- `def deviceHealth( device, interfaces, configuration )`: requests health from the device.
- `def deviceCounters( device, interfaces, configuration )`: queries packet counter.

5.1.2 Cluster APIs

- `def clusterModify( device, interfaces, configuration )`: called when the configuration of the cluster is changed.
- `def clusterAudit( device, interfaces, configuration )`: called when the first device is added, it also clears the configuration.

5.1.3 Service APIs

- `def serviceModify( device, configuration )`: called when the configuration of the service is changed.
- `def serviceAudit( device, configuration )`: called to push the configuration, makes sure that the functionality on the device is in sync with the APIC.
- `def serviceHealth( device, configuration )`: requests health from the service.
- `def serviceCounters( device, configuration )`: queries packet counter.

5.1.4 Endpoint and network event APIs

- `def attachEndpoint(device, configuration, endpoint)`: called when an endpoint is added to the EPG.
- `def detachEndpoint(device, configuration, endpoint)`: called when an endpoint is removed from the EPG.
- `def attachNetwork(device, configuration, network)`: called when a subnet is added.
- `def detachNetwork(device, configuration, network)`: called when a subnet is removed.

Those 4 APIs can be managed via `serviceModify` and no more via these functions with a recent version of the controller.

5.1.5 Configuration parameter

All the parameters of the APIs are Python dictionaries, only `configuration` has a particular format, shown in Listing 5.1.

Listing 5.1: Configuration dictionary format

```
{(type, key, name) : {
                        'state': ...,
                        'transaction': ...,
                        'connector': ...,
                        'value': ...,
                        'target': ...,
                        'device': ...,
                        ...
                        }}
```

- `type`: type of the object represented (parameter, folder, relation,...).
- `key`: key of the attribute, defined in the device specification.
- `name`: parameter or folder instance name.
- `value`: value of the attribute, can contain other configuration parameters.

Listing 5.2: Sample configuration dictionary format

```
{(0, '', 4099): {'ackedstate': 0,
                 'ctxName': 'VRF_NG',
                 'dn': u'uni/vDev-...',
                 'state': 1,
                 'tenant': 'NGINX',
```

```
                'transaction': 0,
                'txid': 10000,
                'value': {(1, '', 39100): {'absGraph': 'NginxLB_SGT',
                                           'ackedstate': 0,
                                           'rn': ...,
                                           'state': 1,
                                           'transaction': 0,
                                           'value': {...}
```

5.1.6 Return value

The format of the return value is shown in Listing 5.3.

Listing 5.3: API return value

```
{
        'state': ...,
        'health': [],
        'fault': []
}
```

- `state`:
  - `SUCCESS=0`
  - `TRANSIENT=1`: does not require immediate action from the user, could be temporary.
  - `PERMANENT=2`: require action from user.
  - `AUDIT=3`: request an audit.
- `health`: list of tuples `(path, health score)`, where `path` is a list of tuples `(type, key, name)` referring to a specific service function within the device.
- `fault`: list of tuples `(object-path, code, fault string)`, where the `object-path` identifies the object that caused the fault, `code` is the code of the fault defined in the model and `fault string` allows to add some more information about the fault.

Chapter 6

NGINX device package

6.1 Device model

The parameters in the functional part are the ones needed to be able to have a NGINX configuration file. Let's first review how a NGINX configuration is built and then how it is applied in a device specification.

6.1.1 NGINX configuration format

The documentation related to NGINX *configuration can be found on the official website [6].*

As an explanation of the format of a NGINX configuration format, let's first take a look at LISTING 6.1, which is a very basic load-balancing configuration. It was used as reference regarding to the format used to make NGINX work as a load-balancer.

Listing 6.1: Simple load-balancing configuration file

```
server {
    listen 10.9.217.1:80;
    location / {
        proxy_pass http://backend;
    }
}
upstream backend {
    server 10.9.218.1:80;
    server 10.9.218.2:80;
}
```

A configuration is composed of 2 main elements: the **block** and the **directive**. The block is composed of a name, can have parameters and delimits a portion of configuration using curly brackets, as `upstream` in LISTING 6.1. As for the directive, it is also composed of a name and parameters, but is written on a single line, ends by a semicolon and do not delimit a portion of configuration, as `listen` in LISTING 6.1.

The `upstream` block defines a group of servers. It has here the name `backend` and contains 2 servers defined by the `server` directives: 10.9.218.1:80 and 10.9.218.2:80.

The `server` block defines the load-balancer, accepting requests from 10.9.217.1:80 (`listen` directive) and processing it depending on the URI. Here, all requests are going to the same server pool (i.e. `backend`) using the `proxy_pass` directive.

6.1.2 Corresponding device description

FIGURE 6.1 shows a diagram representation of the XML. The NGINX function can contain several `configurations`, each of them will be, at the end, exported into a separate file. Each configuration contains a link to one or several frontend (i.e. `frontendServer`, defining the `server` block) and backend (i.e. `upstream`, defining the `upstream` block).

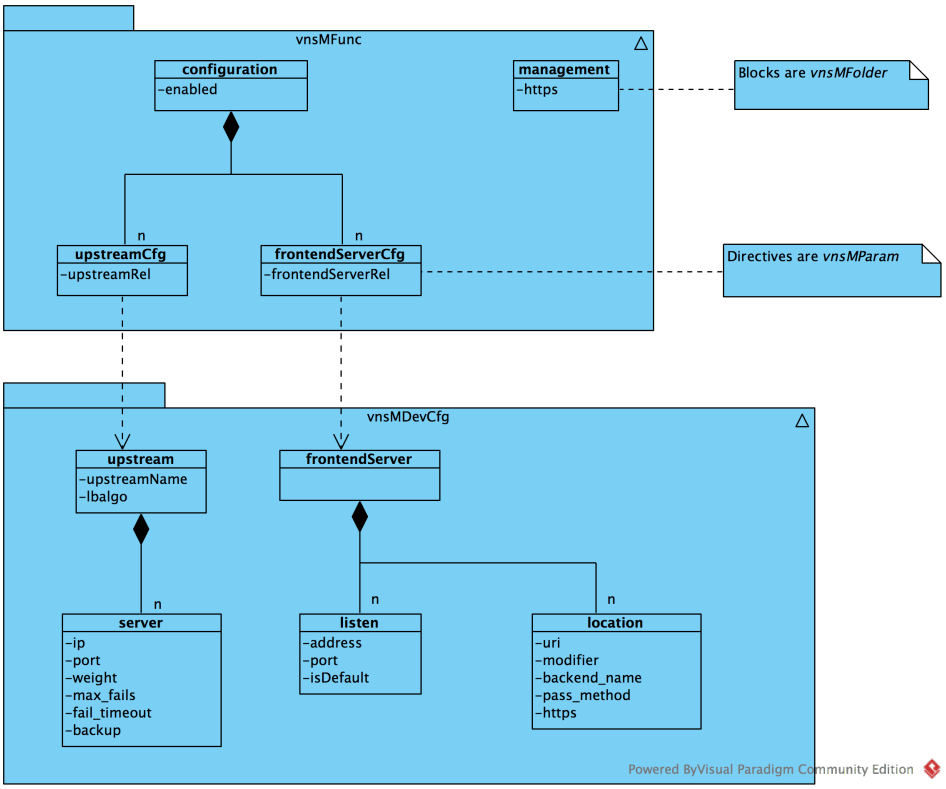

Figure 6.1: Diagram of the device model

`vnsMDevCfg` contains those different configurations. Each `upstream` has a name and a load-balancing algorithm can be specified. It also contains servers defined by an address and a port, to which parameters can also be set. A `frontendServer` has also 2 other blocks: `listen`, defined by an address and a port[1], as for the `location`, it has the URI to match the request, how to match them (`modifier`), the name of the backend to send traffic to, how to send it and if it has to be sent via HTTPS.

6.2 Device script

We will first review the simple use case where `serviceModify` or `serviceAudit` is called to push a configuration given in argument, we'll after see the details of each action.

The flow of actions when `serviceModify` or `serviceAudit` is called is shown in FIGURE A.1 and FIGURE A.2 in appendix.

[1]The `listen` directive in the NGINX configuration can be set to default.

6.2.1 API parameter

The first action is to encapsulate the received parameters from the APIs. The classes are shown in FIGURE 6.2. These classes do not have such an added value, however, they allow the parameters to be more easily handled.

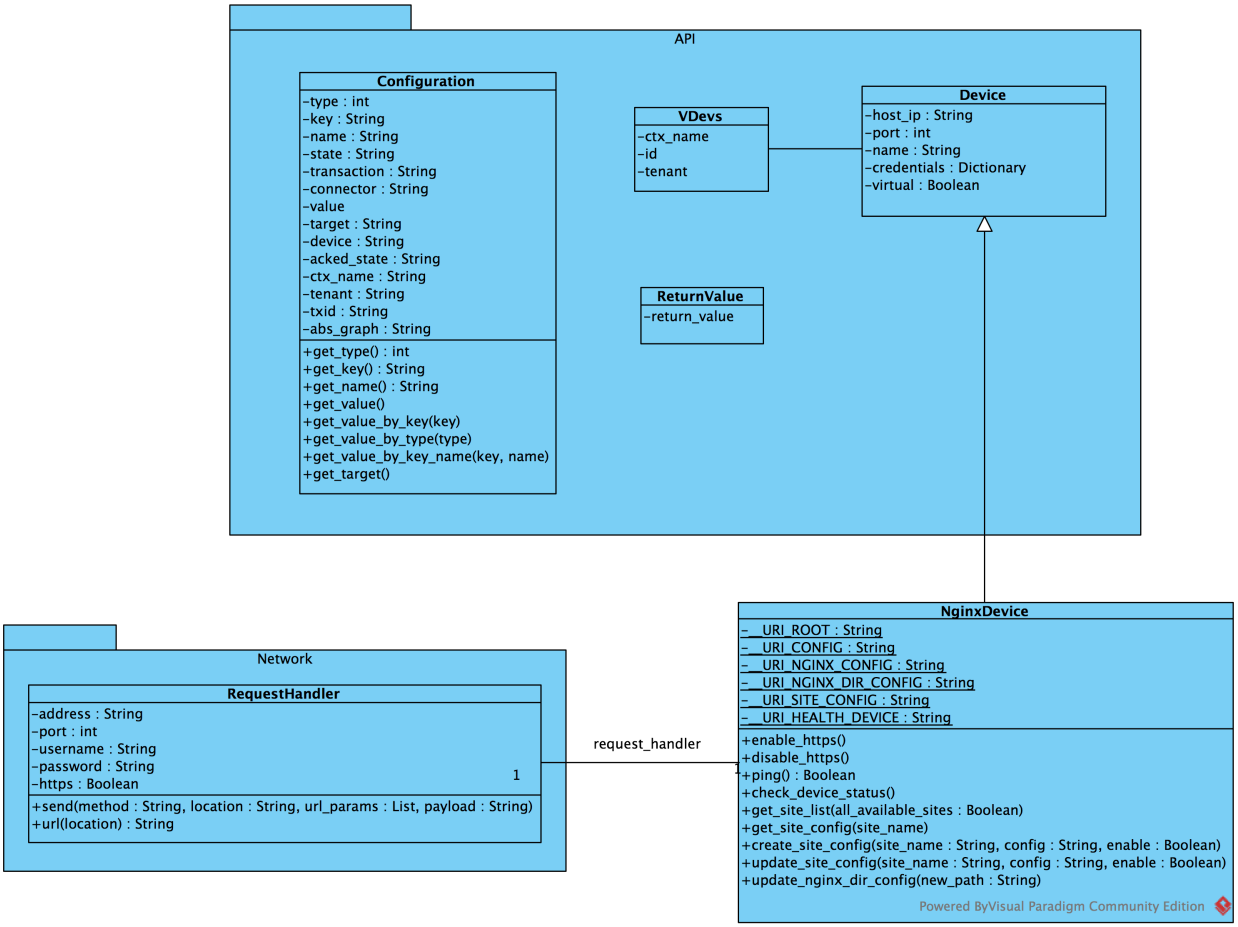

Figure 6.2: Classes encapsulating the API parameters, representing a NGINX device and handling the connectivity

6.2.2 NGINX configuration representation

Before explaining the parsing, we will review how an NGINX configuration file is represented in the script. It is based on the format explained in section 6.1.1. The classes are shown in FIGURE 6.3. They are organised on a "tree-structured" way such as in the configuration file. Each class represents a block or a directive. The similitude between this organisation and the NGINX configuration allows the developer to easily add a new block or directive.

Note that some directives are represented as a variable and not a class, e.g. `method` the load-balancing algorithm. This is a "shortcut" allowing the structure to be clearer by reducing the number of classes. It can be done because the directive doesn't require more than 1 parameter (unlike the `server` directive for instance).

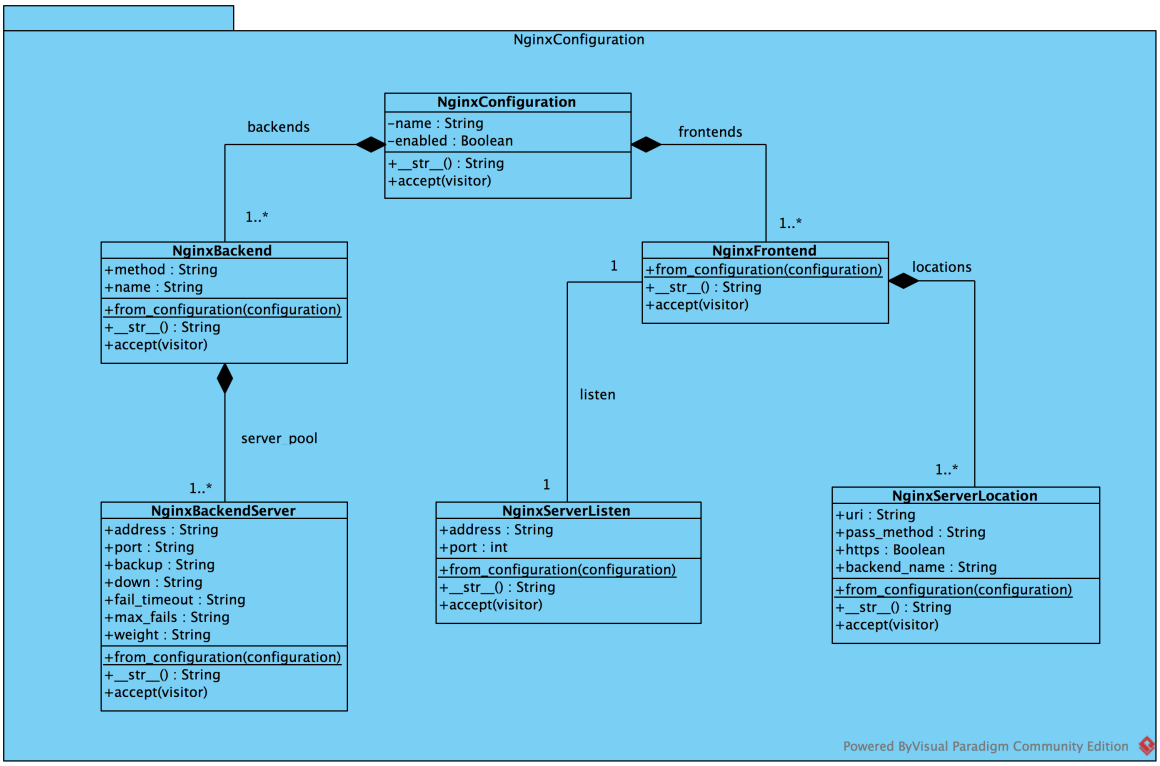

Figure 6.3: Classes representing a NGINX configuration in the device script

6.2.3 Parsing

After receiving the `configuration` argument and encapsulate it with `API.Configuration`, we have to parse it to create the classes representing that configuration (see FIGURE 6.3). The class `ConfigurationParser` handles the parsing and the generation of these classes.

FIGURE B.2 shows a simplified example of values that can be given in argument[2] and that needs to be parsed. The root is `(0, ",*)` and has 2 useful kinds of child:

1. `(4,*,*)`: `vnsMDevCfg` part, i.e. all the values for `frontendServer` and `upstream`. In FIGURE B.1, they are all the values under the 'Device Config' folder.
2. `(1,*,*)`: `vnsMFunc` part, i.e. the multiple NGINX configurations and their linkages to the values mentioned above (1.) and an additional part: 'Management interface'.

Only the `vnsMFunc` part has to be exported at the end. This organisation induces a 2-phases parsing.

First step

Scan across the whole argument and do a different action regarding the child visited (see above):

- If `(4,*,*)`: construct the object regarding to the key (`frontendServer` gives `NginxFrontend` and `upstream` gives `NginxBackend`). The code to instantiate the instance is in the method `from_configuration(configuration)` in the corresponding class, where the argument `configuration` is a subtree where the root is `(4,frontendServer,*)` or `(4,upstream,*)`.
- If `(1,*,*)`: navigate through the tree until having:

[2]This example of configuration is shown in FIGURE B.1 in the APIC.

- `(4,'configuration',*)`: retrieve the values of `target` of the linkages to memorize the names of the `frontendServer` and `upstream` to export.
- `(4, 'management', *)`: retrieve the management values.

Second step

Go through the names `frontendServer` and `upstream` for each configuration, get back the corresponding instances of `NginxFrontend` and `NginxBackend` and instantiate the configuration (`NginxConfiguration`).

Finally, those `nginx` configurations can be retrieved by `get_nginx_configurations()`, as well as the management configuration by using `get_management_configuration()`.

6.2.4 Exportation

After having a `NginxConfiguration` instance, it is possible to export the configuration into the NGINX format (cfr. LISTING 6.1).

The way the exportation works is similar to the visitor pattern. Each class representing a NGINX configuration (FIGURE 6.3) has an `accept(visitor)` method. The `visitor` argument has to be a function that the object will execute. It will first be executed on the root and then will call the `accept` method on its children,... The exportation function is written in the file `FileExporter.py` in the `Exporter` package.

Such a system allows the exportation function to be completely extracted from the structure forming the NGINX configuration. For instance, if a new exportation method should be implemented, it could be integrated in an easier way, outside the structure.

Each call of the exportation function on a node returns either a `Block` or a `Directive`, both shown in FIGURE 6.4. They represent the elements of a NGINX configuration. FIGURE 6.4 shows that a `Block` can contain `Directive`s and other `Block`s. Also, both can have some parameters.

Figure 6.4: Classes representing elements of a NGINX configuration file: Block and Directive

The call to the export function on the root will first call the function on the children,... in such a way that it works in a depth-first fashion. At each step, the function returns a `Directive`, or a `Block` containing `Block`s or `Directive`s from previous calls. The dump into a string works in a similar way, except that the method `__str()__` is implemented in `Block` and `Directive` and it calls the same method on the `Block`s and `Directive`s that it contains, when building the string for the current instance.

6.2.5 Pushing the configuration

Once the configuration generated, the controller needs to send it to the device, and the device needs to process it to activate the new load-balancing configuration. To achieve that, an agent has to be running on the load-balancer, providing a REST API for the controller (see section 6.3).

Before sending the configuration, the script first checks if the configuration is already on the device, so that it can either add or update it. For now, both of them lead to the same result as they do the same flow of actions. However, in the future, some distinctions could be made and different actions could be taken depending if it is an add or an update.

The device is abstracted by the class `NginxDevice` that inherits from `API.Device`, see FIGURE 6.2. It provides high-level methods to communicate with the load-balancer. These methods uses the low-level method `send` from `RequestHandler` that uses the Python `requests` library to send HTTP requests to the device.

6.3 Communication with the device

This section explains with more details the agent and how it is working.

6.3.1 APIs

These APIs are listed in TABLE 6.1.

Ping the agent to know if it is responding	`GET`	`/`
Get the list of existing configurations	`GET`	`/config/site`
Get a particular configuration	`GET`	`/config/site/:site_name`
Push a particular configuration	`POST`	`/config/site/:site_name`
Update a particular configuration	`PUT`	`/config/site/:site_name`
Change the location of the NGINX directory	`POST`	`/config/nginx/directory`
Get health score of the device	`GET`	`/health/device/:device_name`

Table 6.1: Agent REST APIs

6.3.2 *Flask-RESTful*

The APIs are provided using *Flask-RESTful* [5], an extension of *Flask* to build more rapidly a RESTful interface.

The usage of *Flask-RESTful* is quite simple. A *flask.ext.restful.Api* object is instantiated, to which we add an object inheriting from `flask.ext.restful.Resource` for each URI (e.g. '/', '/device',...). Each of these object defines methods corresponding to the action when receiving a particular HTTP method. For example, `PingAPI` is defined for the URI '`/`' and defines the method `get()` which returns a string.

The agent provides a simple HTTP authentication via `flask.ext.httpauth`. In order to have a more "plug & play" solution, the authentication is based on the credentials on the load-balancer system, using `pam`. `pam` provides a simple interface (`authenticate`) to check those credentials. Given that, the resources can be accessed only if the credentials are validated. It is still not secured since the credentials are in clear in the requests. The agent can thus run in HTTPS mode instead of HTTP, but a certificate and a private key need to be specified.

The agent has 2 parameters needed when running it:

- `-ip <ip>` [mandatory]: ip on which the agent is running,
- `-https` [optional]: run in HTTPS if given.

6.3.3 Core

In this section, I will explain how the different capabilities are provided by the agent.

Since NGINX configuration are files, it was needed to have a class to handle all these IO operations. A set of class methods were thus implemented to manage files and, more specifically, configuration files. Within the class `IO`, generic (i.e. "low-level") methods are provided:

- List the files within a directory,
- Read a file,
- Write a file,
- Update a file,
- Check if a file exists.

With these methods, some configuration-related actions are implemented:

- List configurations (enabled or not),
- Get a configuration from its name,
- Create or update a configuration,
- Enable or disable a configuration.[3]

[3] A configuration is enabled when a symbolic link exists in the directory `sites-enabled` (from `sites-available`). To make this work from Python, the UNIX command lines `ln` and `rm` are used as such from in the file, using the library `subprocess`.

Part IV

Results

In this chapter, I will explain how the device package is used, more generally, how to obtain a load-balancer managed by the APIC in an ACI fabric, dispatching the requests coming from one EPG to another.

6.4 Use case overview

Before explaining the different steps of the test, I will first explain the context of the use case. We want to add a load-balancing functionality to an ACI fabric. Requests are coming from the "external world", i.e. outside the fabric. These requests have to go through the load-balancer and have to be dispatched amongst several web servers in a server pool. FIGURE 6.5 shows the network that will be used, as well as the IP addresses and the interfaces.

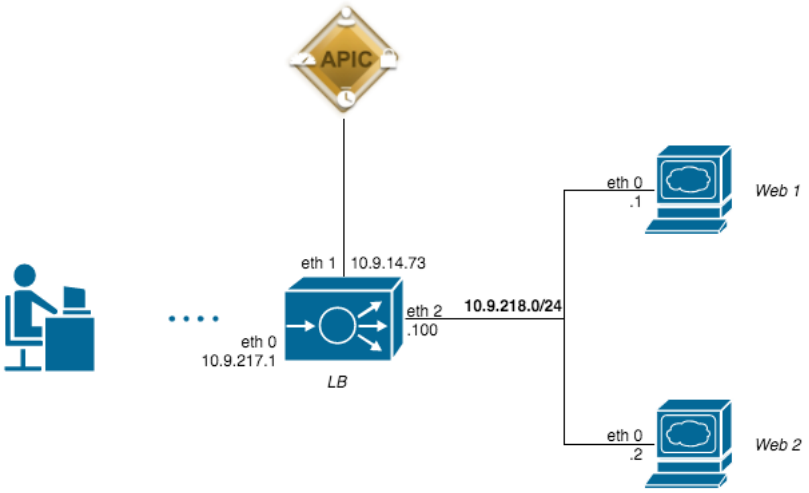

Figure 6.5: Testing network

Web1 and *Web2* are web servers. They both run NGINX as web server and a page "Welcome to WEB1" or "Welcome to WEB2" when they are reached, so that we can differentiate them we will request the load balancer. Both web servers have only one interface that have to belong to an EPG. We can set it up using the vSphere Client by putting the network adapter in the network corresponding to the EPG (`Web_Fabrice`), as shown in FIGURE C.1.

6.5 Test procedure

6.5.1 Launch agent

Before configuring the APIC, one needs to launch the agent on the load-balancer in order to receive the configuration from the controller. To do that, one runs simply the command `sudo python /path/to/agent.py -ip ip`, in our case, `ip` is 10.9.14.73.

6.5.2 Import

First of all, we need to upload the device package on the APIC, so that it can learn the structure of the device, ... As a recall, the device package is a `zip` archive containing the `XML` file (device model) and the Python scripts (device script). The importation is done here through the APIC GUI, as shown in FIGURE C.2.

6.5.3 Creation of the device

After the importation of the device package, one needs to register the device, i.e. where is located the load-balancer, to the APIC. The configuration in our case is shown in Figure C.3.

One must provide the following parameters:

- Name,
- Service type (Firewall, load-balancer (i.e. ADC), etc.),
- Device type (Physical and virtual): in our case, it is virtual and some more parameters have to come along with it, as the virtual machine where the service is running,
- The device package and the model in it,
- *APIC to Device Management Connectivity*: the APIC provides 2 ways to manage the device, either via Out-Of-Band connectivity (not going through the fabric), either via In-Band connectivity (going through the fabric),
- Credentials to access the device,
- Management IP and port: IP and port where the requests (configuration, etc.) will be send at,
- Device interfaces: in our case, with a virtual machine, one has to provide the name and the vNIC associated,
- Cluster interfaces: even though there is no real cluster, the device is included into a 'logical' cluster. Its interfaces have a type (consumer, i.e. where the function is used, and provider, i.e. where the function is provided), a name and a device interface.

6.5.4 Function profile

We will now create a function profile which will contain values for the configuration parameters in the device model. Note also that a function profile is contained within a function profile group. Figure C.4 shows a function profile containing the values of the use case. It is name `NginxLB_FP` and is contained in the function profile group `NginxLB_FPG`.

6.5.5 Service graph template

We will now create a service graph template. It just has the information on which function is used, in which mode (one-arm or two-arm) and the function profile associated. Figure C.5 shows the creation of the template named `NginxLB_SGT`.

6.5.6 Service graph instantiation

We can now *apply* a service graph template. We have to first select the consumer and the provider EPGs. In our case, the consumer EPG is the "outside world", i.e. `RoW_Netw`, and the provider EPG (server pool) is `Web_Fabrice`. Since the service graph is an extension of the concept of contract, we also need to provide information for the contract. Here, it will allow all the traffic between the two EPGs. This first step of service graph instantiation is shown in Figure C.6.

In the second part, we have to link the cluster interfaces to a bridge domain. This part is shown in Figure C.7. In our case, the bridge domain for `pool` was already selected by the APIC. For the `client` interface, we chose the `BD_NG` bridge domain. It serves as an intermediate between the load-balancer and the `RoW_Netw` to provide L3 connectivity between them. For that purpose, `BD_NG` have to have an *L3 Out network* associated to it.

6.6 Results

After those steps, the service graph is instantiated and running. It means that if we try to reach the load-balancer, it will dispatch the requests in a round-robin fashion (default one) amongst the *web1* and *web2*. Indeed, if we now send a request to 10.9.217.1, the "outside" interface of the load-balancer, we receive the pages "Welcome to WEB1" and "Welcome to WEB1" one after the other, proving that the web servers *web1* and *web2*.

6.6.1 Fault

If the agent is not running on the load-balancer, eventually, a call to `deviceHealth` will notice it and will trigger a custom fault **AgentNotResponding** that will go back up to the APIC. Such a fault was arbitrarily given a score of 0 in the script. Such an event is shown is Figure C.8.

Chapter 7

Conclusion

The goal, as mentioned before, is to develop a device package to integrate NGINX as a load-balancer in an ACI fabric. The final work serves as a being a proof-of-concept that, requiring some Python programming skills, it is possible to develop a device package and be able to insert correctly a new service. It does not serve the purpose of being a production-ready solution, even though this idea has been kept in mind during the development.

Furthermore, it was built in such a way that it has to be quite easy for new developers to take back the work and to continue the work as quickly as it could be in order to form a production-ready solution, some enhancements have to be made:

- *Improve the device model*: the model is not complete regarding to all the parameters and configurations made available by NGINX. In this case, only a subset of them has been implemented, the ones needed to have a functional configuration. For example, more interfaces could be added, as well as more parameters, more blocks, etc. If parameters or blocks are added, the corresponding script/class should be added as well in the script.
- *Dynamic endpoints*: at the time of writing, the exportation of the configuration relies on the `configuration` parameter in the API which comes from the function profile provided by the user. A call to `serviceModify` or `serviceAudit` will create a configuration String that only depends on its parameters.
  If a new endpoint (i.e. web server) comes into the EPG (or leaves an EGP), the APIC provides a way to trigger a notification when it happens. The method `attachEndpoint` or `detachEndpoint` are called with the information of the device as parameter. I ran into several problems, that are not currently solved at this time:
  - The information received through the API are device-related (e.g. address, name, etc.), there is no way the APIC can give service-related (e.g. load-balancing algorithm, etc.) information.
  - The information can not be added to the knowledge of the APIC of the configuration (i.e. what has been exported, see FIGURE C.4 as a recall). This way, a call to `serviceModify` will reinitialize the configuration without taking into account the new endpoint.
- *Intelligent pushing system*: for now, when `serviceModify` or `serviceAudit` is called, a String containing the configuration is exported only from the parameters received. This way is consistent with the fact that a device package is stateless, it only depends on its inputs, however, it leads to some limitations like new endpoints (see bullet above).
- *Improve health score*: the health is very simply and naively computed, although the usage of an external monitor can be used. For example, a *New Relic* agent was installed in the testing setup to try an external solution. However, this construction could bring a too big overhead or may not be suitable for all usages.

Part V

Appendices

Appendix A

Diagrams

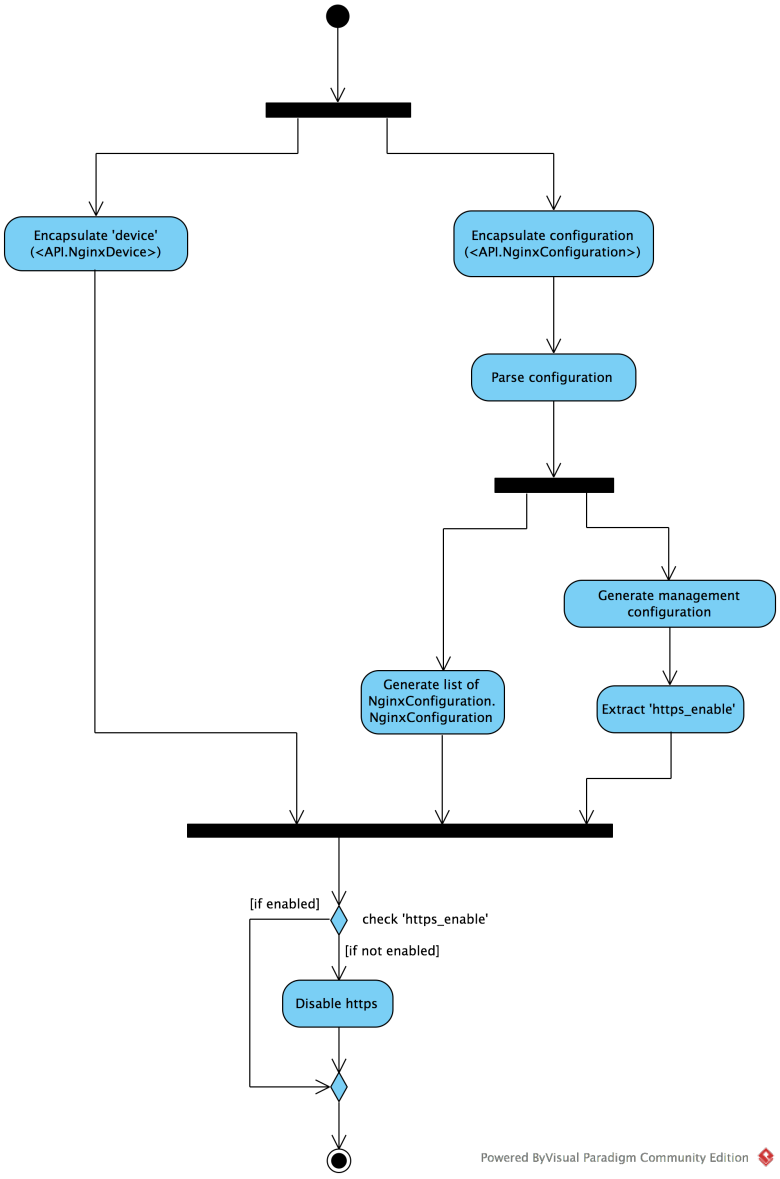

Figure A.1: Use case flow (pre-processing)

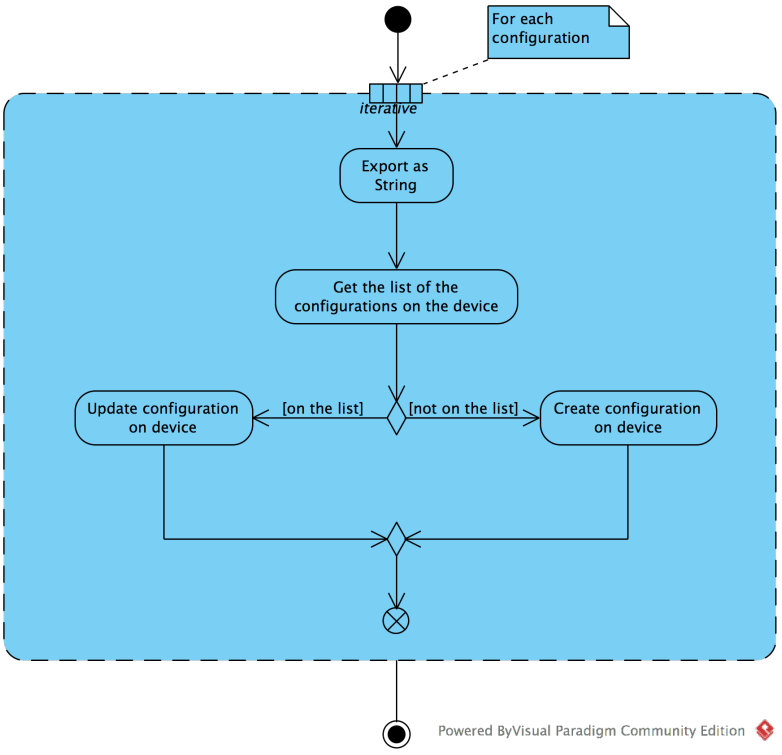

Figure A.2: Use case flow (generation and sending)

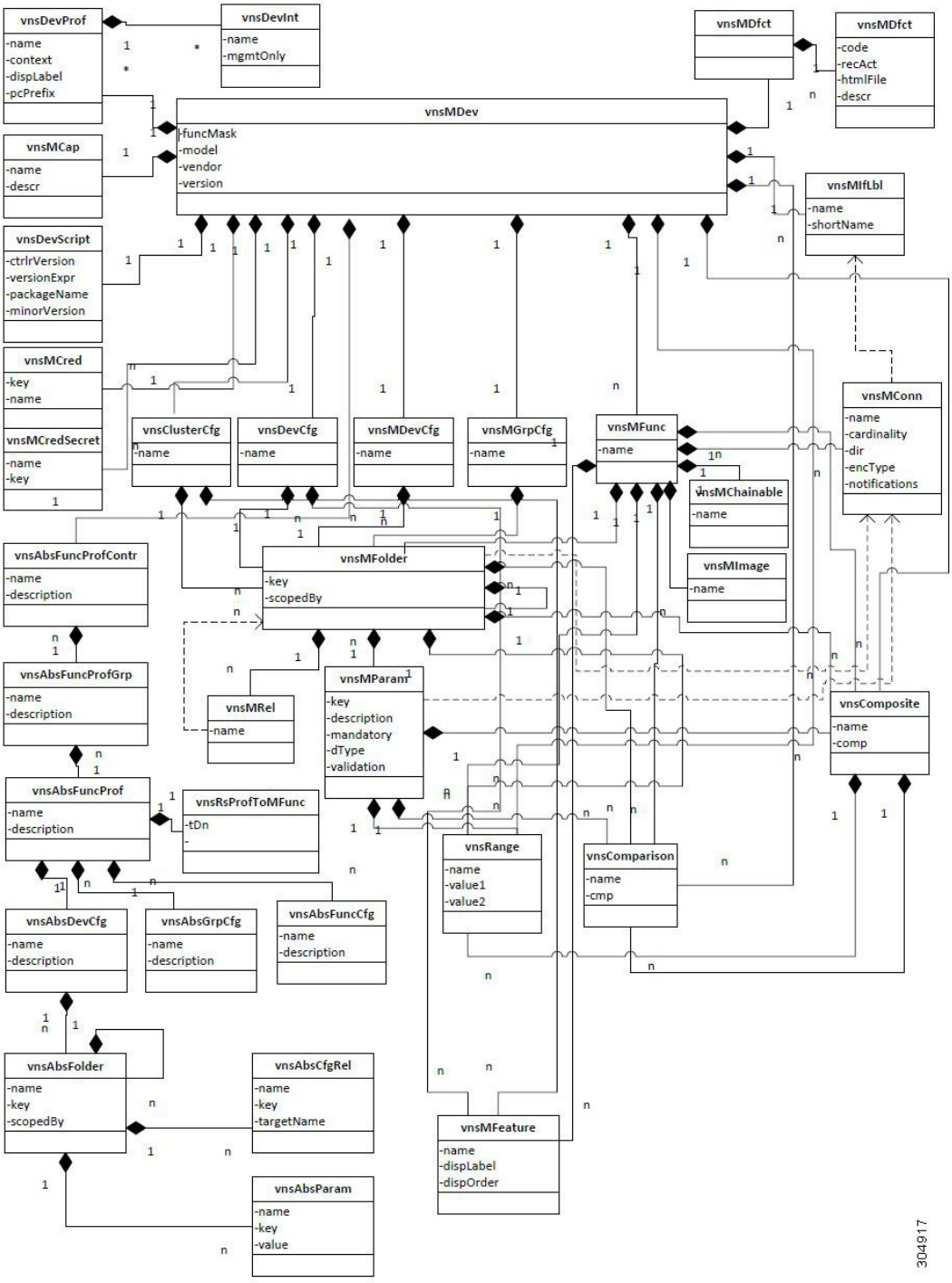

Figure A.3: Managed Object Model for representing a device [2]

Appendix B

Example

Folder/Param	Name	Value	Mandatory	Locked	Shared
Device Config	Device				
Backend	upstream			false	false
Server	web1			false	
Address	ip	10.9.218.1	false	false	
Port	port	80	false	false	
Weight	weight	3		false	
backup	backup	true		false	
fail_timeout	fail_timeout	5		false	
max_fails	max_fails	10		false	
Server	web2			false	
Address	ip	10.9.218.2	false	false	
Port	port	80	false	false	
Weight					
backup					
fail_timeout					
max_fails					
Load-balancing algorithm					
Name	upstreamName	backend	false	false	
Listen	listen			false	
Address	address	10.9.217.1	false	false	
Default (True/False)					
Port	port	80	false	false	
Location	location			false	
Backend name	backend_name	backend	false	false	
HTTPS (True/False)					
Modifier					
Pass method					
URI	uri	/	false	false	
Function Config	Function				
Configuration	Configuration			false	false
Front-end server	frontendServerCfg			false	
Select front-end server	frontendServerRel	frontendServer	false	false	
backend	upstreamCfg			false	
Select backend	upstreamRel	upstream	false	false	
Enable the configuration (Tr…	enabled	true	false	false	
Management interface	management			false	false
HTTPS	https	false	false	false	

Figure B.1: Example of configuration on the APIC

```
{(0, '', 5480): {'value':
    {(7, '', '2129920_49162'): {...},
     (4, 'frontendServer', 'frontendServer'): {'value':
        {(4, 'listen', 'listen'): {'value':
            {(5, 'address', 'address'): {'value':
                '10.9.217.1'},
             (5, 'port', 'port'): {'value':
                '80'}}},
         (4, 'location', 'location'): {'value':
            {(5, 'backend_name', 'backend_name'): {'value':
                'backend'},
             (5, 'uri', 'uri'): {'value':
                '/'}}}}},
    (8, '', 'NginxDevice_pool_2129920_16389'): {...},
    (10, '', 'NginxDevice_client'): {'cifs': {'NginxDevice_Device_1': 'eth0'}},
    (7, '', '2129920_16389'): {...},
    (10, '', 'NginxDevice_pool'): {'cifs': {'NginxDevice_Device_1': 'eth2'}},
    (8, '', 'NginxDevice_client_2129920_49162'): {'vif': 'NginxDevice_client'},
    (1, '', 20350): {'value':
        {(3, 'LoadBalancer', 'N1'): {'value':
            {(4, 'management', 'management'): {'value':
                {(5, 'https', 'https'): {'value':
                    'false'}}},
             (2, 'external', 'consumer'): {'value':
                {(9, '', 'NginxDevice_client_2129920_49162'): {...}}},
             (4, 'configuration', 'Configuration'): {'value':
                {(4, 'upstreamCfg', 'upstreamCfg'): {'value':
                    {(6, 'upstreamRel', 'upstreamRel'): {'target': 'upstream'}}},
                 (5, 'enabled', 'enabled'): {'value':
                    'true'},
                 (4, 'frontendServerCfg', 'frontendServerCfg'): {'value':
                    {(6, 'frontendServerRel', 'frontendServerRel'): {'target': 'frontendSer
ver'}}}}},
             (2, 'internal', 'provider'): {'value':
                {(9, '', 'NginxDevice_pool_2129920_16389'): {...}}}}}}},
     (4, 'upstream', 'upstream'): {'value':
        {(4, 'server', 'web2'): {'value':
            {(5, 'port', 'port'): {'value':
                '80'},
             (5, 'ip', 'ip'): {'value':
                '10.9.218.2'}}},
         (4, 'server', 'web1'): {'value':
            {(5, 'backup', 'backup'): {'value':
                'true'},
             (5, 'weight', 'weight'): {'value':
                '3'},
             (5, 'port', 'port'): {'value':
                '80'},
             (5, 'max_fails', 'max_fails'): {'value':
                '10'},
             (5, 'ip', 'ip'): {'value':
                '10.9.218.1'},
             (5, 'fail_timeout', 'fail_timeout'): {'value': '5'}}},
          (5, 'upstreamName', 'upstreamName'): {'value':
            'backend'}}}}}}
```

Figure B.2: Simplified example of the `configuration` argument

```
from API.Configuration import Configuration
from Exporter.FileExporter import file_exporter
from NGINXConfiguration.ConfigurationParser import ConfigurationParser
from NginxDevice import NginxDevice

__author__ = 'Fabrice Servais'
device = {'name': 'NginxLoadBalancer', 'virtual': True, 'devs': {'NginxLoadBalancer_Device_
1': {'creds': {'username': 'fservais', 'password': '<hidden>'}, 'host': '127.0.0.1', 'port'
: 80, 'virtual': True}}, 'host': '127.0.0.1', 'contextaware': False, 'port': 5000, 'creds':
 {'username': 'fservais', 'password': '<hidden>'}}
configuration = {...}


print("Initialize the configurations...")
# Convert configuration into API object
api_config = Configuration(configuration)
print("> Configuration\n{}".format(api_config))

# Create NginxDevice
nginx_device = NginxDevice(device)
print("> Device\n{}".format(nginx_device))

# Convert configuration into NGINX objects
parser = ConfigurationParser()
parser.from_API_configuration(api_config)
nginx_configurations = parser.get_nginx_configurations()
management_configuration = parser.get_management_configuration()
https_enable = management_configuration['https']

print("Configuration: {} (len {})".format(nginx_configurations, len(nginx_configurations)))

for nginx_configuration in nginx_configurations:
    print(">> For configuration {}".format(nginx_configuration.name))
    print("Generating the string...")
    # Generate (nginx) string of the configuration
    string_config_file = nginx_configuration.accept(file_exporter())
    print(string_config_file)
```

Figure B.3: Sample code to export the String configuration, given Figure B.2

Appendix C

Testing

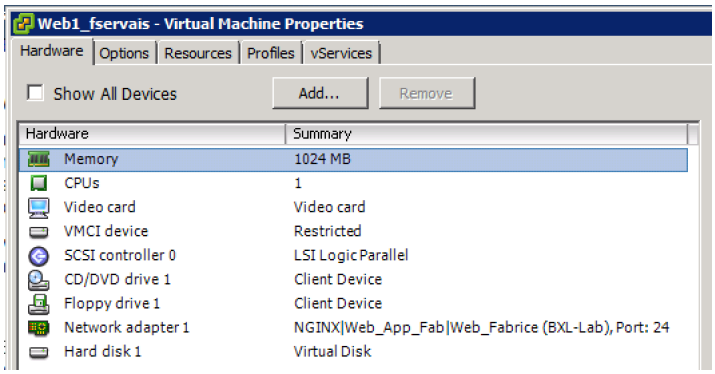

Figure C.1: Virtual machine configuration: Web1

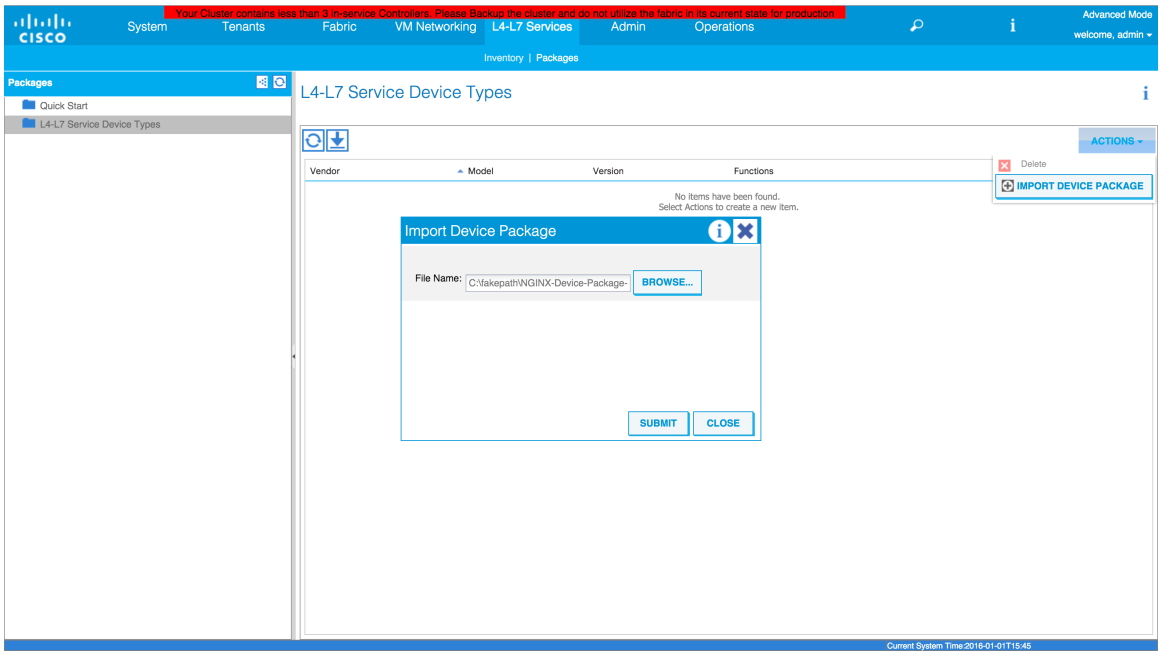

Figure C.2: Import of the device package

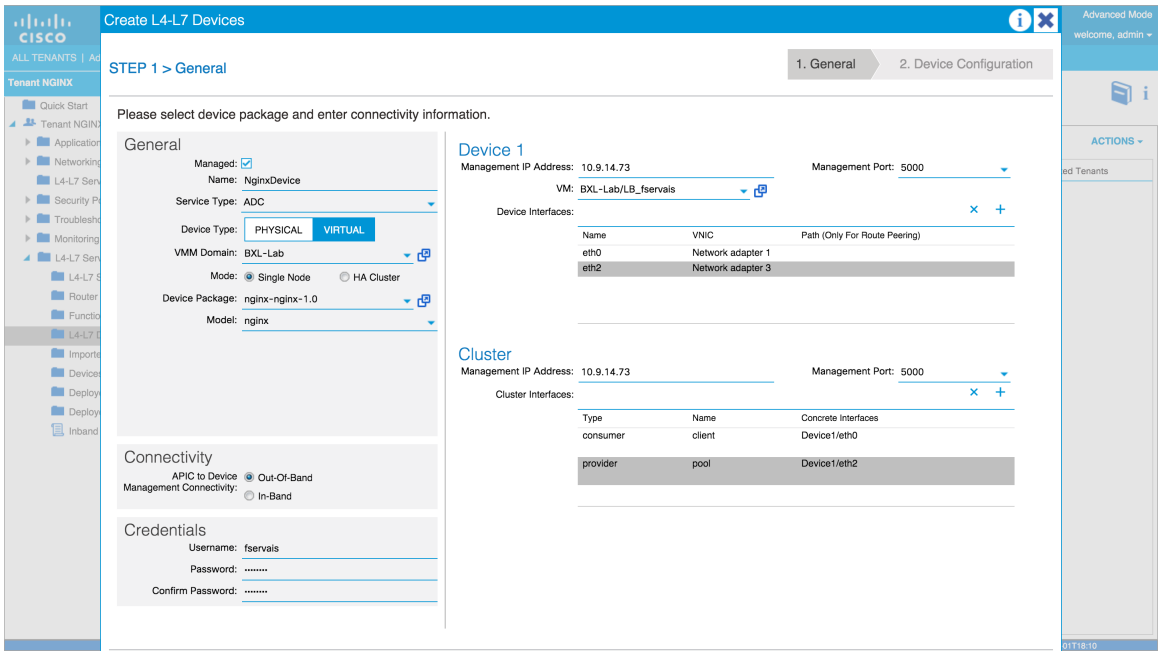

Figure C.3: L4-L7 device creation

Folder/Param	Name	Value	Mandatory	Locked	Shared
Device Config	Device				
Backend	upstream			false	false
Server	web1			false	
Address	ip	10.9.218.1	false	false	
Port	port	80	false	false	
Weight				false	
backup				false	
fail_timeout				false	
max_fails				false	
Server	web2			false	
Address	ip	10.9.218.2	false	false	
Port	port	80	false	false	
Weight					
backup					
fail_timeout					
max_fails					
Load-balancing algorithm					
Name	upstreamName	backend	false	false	
Listen	listen			false	
Address	address	10.9.217.1	false	false	
Default (True/False)					
Port	port	80	false	false	
Location	location			false	
Backend name	backend_name	backend	false	false	
HTTPS (True/False)					
Modifier					
Pass method					
URI	uri	/	false	false	
Function Config	Function				
Configuration	Configuration			false	false
Front-end server	frontendServerCfg			false	
Select front-end server	frontendServerRel	frontendServer	false	false	
backend	upstreamCfg			false	
Select backend	upstreamRel	upstream	false	false	
Enable the configuration (Tr...	enabled	true	false	false	
Management interface	management			false	false
HTTPS	https	false	false	false	

Figure C.4: Function profile

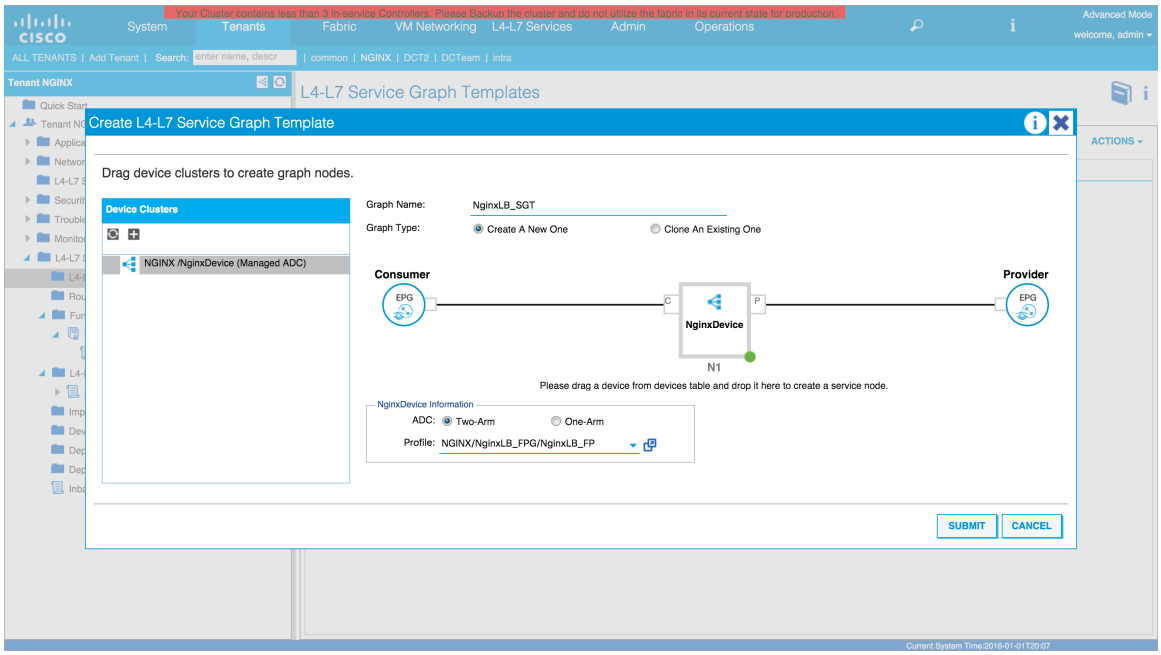

Figure C.5: Service graph template creation

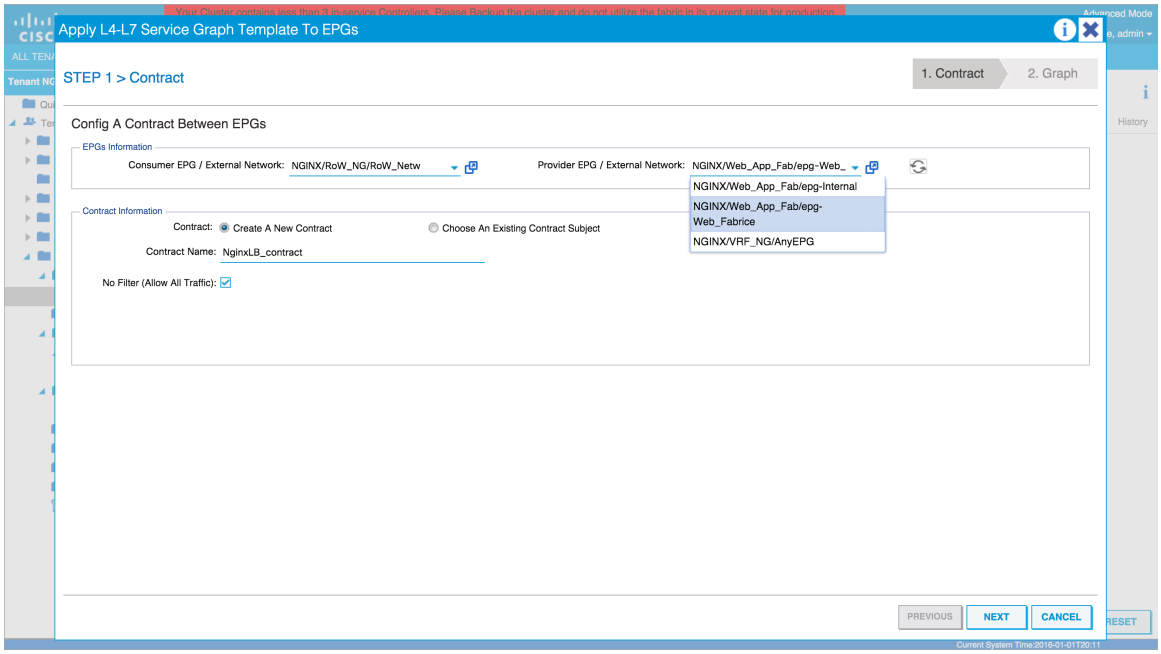

Figure C.6: Service graph instantiation - Contract

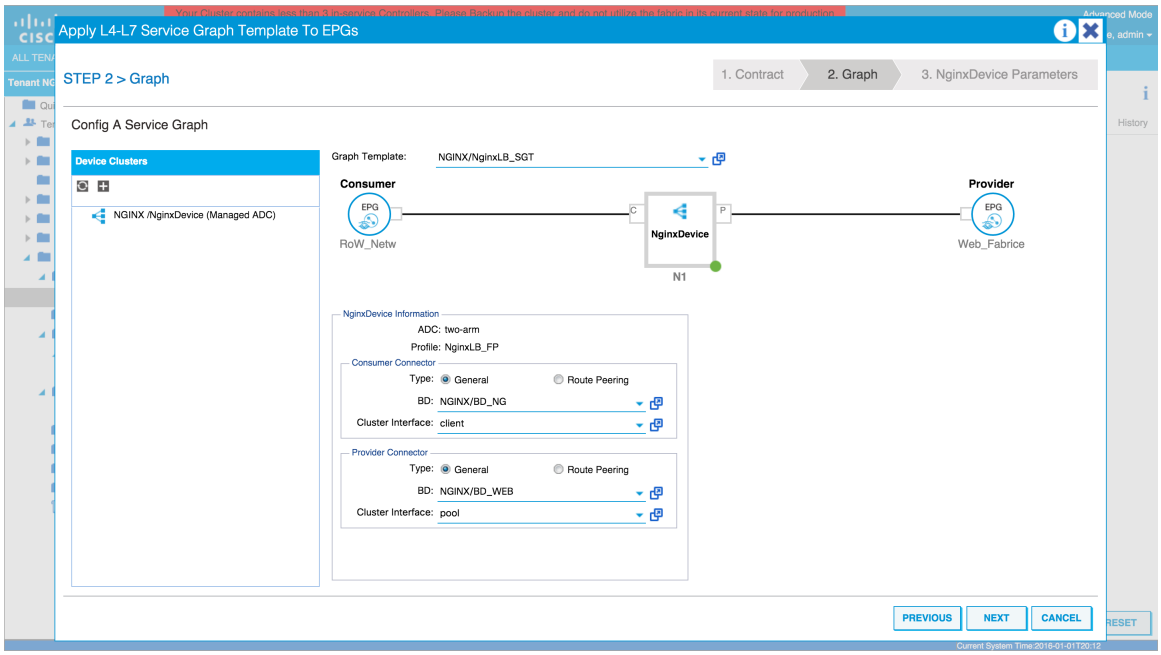

Figure C.7: Service graph instantiation - Graph

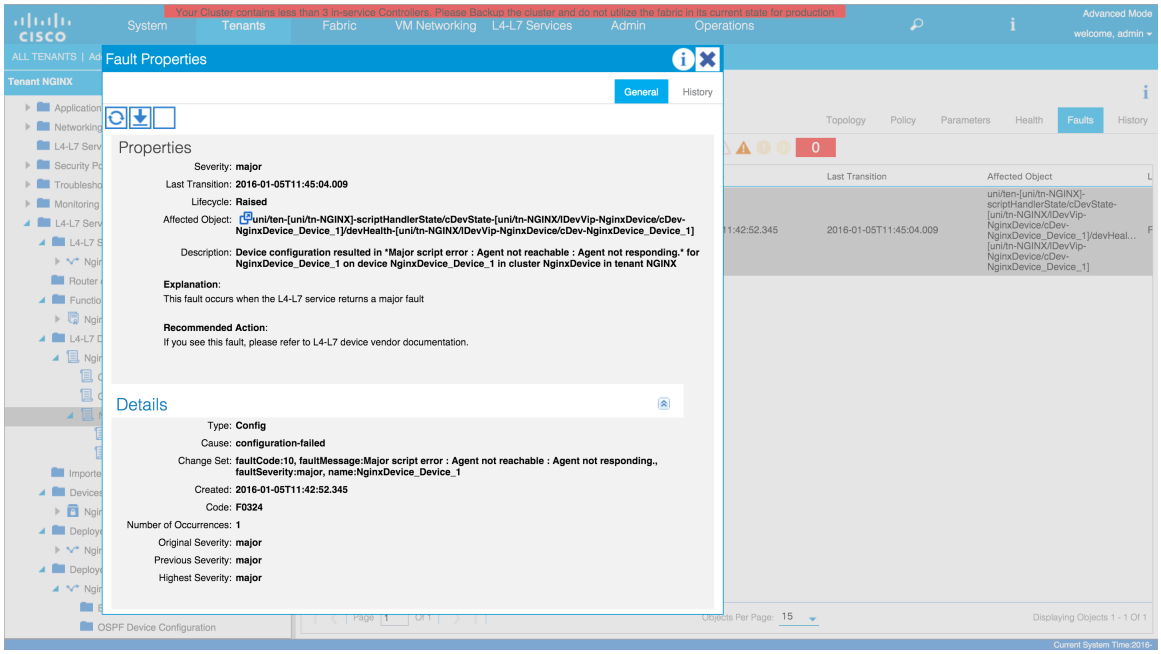

Figure C.8: Custom fault occurrence on the APIC

Bibliography

[1] *Apache vs Nginx: Practical Considerations*. 2015. URL: https://www.digitalocean.com/community/tutorials/apache-vs-nginx-practical-considerations.

[2] *Cisco APIC Layer 4 to Layer 7 Device Package Development Guide, Release 1.2(1x)*. 2015.

[3] *Cisco Application Centric Infrastructure Fundamentals*. 2015. URL: http://www.cisco.com/c/en/us/td/docs/switches/datacenter/aci/apic/sw/1-x/aci-fundamentals/b_ACI-Fundamentals.pdf.

[4] *Cisco's corporate information*. 2015. URL: http://newsroom.cisco.com/overview.

[5] *Flask-RESTful*. 2015. URL: http://flask-restful-cn.readthedocs.org/en/0.3.4/.

[6] NGINX *documentation*. 2015. URL: http://nginx.org/en/docs/.

[7] Maurizio Portolani. *The Policy Driven Data Center with ACI. Architecture, Concepts, and Methodology*. Cisco Press, 2015.

[8] *Using nginx as HTTP load balancer*. URL: http://nginx.org/en/docs/http/load_balancing.html.