Automation Leveraging NSX REST API

Automate the network via programming languages, vRO, and other tools leveraging NSX REST API

NSBU
UPDATED 6/21/2016

VERSION 1.0

Intended Audience

The intended audiences for this paper are devops/automation/system engineers and network engineers looking to automate logical networking and security within their virtualized environment. The goal of this paper is to demonstrate how automation can be accomplished in several ways by leveraging the NSX REST API. This paper explains the fundamental concepts around REST API, programming languages, and automation tools; nevertheless, the reader should already have some general basic understanding of these tools and concepts. The main goal of this document is to enable the reader to easily leverage NSX REST API automation within their environments.

Overview

VMware has helped transform the data center landscape by introducing virtualization technologies for compute, storage, and networking. By replicating traditional hardware constructs in software, VMware allows for not only efficiency in hardware utilization, improved resiliency, lower costs, but also automation at a completely new level that has never been possible before. Gone are the days of hardware specific APIs, screen/CLI scraping and vendor-specific tool sets that would have to be stitched together to create a complete solution.

VMware NSX provides a RESTful API service via NSX Manager that can be consumed in several ways.

The NSX REST API can be consumed directly via a tool/library such as cURL or a REST Client like Postman, via multiple popular programming languages, and via orchestration cloud management tools. Popular programming languages such as Python, PowerShell, Perl, Go, and Java have REST client libraries which can easily be utilized to consume the NSX REST API. This means that elaborate workflows and complete systems/portals can be created to provide custom automation, management, and monitoring capabilities.

Tools such as VMware vRealize Orchestrator (vRO) or configuration frameworks like Ansible can also be used to create advanced workflows for NSX. vRO also provides a HTTP REST API client allowing for NSX REST API to be called directly from the tool; a NSX vRO Plugin is also available.

Further, cloud management systems such as vRealize Automation (vRA) and Openstack incorporate and leverage pre-packaged automation solutions. A vRO plugin for vRA is also available allowing for purpose-built custom automation of NSX objects.

Why You Should Care

By consuming the NSX REST API natively or through an automation tool or cloud management platform, customers can achieve efficiency through automated processes and workflows. There are several important reasons why automation is center-stage in modern data centers:

- Increases productivity by decreasing manual steps for configuration, provisioning, failover/recovery, etc.
- Avoids manual configuration mistakes by automating tasks to follow preset, defined processes
- Allows for an increase in the number of managed systems without a significant opex increase
- Allows the infrastructure to support continuous operations and automated scaling; automatic creation of testing environments can be part of continuous integration workflows
- Repeatable and consistent application environments can be created which minimizes customizations and provides consistency for ease of maintenance and troubleshooting

In a nutshell, by leveraging the NSX automation capabilities, customers can not only increase productivity, but can also avoid increased costs, reduce errors due to manual error-prone processes, simplify processes, and enable ease of troubleshooting and recovery.

Introduction to Rest API

Representational State Transfer (REST) is a software architecture that enables users to query and modify applications states using simple HTTP primitives.

REST defines the use of POST, GET, PUT and DELETE to implement CRUD (create, read, update and delete) operations. The below table describes various HTTP verbs and their corresponding operations; the context used is NSX as the system being utilized.

HTTP "Verbs"	Use	CRUD
POST	Create an NSX object (e.g. logical switch)	Create
GET	Retrieve data about a single NSX object or multiple objects	Read
PUT	Modify the properties of an already existing NSX object	Update
DELETE	Remove an NSX object	Delete

Table 1: HTTP Verbs and CRUD Functions

A system with a northbound REST-based API enables any software that can consume standard-based protocols and formats such as HTTP, JSON, and XML to create clients that are able to interact with its objects and data structures. Simple tools such as **wget**, **cURL,** or even a web browser could be used to interact with an application that supports REST APIs. Programming and scripting languages that support HTTP operations can also be used to interact with REST API based applications.

API Specification Formats

While REST APIs are extremely useful to allow different software components to communicate, they can become complex to understand and to document. In the past, when developers had to write their implementation of an API (for either a client or a server) they had to analyze the documentation and manually build all the required interfaces. API documentation is also often presented in documents that do not follow any specific guideline for formatting and definition.

To solve these issues, several specification formats emerged in order to provide a consistent format to document and consume APIs.

API specification formats provide a number of benefits for developers, including:

- Non-ambiguous ways to define and document APIs
- Dramatic reduction of errors during implementation, testing and troubleshooting
- Diffusion of tools that can automatically generate API implementations in different programming languages using the specification as source

Two of the most common API documentation and specification formats, at the time of writing this document, are RAML and OpenAPI Specification (formerly known as Swagger). Both are widely utilized and have a large open source community. There are also tools that automate the conversion across the different API documentation and specification formats.

RAML is a language based on YAML; YAML is a syntax used to describe an API in a 'human friendly' way. YAML can easily be mapped to data types common to most high-level languages. RAML syntax can be parsed automatically to produce documentation, dynamic clients in different languages such as Python, and collections to be used with tools like Postman.

As an example, a company can automatically generate, from the application's source code, a RAML specification file for a product. This RAML file can then be then used by the same company to generate the Java source code to implement the server-side API for the product which can then be shared with third parties. A partner could use the same RAML file to generate Python source code to implement the client-side API. It's somewhat similar to what Web Service Definition Language (WSDL) enables for Simple Object Access Protocol (SOAP) based services: automatically generate client code in multiple languages for calling the web service specified or server stubs for implementing the service(s) instead. Consistency is a major benefit here: since all the implementations in different languages adhere to the same specification, there is less space for ambiguity.

Additional information on RAML and OpenAPI Specification can be found on the respective project web sites (RAML and OpenAPI Specification).

VMware offers a community supported RAML specification of NSX for vSphere API (nsxraml) that can be used to simplify the consumption of NSX services. Some of the examples and tools mentioned in this document use nsxraml. For additional details please check the nsxraml page on GitHub.

NSX - REST API

NSX provides a REST API interface via NSX Manager. Any REST Client can utilize this REST API interface provided by NSX Manager.

The next image shows the typical structure of a REST API call for NSX – using an example that shows scopes.

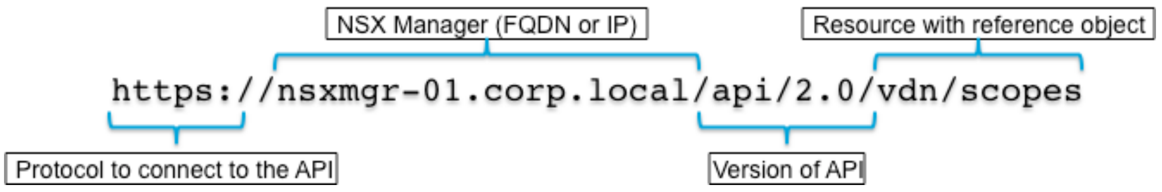

NSX Manager (FQDN or IP)
Resource with reference object
https://nsxmgr-01.corp.local/api/2.0/vdn/scopes
Protocol to connect to the API
Version of API

Figure 1: NSX API Structure (example)

For security reasons, NSX enforces authentication before querying or changing the state using REST APIs.

Using REST API with cURL

cURL is a simple tool used to transfer data from server to client. cURL supports various methods, one being HTTP. While cURL may be used to manage NSX manager, it’s not a very scalable approach unless cURL is being used from within another programming language etc., to help automate. As such, the following examples with cURL are primarily to help with the understanding how REST APIs work.

If you are trying this out on a Mac, use ***xmllint*** *to parse the output*

The below example uses cURL to call the REST API described in Figure 1 above. Note the usage of various cURL options: ***-k*** to ignore any SSL related warnings; ***-u*** to specify username and password.

```
skommu-mac:~ skommu$ cURL -k -u admin:password https://nsxmgr-01.corp.local/api/2.0/vdn/scopes | xmllint --format -
```

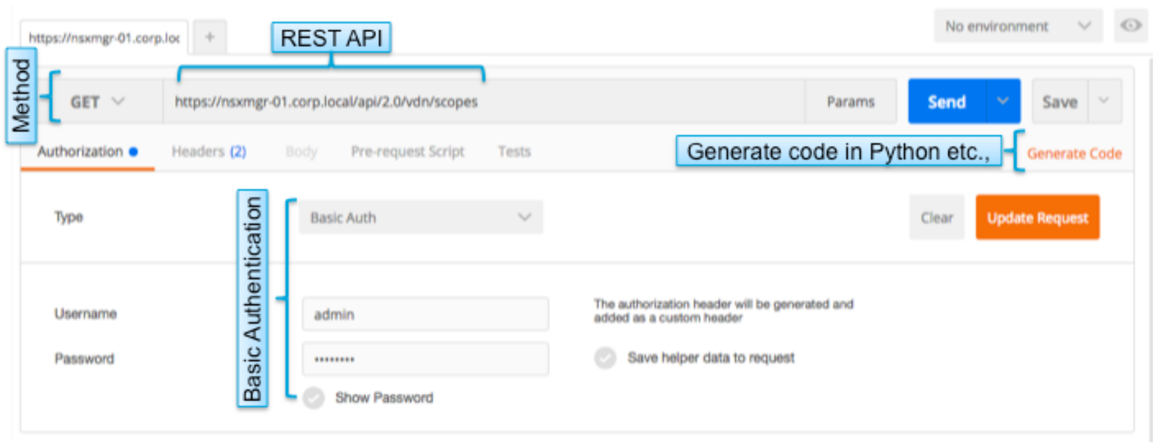
Following is the output for the above cURL request:

```
<?xml version="1.0" encoding="UTF-8"?>
<vdnScopes>
  <vdnScope>
    <objectId>vdnscope-9</objectId>
    <objectTypeName>VdnScope</objectTypeName>
    <vsmUuid>421DE20E-63C4-914C-785F-C760AB465B3F</vsmUuid>
    <nodeId>421DE20E-63C4-914C-785F-C760AB465B3F</nodeId>
    <revision>1</revision>
    <type>
      <typeName>VdnScope</typeName>
    </type>
    <name>Global NSX Zone</name>
    <description/>
    <clientHandle/>
    <extendedAttributes/>
    <isUniversal>false</isUniversal>
    <universalRevision>0</universalRevision>
    <id>vdnscope-9</id>
    <clusters>
      <cluster>
        <cluster>
          <objectId>domain-c332</objectId>
          <objectTypeName>ClusterComputeResource</objectTypeName>
          <vsmUuid>421DE20E-63C4-914C-785F-C760AB465B3F</vsmUuid>
          <nodeId>421DE20E-63C4-914C-785F-C760AB465B3F</nodeId>
          <revision>230</revision>
          <type>
            <typeName>ClusterComputeResource</typeName>
          </type>
          <name>Compute-POD-02</name>
          <scope>
            <id>datacenter-2</id>
            <objectTypeName>Datacenter</objectTypeName>
            <name>DC</name>
          </scope>
          <clientHandle/>
          <extendedAttributes/>
          <isUniversal>false</isUniversal>
          <universalRevision>0</universalRevision>
        </cluster>
      </cluster>
      <cluster>
        <cluster>
          <objectId>domain-c219</objectId>
          <objectTypeName>ClusterComputeResource</objectTypeName>
          <vsmUuid>421DE20E-63C4-914C-785F-C760AB465B3F</vsmUuid>
          <nodeId>421DE20E-63C4-914C-785F-C760AB465B3F</nodeId>
          <revision>243</revision>
          <type>
            <typeName>ClusterComputeResource</typeName>
          </type>
          <name>Compute-POD-01</name>
          <scope>
            <id>datacenter-2</id>
            <objectTypeName>Datacenter</objectTypeName>
            <name>DC</name>
          </scope>
          <clientHandle/>
          <extendedAttributes/>
          <isUniversal>false</isUniversal>
          <universalRevision>0</universalRevision>
        </cluster>
      </cluster>
    </clusters>
    <virtualWireCount>498</virtualWireCount>
    <controlPlaneMode>UNICAST_MODE</controlPlaneMode>
  </vdnScope>
</vdnScopes>
```

Figure 2: Result of REST API Call

Using REST API with Postman

Postman is a REST Client tool to execute REST APIs from the Chrome web browser. There are many different REST Client tools available for different browsers. RESTClient on the Firefox browser is another specific example. This example shows how to use Postman with the Chrome browser to call the same REST API used above.



GET https://nsxmgr-01.corp.local/api/2.0/vdn/scopes

Type: Basic Auth

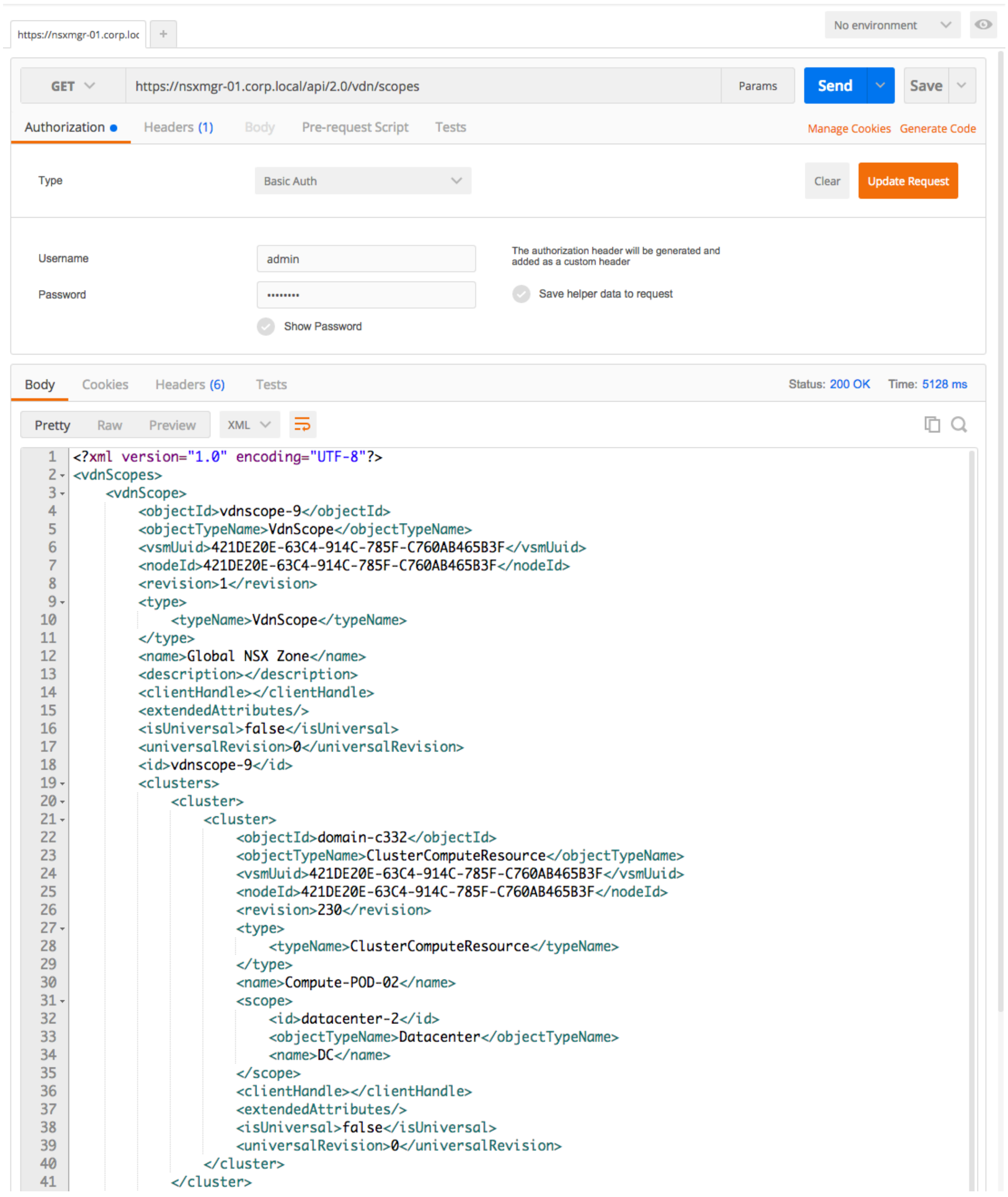
Username: admin

Password: ••••••••

The authorization header will be generated and added as a custom header

Figure 3: Postman GUI

The below figure shows the output excerpt from Postman, which can be seen is the same result obtained when using cURL.



GET https://nsxmgr-01.corp.local/api/2.0/vdn/scopes

Type: Basic Auth

Username: admin

Password: ••••••••

Show Password

The authorization header will be generated and added as a custom header

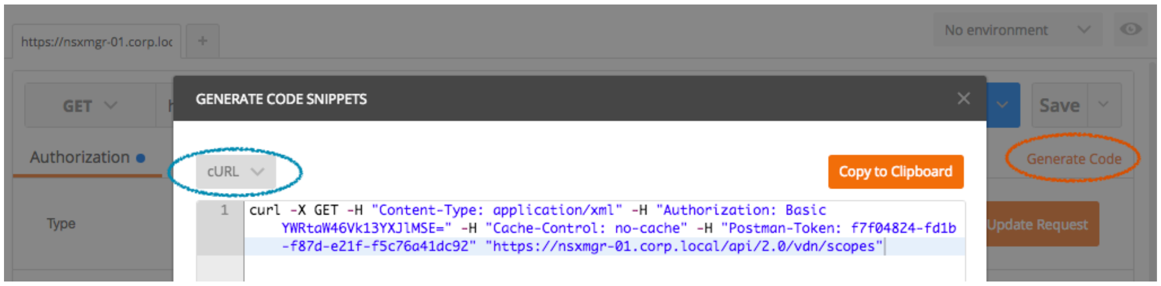
Save helper data to request

Status: 200 OK Time: 5128 ms

```
<?xml version="1.0" encoding="UTF-8"?>
<vdnScopes>
    <vdnScope>
        <objectId>vdnscope-9</objectId>
        <objectTypeName>VdnScope</objectTypeName>
        <vsmUuid>421DE20E-63C4-914C-785F-C760AB465B3F</vsmUuid>
        <nodeId>421DE20E-63C4-914C-785F-C760AB465B3F</nodeId>
        <revision>1</revision>
        <type>
            <typeName>VdnScope</typeName>
        </type>
        <name>Global NSX Zone</name>
        <description></description>
        <clientHandle></clientHandle>
        <extendedAttributes/>
        <isUniversal>false</isUniversal>
        <universalRevision>0</universalRevision>
        <id>vdnscope-9</id>
        <clusters>
            <cluster>
                <cluster>
                    <objectId>domain-c332</objectId>
                    <objectTypeName>ClusterComputeResource</objectTypeName>
                    <vsmUuid>421DE20E-63C4-914C-785F-C760AB465B3F</vsmUuid>
                    <nodeId>421DE20E-63C4-914C-785F-C760AB465B3F</nodeId>
                    <revision>230</revision>
                    <type>
                        <typeName>ClusterComputeResource</typeName>
                    </type>
                    <name>Compute-POD-02</name>
                    <scope>
                        <id>datacenter-2</id>
                        <objectTypeName>Datacenter</objectTypeName>
                        <name>DC</name>
                    </scope>
                    <clientHandle></clientHandle>
                    <extendedAttributes/>
                    <isUniversal>false</isUniversal>
                    <universalRevision>0</universalRevision>
                </cluster>
            </cluster>
```

Figure 4: Data Returned via the REST API Cal From Postman

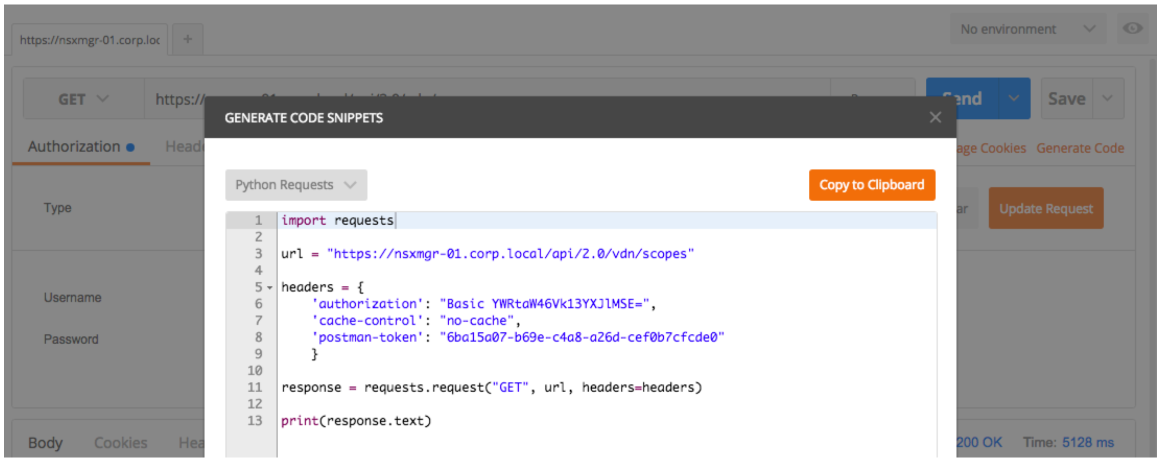
Postman is very useful for quickly and automatically generating code snippets for a given API call. To use this feature, one can click on the **Generate Code** option in the upper right corner of the Postman application and select the desired programming language for which the code should be generated for. This auto-generated code will work without any changes; however, if SSL related issues are encountered, the ***-k*** option can be added to the respective cURL call.



```
curl -X GET -H "Content-Type: application/xml" -H "Authorization: Basic
    YWRtaW46Vk13YXJlMSE=" -H "Cache-Control: no-cache" -H "Postman-Token: f7f04824-fd1b
    -f87d-e21f-f5c76a41dc92" "https://nsxmgr-01.corp.local/api/2.0/vdn/scopes"
```

Figure 5: Use Postman to generate cURL code snippet

Below is another example to generate Python code for the same example call using Postman. The procedure is similar to getting code for cURL; just select Python instead of cURL.



```
import requests

url = "https://nsxmgr-01.corp.local/api/2.0/vdn/scopes"

headers = {
    'authorization': "Basic YWRtaW46Vk13YXJlMSE=",
    'cache-control': "no-cache",
    'postman-token': "6ba15a07-b69e-c4a8-a26d-cef0b7cfcde0"
    }

response = requests.request("GET", url, headers=headers)

print(response.text)
```

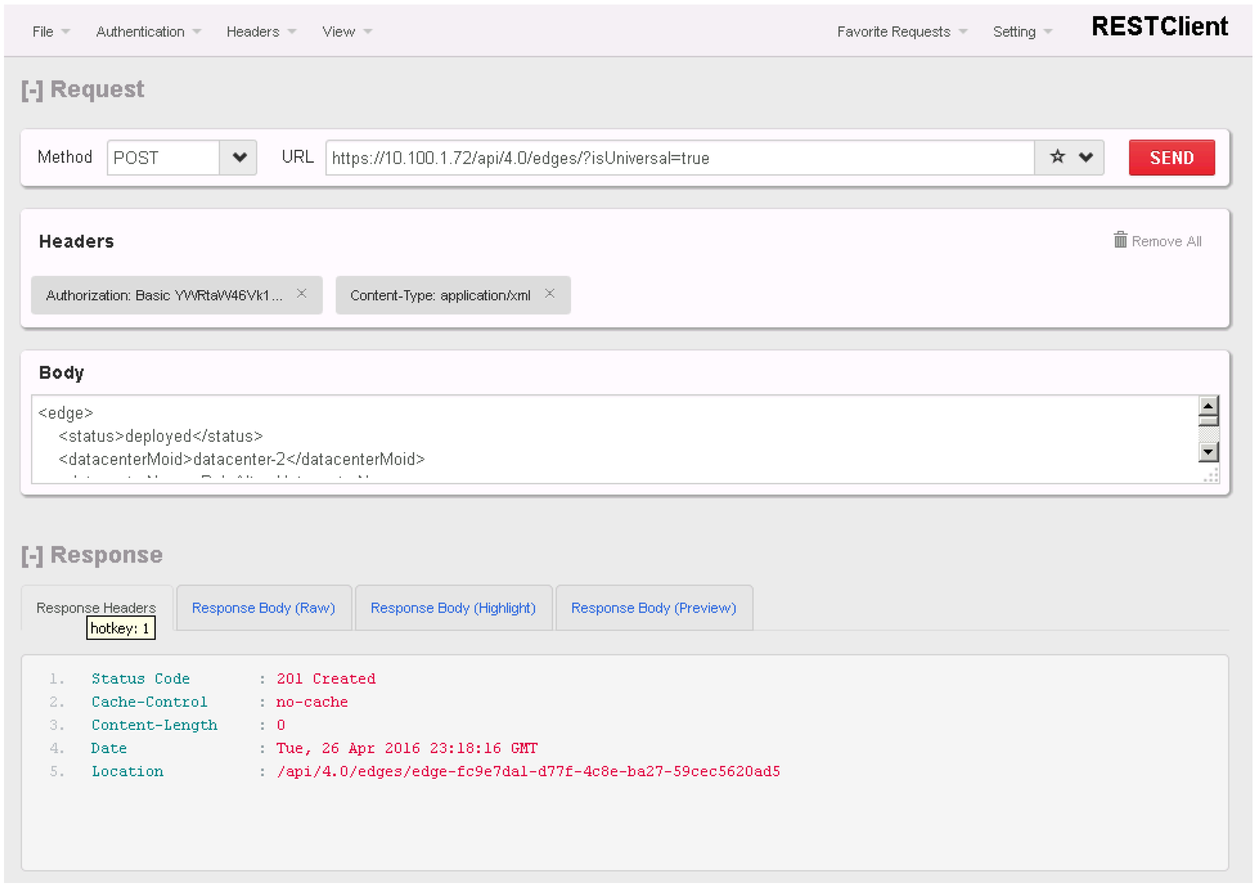
Figure 6: Use Postman to generate Python code snippet

*If SSL related issues are encountered with the generated Python code, pass the variable **Verify=False** to the the **requests.request** method:*

```
response = requests.request("GET", url, headers=headers, verify=False)
```

Example: create a Universal Distributed Logical Router (UDLR) in NSX 6.2.2

This example demonstrates using the RESTClient plugin on the Firefox browser to create a Universal Distributed Logical Router (UDLR) in NSX 6.2.2. The successful creation of the UDLR named **Universal DLR Test** is confirmed from the **Status Code** returned by the NSX Manager. The related XML data structure within the **Body** field shows all the available fields with their respective values for the desired NSX object. If an error is received when submitting the NSX REST API call, the returned **Status Code** and respective error message will be helpful to understand and troubleshoot the issue.



Method POST URL https://10.100.1.72/api/4.0/edges/?isUniversal=true

Headers

Authorization: Basic YWRtaW46Vk1... Content-Type: application/xml

Body

```
<edge>
  <status>deployed</status>
  <datacenterMoid>datacenter-2</datacenterMoid>
```

```
Status Code       : 201 Created
Cache-Control     : no-cache
Content-Length    : 0
Date              : Tue, 26 Apr 2016 23:18:16 GMT
Location          : /api/4.0/edges/edge-fc9e7dal-d77f-4c8e-ba27-59cec5620ad5
```

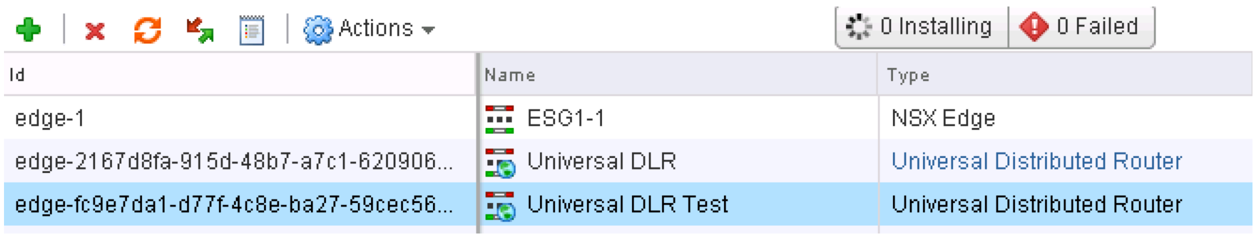
Figure 7: Using RESTClient on Firefox to Create a UDLR in NSX 6.2.2

Below is the body of the REST API call shown in Figure 7 used to create a UDLR. This is entered within the **Body** field of the RESTClient in Figure 7. Note, parameters can be adjusted based on requirements.

```
<edge>
    <status>deployed</status>
    <datacenterMoid>datacenter-2</datacenterMoid>
    <datacenterName>PaloAlto</datacenterName>
    <tenant>default</tenant>
    <name>Universal DLR Test</name>
    <fqdn>NSX-edge-Test</fqdn>
    <enableAesni>false</enableAesni>
    <enableFips>false</enableFips>
    <vseLogLevel>info</vseLogLevel>
    <appliances>
      <applianceSize>compact</applianceSize>
      <appliance>
        <highAvailabilityIndex>0</highAvailabilityIndex>
        <vcUuid>5013a4d4-9fa5-d86c-2166-3df1636e7d5a</vcUuid>
        <resourcePoolId>domain-c22</resourcePoolId>
        <resourcePoolName>Edge Cluster</resourcePoolName>
        <datastoreId>datastore-38</datastoreId>
        <datastoreName>EMC_VNX_1-1</datastoreName>
        <hostId>host-32</hostId>
        <hostName>10.100.1.72</hostName>
        <vmFolderId>group-v3</vmFolderId>
        <vmFolderName>vm</vmFolderName>
        <vmHostname>NSX-edge-Test</vmHostname>
        <vmName>Universal DLR</vmName>
        <deployed>true</deployed>
      </appliance>

      <deployAppliances>true</deployAppliances>
    </appliances>
    <cliSettings>
      <remoteAccess>true</remoteAccess>
      <userName>admin</userName>
      <password>VMwareVMware1!</password>
      <passwordExpiry>99999</passwordExpiry>
    </cliSettings>

    <autoConfiguration>
      <enabled>true</enabled>
      <rulePriority>high</rulePriority>
    </autoConfiguration>
    <type>distributedRouter</type>
    <isUniversal>true</isUniversal>
    <universalVersion>21</universalVersion>
    <mgmtInterface>
      <label>vNic_0</label>
      <name>mgmtInterface</name>
      <addressGroups/>
      <mtu>1500</mtu>
      <index>0</index>
      <connectedToId>universalwire-11</connectedToId>
      <connectedToName>Universal HA</connectedToName>
    </mgmtInterface>

    <localEgressEnabled>false</localEgressEnabled>
    </edge>
```

Figure 8: XML Data Structure Returned After RESTClinet API Call

Once the **Send** button is clicked, in NSX, the new UDLR can be observed as shown below in Figure 9.



Id	Name	Type
edge-1	ESG1-1	NSX Edge
edge-2167d8fa-915d-48b7-a7c1-620906...	Universal DLR	Universal Distributed Router
edge-fc9e7da1-d77f-4c8e-ba27-59cec56...	Universal DLR Test	Universal Distributed Router

Figure 9: Verification on the UDLR Creation in the NSX Manager GUI

Programming Languages with NSX Rest API

This section demonstrates how the NSX Rest API can be leveraged via different programming languages to further automate tasks and create workflows/programs with advanced logic.

Why Use Programming Languages with NSX REST API?

Leveraging programming languages with the NSX REST API goes a step beyond simply calling NSX REST API calls via REST Client tools. Programming languages such as Python or Perl allow for introducing custom advanced logic and workflows.

Using conditional constructs available in all scripting/programming languages such as if/then statements and control flow logic such as looping mechanisms and boolean constructs, advanced programs and custom solutions can be created.

Further, an organization may already have a custom portal for managing/monitoring their infrastructure. By leveraging the NSX REST API with respective programming languages, organizations can build NSX support into their custom management portals or even create a new custom cloud management platform.

Below, a basic concept and architecture discussion of how programming languages can be utilized with NSX REST API is provided. In later sections, the programming languages Python, Perl, and Go are briefly discussed and examples provided.

Basic Concept and Architecture

As mentioned prior, NSX provides a REST API interface via NSX Manager, and any REST Client can utilize this REST API interface provided by NSX Manager. All popular programming languages have REST Client libraries/methods that can be utilized for REST operations discussed prior such as POST, GET, PUT, and DELETE. The respective REST Client library is typically imported into a program allowing accessibility to its REST Client methods. The below table provides examples of popular REST Client libraries for several common programming languages.

Programming Language	REST Client Library
Python	“requests” is the most popular library and one of the simplest to consume; some other libraries that can also be utilized include “siesta”, “httplib”, “httplib2”, “urlib”
PowerShell	“Invoke-RestMethod” cmdlet
Perl	“REST:Client” is the most popular library; “LWP::UserAgent” can also be used
Java	Several libraries that can be leveraged include: “Unirest”, “Javalite HTTP”, “Apache HttpClient”, “Jersey”
Go	“net/http” is the most popular library; some other libraries that can also be utilized include “sling”, “resty”, “napping”, “grequests”

Table 2: REST API Client Libraries for Common Programming Languages

Simple Scripts

Many simple utility scripts composed of one file may be written as shown in Figure 10 below. The script is short and steps are sequential in nature within the file. This may be adequate for such tasks as returning the NSX Distributed Firewall (DFW) to default state. This script serves its purpose for resetting the DFW to default state in-case one accidentally blocks vCenter communication and can no longer access the GUI to make the change.

In some programming languages this would consist of less than ten lines of code. A generalized example layout below shows what this may look like using any programming language

```
Simple Generalized Program


Program Variables here such as NSX Manager URL, Username, and Password


Main body of script calling NSX REST API to delete DFW section and reset to default
```

Figure 10: Basic Script/Program Structure

However, as programs get more advanced or additional functionality is added, it soon becomes clear splitting the program up into compartments or smaller functional units becomes helpful not only for management but also for rapid development, troubleshooting, and code reuse. In such cases the desire is to make the programs more modular. The next section discusses how such programs can be made more modular.

Functions and Modular Programs

Assume additional functionality needs to be added to the prior example of returning the NSX Distributed Firewall (DFW) to default state. New requirements now include the following:

- display a warning message that the NSX DFW will be reset to a default state
- user must acknowledge to proceed by clicking **y**
- some new DFW rules must be added automatically to represent custom default state

From the above requirements we can compartmentalize these into three tasks:

1. Display a warning message and confirm acknowledgement
2. Delete DFW rules and put DFW into default state
3. Create new DFW rules to represent custom default state

Each of these identified tasks can be a separate function. A function in programming languages is a specific reusable section of a code that can be called at any time to perform a specific task. If needed, functions can also be written to accept arguments that are used by the logic within the function to achieve needed results. Additionally, functions can also be written to return values once the respective function is complete; a value can consist of perhaps a value representing status or some other specific criteria.

So, instead of writing all logic sequentially within a file, specific functions can be created as separate blocks of code and be called from within the program. These functions can be within the same file or even in a separate file and imported in. Such a workflow would look like the below.

```
Simple Generalized Program


displayWarning() - Function for displaying warning message


deleteDfwSection() - Function for deleting DFW rules (putting DFW into default state)


addCustomDfwRules() - Function for adding custom default DFW rules


Main body of script calling respective functions above in sequence
displayWarning()
deleteDfwSection()
addCustomDfwRules()
```

Figure 11: Basic Script/Program Structure Using Functions

To further organize code, enable code reuse, and allow for faster deployment of applications custom libraries of code can be created. These libraries typically group together specific features/functionality that are related such as file manipulation or Internet connectivity. The next section discusses libraries in more detail.

Libraries

Programming languages will include a standard library out of the box for common tasks such as manipulating files, accessing operating system services, Internet connectivity, etc. Many additional external libraries exist, and custom libraries can be created as needed. Continuing with the example displayed above, below all custom functions prior created for modifying DFW are placed into an external custom library called **CustomDFW** which is then imported into the program to be consumed. **CustomDFW** represents the custom library created providing respective DFW functions.

```
Simple Generalized Program


import CustomDFW - Custom DFW Library


displayWarning() - Function for displaying warning message


deleteDfwSection() - Function for deleting DFW rules (putting DFW into default state)


addCustomDfwRules() - Function for adding custom default DFW rules
```

Figure 12: Basic Script/Program Library Structure

The below generalized program in Figure 13 imports the **CustomDFW** library and calls respective functions. It can be seen how creating libraries enables a clean organized structure which allows for ease of code development, code maintenance, and code reuse.

```
Simple Generalized Program


import CustomDFW


Main body of script calling respective functions from imported library
displayWarning()
deleteDfwSection()
addCustomDfwRules()
```

Figure 13: Basic Script/Program Using Libraries

In the next few sections we'll take a look at some specific programming languages and respective examples.

Python

Python is a popular open-source, high-level programming language. Its popularity is in part due to its ease of use, readability, low learning curve, rapid deployment nature, cross-platform capabilities, and library support. As such, it has become one of the most popular programming languages and a favorite among network admins and devops engineers.

A focus of Python is both rapid and ease of development. For example, to allow for rapid and ease of development, Python is dynamically typed meaning that Python allows the user to assign different object types to the same variable without having to declare that variable of a certain type first. This allows quickly creating data constructs such as lists containing object of different types. Python is also very readable and easy to troubleshoot because it does not require semicolons to terminate statements or use braces to designate blocks of code like other programming languages. Instead indentation is used to designate specific blocks of code.

Figure 14 below demonstrates how indentation is used with Python to structure blocks of code.

```
First line of Python code

Loop 1
    Loop line 1
    Loop line 2

Fifth line of Python code

If this is true
    Do this
Else
    Do this

Last line of Python code
```

Figure 14: Python Code Structure

Additionally, Python is a very expressive language in the sense that the same task can usually be done in Python in less lines of code than other languages. Python also supports object oriented features such as user-defined classes and inheritance where objects can inherit from other parent objects.

Further, Python's a cross-platform language able to run on multiple platforms such as MAC, Unix/Linux, and Windows. Python is an interpreted language, and, as such, can run on any platform that has a Python interpreter; this allows for its cross-platform capabilities.

Python also has a vast assortment of natively included and externally importable libraries; this enables rapid development of solutions as it allows programmers to reuse efficiently developed code within their custom projects. It's important to note that when using the popular REST Client **requests** library with Python as shown in the examples further below, the **requests** library must be manually installed as it is not part of the Python standard library. Example code below is written in Python 2.7.7 and the Python package management system, **pip**, was used to install the **requests** package from the cli with the following command: **pip install requests.**

Another important detail to note about Python is that there are two lines of versioning available: Python 2.x and Python 3.x. Python 2.x is the traditional Python that simply builds upon earlier versions of Python without breaking backward compatibility. Python 3.x is the first version of Python to break backward compatibility with older Python versions; the reason for this was simply to improve some of the traditional Python language syntax and details for consistency and to evolve the language in a way to make it even more easy to understand, use, and build upon.

Whether to use Python 2.x or Python 3.x is a choice based on preference. NSX REST API can be used with either one. However, as Python 2.x has been around longer, there are likely Python 2.x libraries that have not been converted over to Python 3.x. If one of these libraries are needed for respective development purposes or the program needs to be merged into a larger existing Python 2.x program, Python 2.x can be used. Most popular Python libraries have been converted over to Python 3.x and as such Python 3.x would be a good choice for new projects.

To learn more about Python, see the official Python website.

Example: Automating Security Group and Policy Creation

Imagine creating a security posture and having to identify

1. The appropriate workloads
2. What security policies need to be applied to respective identified workloads

With NSX, a user can create what is known as **Security Groups**. Security Groups can easily be defined via the NSX Service Composer or via the Grouping Objects section (**NSX Managers → [Select NSX Manager] → Manage → Grouping Objects → Security Groups**) and allows for a unique model in consuming network and security services.

Objects such as virtual machines (VMs) across which similar policies must be applied can be defined inside a Security Group, and a consistent policy can then easily be mapped across all the respective objects. Furthermore, objects can be dynamically included within Security Groups and the security policies leveraging these Security Groups can then be configured based on the built-in VMware NSX features or by those offered by 3rd party solutions.

Assume the tasks at hand are:

1. Identify workload traffic and create security groups to dynamically include the relevant workloads. This will help determine what security policies should be applied to the specific workloads based on requirements

2. Create security policies leveraging created Security Groups

With NSX REST API, we can automate all of these steps and avoid a manual and time intensive process. In this example, the automatic creation of security groups with dynamic matching criteria will be demonstrated. Figure 15 below displays the NSX lab setup for this example.

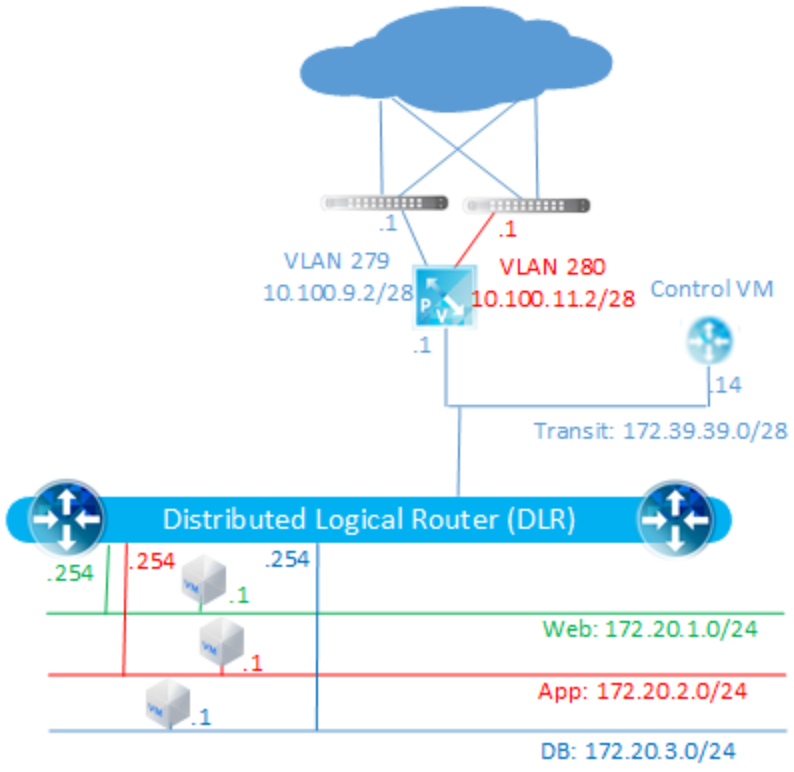
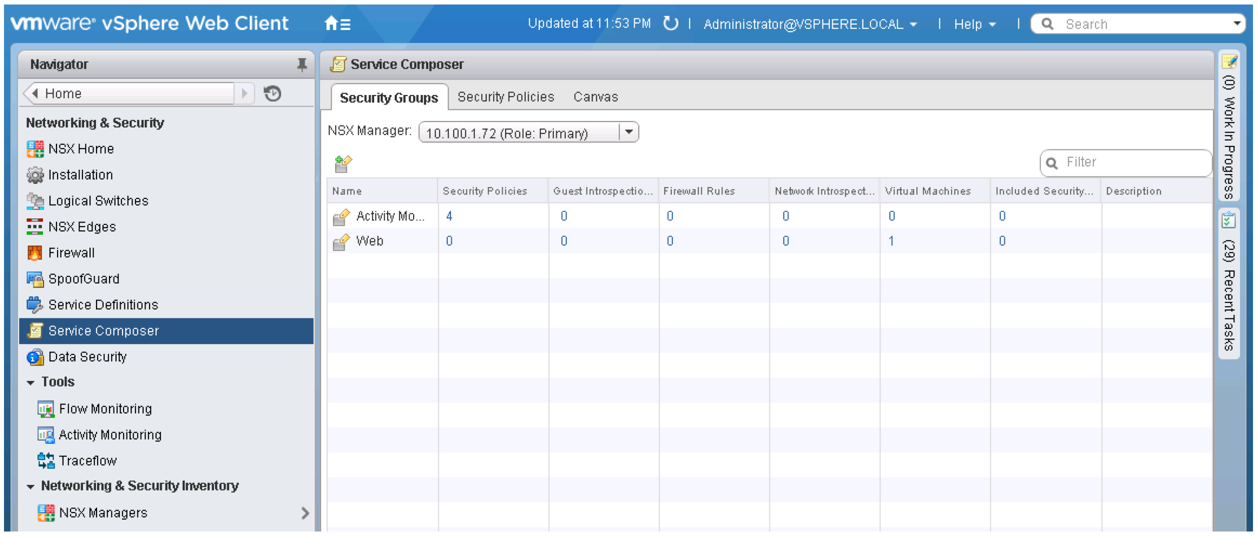


Figure 15: Example NSX Lab Setup

Figure 16 below displays a screenshot from NSX 6.2.2 where it shows a Web Security Group already created with one dynamically identified VM that's included as part of the Security Group; this correlates to the diagram above in Figure 15. It should be noted here that the Web Security Group is only being used locally on a local logical switch. The screenshot below shows the NSX Manager role as **Primary** because the deployment is part of a Cross-VC NSX deployment, but in this case only local objects and no universal objects are being used. To learn more about Cross-VC NSX and universal objects, see the following blog post on the VMware NSX Network Virtualization Blog : Cross-VC NSX for Multi-site Solutions.

The Security Groups and policies discussed in this example are for local objects within one vCenter domain.



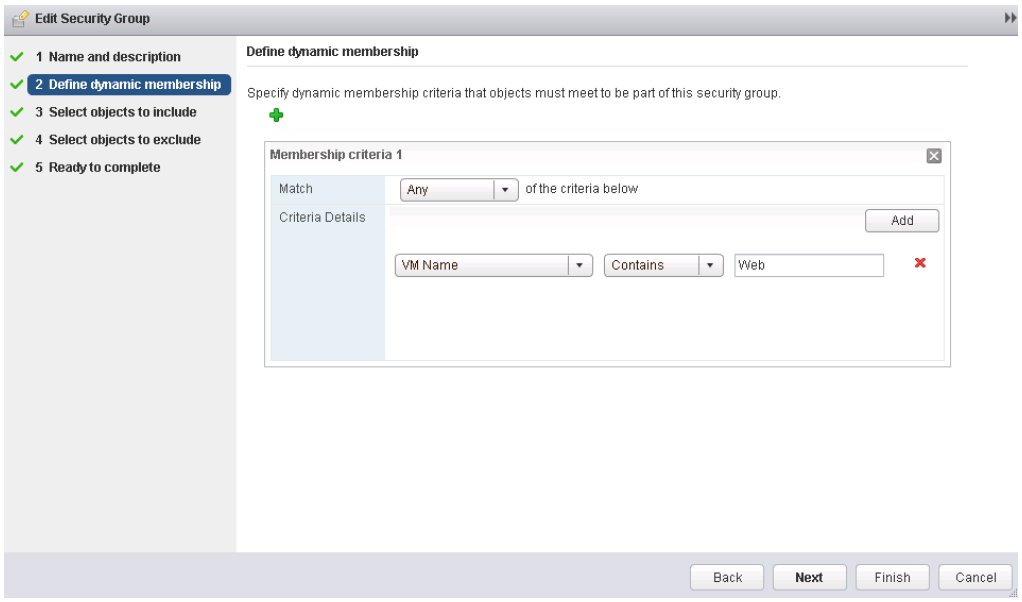
NSX Manager: 10.100.1.72 (Role: Primary)

Name	Security Policies	Guest Introspectio...	Firewall Rules	Network Introspect...	Virtual Machines	Included Security...	Description
Activity Mo...	4	0	0	0	0	0	
Web	0	0	0	0	1	0	

Figure 16: Existing Security Groups in Setup

A Security Group can be manually created as shown below where dynamic matching membership criteria is being defined. Imagine creating 100+ or even a 1000+ Security Groups manually.

Figure 17 below shows the dynamic membership criteria that an object must meet to be part of the **Web** Security Group. In this case the VM just has to contain the word **Web** in the VM Name.


Edit Security Group
1 Name and description
2 Define dynamic membership
3 Select objects to include
4 Select objects to exclude
5 Ready to complete
Define dynamic membership
Membership criteria 1
Add
Back
Next
Finish
Cancel

Specify dynamic membership criteria that objects must meet to be part of this security group.

Match: Any of the criteria below

Criteria Details: VM Name | Contains | Web

Figure 17: Using NSX Manager GUI to Create a Security Group

Currently, from Figure 16, it can be seen that the **Web** Security Group has been created yet the Security Group is not yet utilized in any security rules/policies. The procedural steps for creating the security posture in this case are:

1. Identify workload traffic

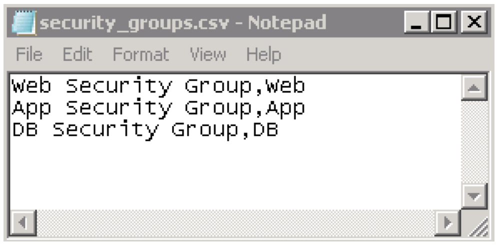
2. Create Security Groups to dynamically include workloads that should be included as part of the Security Group. This will help determine what security policies should be applied to the specific workloads based on requirements.

3. Create security policies leveraging created Security Groups

Let's apply this scenario to a larger scale; depending on specific environment and workloads, there can potentially be many Security Groups and respective security policies that need to be created. Let's assume the end user has 100+ Security Groups that need to be created. It can become tedious if one has to create so many Security Groups manually. One method of automation can be to leverage the NSX REST API and a bit of code. In this case we'll demonstrate with python.

For our example, first, we create a CSV file with one Security Group per row. Each row has two fields; the first field is the Security Group name, and, the second field is the matching criteria, or, in this case, specifically the word to match in the VM name (meaning, if the VM name contains this word, the VM will meet the criteria to dynamically be added to respective Security Group). The fields per row are separated via a comma within a CSV file.

Figure18 below shows a screenshot of the file used as input to the python script in this example. Note, we demonstrate using just three Security Groups here based on our setup but many more as needed can be added and created with this batch-type Security Group creation script.



```
Web Security Group,Web
App Security Group,App
DB Security Group,DB
```

Figure 18: CSV File Used as Input to Python Script

By running the below python script, which reads in the CSV file as input and leverages the respective NSX REST API calls, all the Security Groups in the CSV file with the respective dynamic matching criteria are created.

First, the needed python libraries are imported, python variables used for the script are set, and a reader object is created to iterate over the lines in the CSV file. The python requests library is leveraged for the NSX REST API call and the python csv library is leveraged for reading and parsing the CSV input file. It's important to note that the requests library is not part of the Python standard library and must be installed. The below code is written in Python 2.7.7 and the Python package management system, **pip**, was used to install the requests package from the cli with the following command: **pip install requests**.

Next, the python script iterates through the CSV file data via a loop reading the CSV data line by line and the respective NSX REST API to create the Security Group is called. The **security_group** python list variable contains two items, **security_group[0]**, which contains the Security Group name, and, **security_group[1]**, which contains the Security Group matching criteria for dynamic inclusion as discussed prior. For each iteration through the list, the NSX REST API call is made to create the respective Security Group.

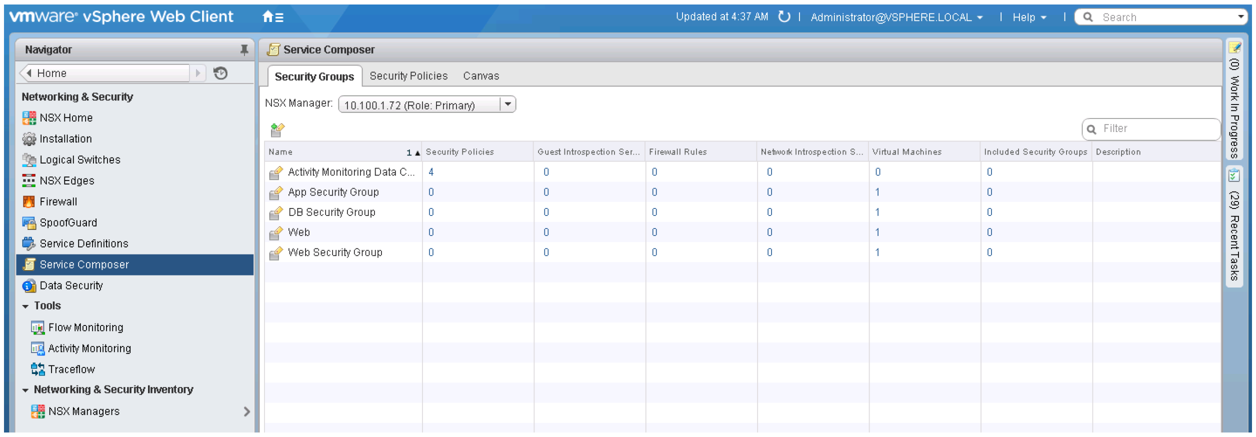
The full python code is displayed below and was run in this example with NSX 6.2.2. The Python script is executed from the cli with the **python create_security_groups.py** command. The script could be run in Unix/Linux or Windows as long as the Python interpreter and respective **requests** library is installed.

```python
#Script: create_security_groups.py
#
#Description: Script reads input from a CSV file and creates Security Groups with
#dynamic inclusion criteria. Used with NSX 6.2.2 and Python 2.7.11.


import csv         #library used for reading and parsing data from CSV file
import requests    #library used for making REST API calls

#ignore specific warnings
from requests.packages.urllib3.exceptions import InsecureRequestWarning
requests.packages.urllib3.disable_warnings(InsecureRequestWarning)

#initialize variables with needed info for input file and to make NSX REST API call
nsx_username = "nsxadmin"
nsx_password = "notMyPassword!"
nsx_url = "https://10.100.1.72/api/2.0/services/securitygroup/bulk/globalroot-0"
csv_filename = "security_groups.csv"
myheaders={'content-type':'application/xml'}


try:
    csv_file = open(cv_filename)        #open CSV file for reading
except FileNotFoundError as e:
    print "Input CSV file not found!"
      `
try:
    csv_data = csv.reader(csv_file)     #get reader object to iterate over lines in CSV file
except IOError as e:
    print "Error reading CSV file!"

for security_group in cv_data:          #loop through parsed lines read in from CSV file
    #print security_group[0]    #uncomment for debugging - contains Security Group name

    #print security_group[1]    #uncomment for debugging - contains Security Group word to match in VM name

    #create XML payload with Security Group data read in from CSV file
```

```
        payload ='''
        <securitygroup>
          <objectId>securitygroup-1</objectId>
          <objectTypeName>SecurityGroup</objectTypeName>
          <vsmUuid>422B2F08-25CF-AE4D-D78E-9450BA33618F</vsmUuid>
          <nodeId>d4fa9b9f-7973-4971-9ae6-c8f7e693ab30</nodeId>
          <revision>1</revision>
          <type>
            <typeName>SecurityGroup</typeName>
          </type>
          <name>''' +  security_group[0] +  '''</name>
          <description>
          </description>
          <scope>
            <id>globalroot-0</id>
            <objectTypeName>GlobalRoot</objectTypeName>
            <name>Global</name>
          </scope>
          <clientHandle>
          </clientHandle>
          <extendedAttributes/>
          <isUniversal>false</isUniversal>
          <universalRevision>0</universalRevision>
          <inheritanceAllowed>false</inheritanceAllowed>
          <dynamicMemberDefinition>
            <dynamicSet>
              <operator>OR</operator>
              <dynamicCriteria>
                <operator>OR</operator>
                <key>VM.NAME</key>
                <criteria>contains</criteria>
                <value>''' +  security_group[1] +  '''</value>
                <isValid>true</isValid>
              </dynamicCriteria>
            </dynamicSet>
          </dynamicMemberDefinition>
        </securitygroup>'''

        #print payload    #uncomment this for debugging - payload for REST API request call


        #call NSX REST API to create Security Group with XML payload just created
        try:
            response = requests.post(nsx_url, data=payload, headers=myheaders, auth=(nsx_username,nsx_password), verify=False)
        except requests.exceptions.ConnectionError as e:
            print "Connection error!"

        print response.text
```

Figure 19: Python Script Creates Security Groups Based on Input From CSV File

Figure 20 below shows the results within the NSX GUI. As expected, based on the diagram in Figure 15, three Security Groups have been created each with one member VM. Note, the prior **Web** Security Group is still present which displays the same number of dynamically identified VMs as the **Web Security Group** created via python script, thus further validating the results.



NSX Manager: 10.100.1.72 (Role: Primary)

Name	Security Policies	Guest Introspection Ser...	Firewall Rules	Network Introspection S...	Virtual Machines	Included Security Groups	Description
Activity Monitoring Data C...	4	0	0	0	0	0	
App Security Group	0	0	0	0	1	0	
DB Security Group	0	0	0	0	1	0	
Web	0	0	0	0	1	0	
Web Security Group	0	0	0	0	1	0	

Figure 20: Three Additional Security Groups Created

Perl

Perl is also quite a popular open-source programming language. One of Perl's key strengths is the vast amount of Perl modules/libraries available via the Comprehensive Perl Archive Network (CPAN). There is also a CPAN module included with Perl which is used to download and install Perl modules automatically from the CPAN repository.

Perl is a much older language than Python and thus has a larger amount of external libraries available. It's important to note that when using the popular REST Client **REST:Client** module with Perl as shown in the examples further below, it must be manually installed as it is not part of the Perl standard library. Example code below is written in Perl 5.24.0.1, and, the Perl CPAN module/command was used to install the **REST:Client** package from the cli with the following command: **cpan install REST:Client.** The **Data::Validate::IP library**, used to validate IPv4 and IPv6 addresses, is also not part of the Perl standard library and was also installed via CPAN.

Many beginning programmers also find Python easier to learn than Perl. Perl is more traditional in the sense that like other programming languages, outside of Python, it uses braces to designate blocks of code like loops, conditional statements, and functions. It also uses semicolons to designate end of statements. Perl is a powerful and very flexible programming language in the sense that there are many ways to accomplish the same task.

Perl also supports object oriented features such as user-defined classes and inheritance where objects can inherit from other parent objects.

Similar to Python, Perl's a cross-platform language able to run on multiple platforms such as MAC, Unix/Linux, and Windows. Perl is an interpreted language, and, as such, can run on any platform that has a Perl interpreter; this allows for its cross-platform capabilities.

Figure 21 below demonstrates how braces and semicolons are used with Perl to structure blocks of code.

```
First line of Perl code;

Loop 1
{
    Loop line 1;
    Loop line 2;
}
Seventh line of Perl code;

If this is true
{
    Do this;
}
Else
{
    Do this;
}

Last line of Perl code;
```

Figure 21: Perl Code Structure

To learn more about Perl, see the Perl Programming Language Website.

Example: Monitoring VMware NSX SpoofGuard with REST API and Perl

In this example, the program does a query to get Virtual NICs that have the SpoofGuard state of **Active**. SpoofGuard is a feature that can prevent the spoofing of IP addresses by trusting an IP address of a VM upon first use. In this state the IP address will be Active. If the IP address is later changed, communication will be blocked unless the IP address change is manually approved. Figure 22 below displays the NSX lab setup used in this example.

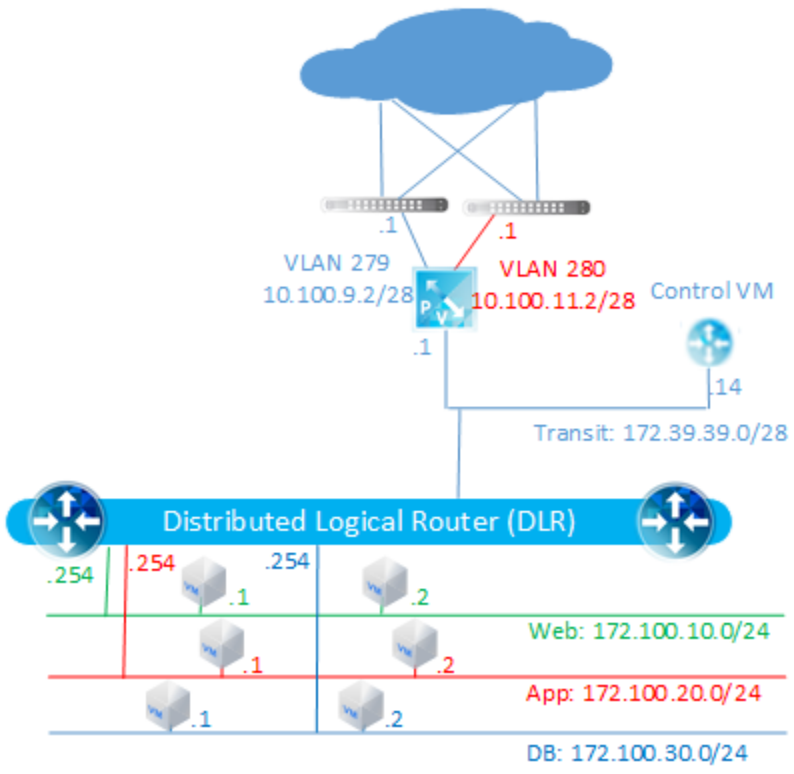


Figure 22: Lab Setup

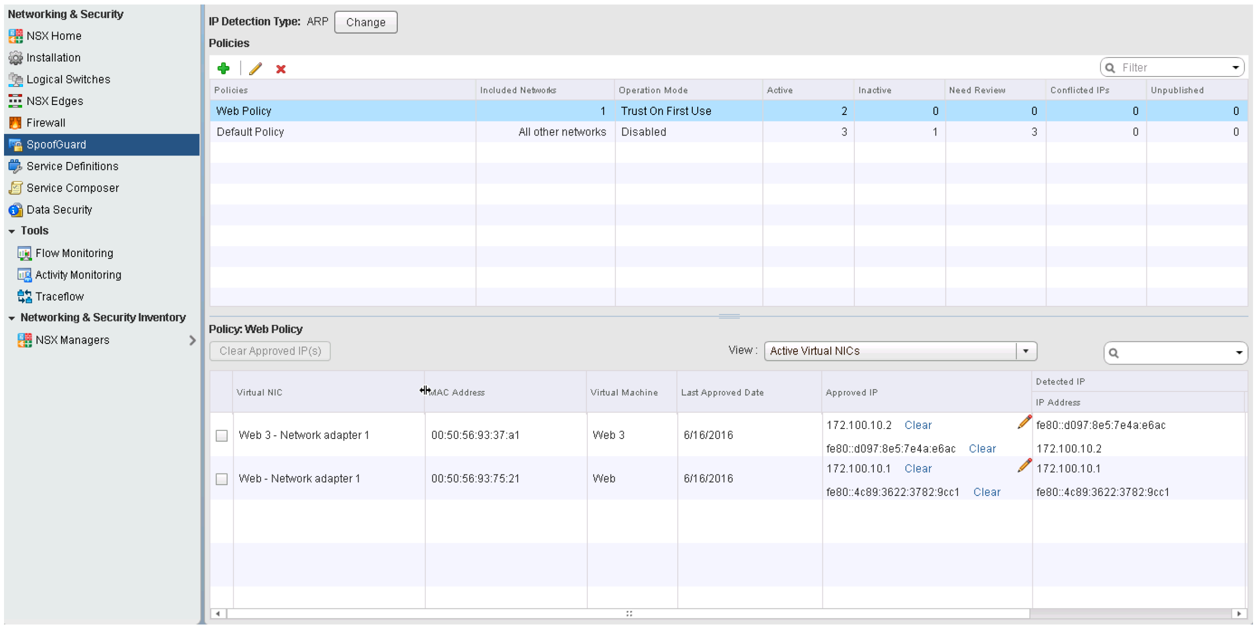
The complete Perl code is shown further below; the script runs against an environment where each VM has one vNIC with both an IPv4 and IPv6 address. The program returns all **Active** Virtual NICs and associated information in regards to where SpoofGuard is enabled. NSX 6.2.2 and Perl 5.24.0.1 were utilized in this example.

The **REST:Client** module/library is used to provide the REST Client capabilities to communicate with NSX Manager. The **XML:Simple** library is used to easily read the XML output returned from the NSX REST API calls. It can be observed from the code that different tasks within the program are broken into functions and the respective functions are called in sequence based on requirements within the program.

First, the respective needed libraries are imported into the program and respective variables to connect to NSX Manager are initialized. Next a function call is made to obtain a REST Client object. Once the REST Client object is obtained, another function call is made to call the respective NSX REST API to get **Active** IP addresses and associated information in regards to where SpoofGuard is enabled.

Within the **getXMLData** function, once the respective data is returned in XML format from the NSX REST API call, the **XMLin** method of the **XML:Simple** object is used to read and parse the data. The resulting data is returned to the main program where it is looped through, and, for each entry/record, Nic Name, approved MAC address, approved IPv4 address, and approved IPv6 address is output. The full code is shown below in Figure 24.

Before running the Perl script, we confirm via the NSX GUI for SpoofGuard what the **Active** Virtual NICs are. The **Web Policy** SpoofGuard policy in Figure 23 below is enabled on the web-tier shown as the **Web** logical switch in Figure 22 above. Further, it can be seen that there are two entries, one for the **Web** VM with IP address **172.100.10.1** and another for the **Web 3** VM with IP address **172.100.10.2;** this correlates with the lab diagram in Figure 22. Since the SpoofGuard operation mode is set to **Trust On First Use**, both these respective VMs' IP addresses are automatically initially trusted/approved and considered **Active**.



Policies	Included Networks	Operation Mode	Active	Inactive	Need Review	Conflicted IPs	Unpublished
Web Policy	1	Trust On First Use	2	0	0	0	0
Default Policy	All other networks	Disabled	3	1	3	0	0

Virtual NIC	MAC Address	Virtual Machine	Last Approved Date	Approved IP	Detected IP IP Address
Web 3 - Network adapter 1	00:50:56:93:37:a1	Web 3	6/16/2016	172.100.10.2 Clear fe80::d097:8e5:7e4a:e6ac Clear	fe80::d097:8e5:7e4a:e6ac 172.100.10.2
Web - Network adapter 1	00:50:56:93:75:21	Web	6/16/2016	172.100.10.1 Clear fe80::4c89:3622:3782:9cc1 Clear	172.100.10.1 fe80::4c89:3622:3782:9cc1

Figure 23: NSX GUI Displaying SpoofGuard Active Virtual NICs

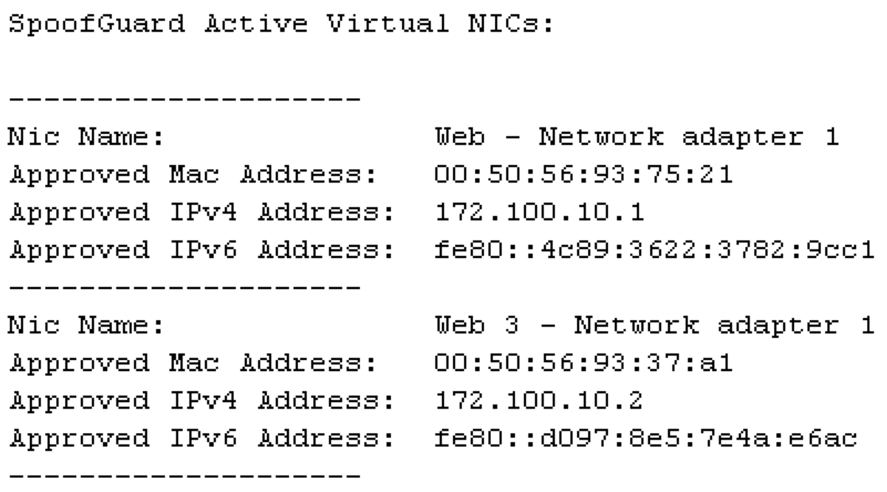
The Perl script is executed from the cli with the **perl spoofguard_active_ips.pl** command. The full code is shown below. The script could be run in Unix/Linux or Windows as long as the Perl interpreter and respective **REST:Client** and **Data::Validate::IP Perl** modules are installed.

```
#Script: spoofguard_active_ips.pl
#
#Description: Script runs against an environment where each VM has a single vNIC with
#both an IPv4 and IPv6 address. The program returns all "Active" IP addresses in
#relation to SpoofGuard and respective Nic Name, approved MAC Address, approved IPv4
#Address, and approved IPv6 Address. Used with NSX 6.2.2 and Perl 5.24.0.1.


use REST::Client;  #REST Client library used to make REST API calls
use MIME::Base64;  #library used for base64 encoding
use XML::Simple;   #library used for ease of reading XML
use Data::Validate::IP qw(is_ipv4 is_ipv6);  #library used to validate IPv4 and IPv6 
addresses


#uncomment below line if getting SSL connectivity/certificate error
#$ENV{"PERL_LWP_SSL_VERIFY_HOSTNAME"} = 0; #disable certificate check

#function to draw separator between records
sub drawSeparator
{
  print "--------------------\n";
  return;
}


#function to print heading
sub printHeader
{
  print "SpoofGuard Active Virtual NICs: \n\n";
  return;
}


#function creates and returns REST Client object
#takes NSX Manager URL with IP address as argument
sub getRestClient
{
       #create REST Client object
       my ($arg) = @_;

       my $restclient = REST::Client->new(host => $arg);
       return $restclient;
}


#function calls respective NSX REST API call and returns results
#takes REST Client object, NSX REST API call, base64 encoded
#username and password, and REST API 'content-type' header variable
#as arguments
sub getXMLData
{
       my ($client, $nsx_rest_spoofguard_active_api, $encoded_auth, $nsx_content_type) 
= @_;

       #make NSX REST API call to get "Active" Virtual NICs and respective info for 
spoofguard
       $client->GET($nsx_rest_spoofguard_active_api,
             {"Authorization" => "Basic $encoded_auth",
              "Accept" => $nsx_content_type});

       $xmlResponse = new XML::Simple; #create object for ease of reading XML

       #read in respective XML data from NSX API response using XMLin method
       $xmlData = $xmlResponse->XMLin($client->responseContent(), KeyAttr => 
['spoofguard']);

       return $xmlData
}


#initialize variables for connecting to NSX Manager and calling NSX REST API
```

```
    my $nsx_manager = "https://10.10.10.72";
    my $nsx_username = "admin";
    my $nsx_password = "notMyPassword!";
    my $nsx_rest_spoofguard_active_api = "/api/4.0/services/spoofguard/spoofguardpolicy-
    5?list=ACTIVE";
    my $nsx_content_type = "application/xml";

    #use Base64 ecoding for transmitting respective data
    my $encoded_auth = encode_base64("$nsx_username:$nsx_password");


    my $client = getRestClient($nsx_manager); #get REST Client object

    #get NSX REST API call result data
    my $xmlData = getXMLData($client, $nsx_rest_spoofguard_active_api, $encoded_auth,
    $nsx_content_type);

    printHeader();

    #loop through respective results and for each entry/record, print Nic Name,
    #approved MAC Address, approved IPv4 Address, and approved IPv6 Address
    foreach $xmlRecord (@{$xmlData->{'spoofguard'}})
    {
            drawSeparator();
            print "Nic Name: \t\t" . $xmlRecord->{'nicName'} . "\n";
            print "Approved Mac Address: \t" . $xmlRecord->{'approvedMacAddress'} . "\n";

            $ip_address_1 = $xmlRecord->{'approvedIpAddress'}->{'ipAddress'}->[0];
            $ip_address_2 = $xmlRecord->{'approvedIpAddress'}->{'ipAddress'}->[1];

            if(is_ipv4($ip_address_1))
            {
                    print "Approved IPv4 Address:\t" . $ip_address_1 . "\n";
                    print "Approved IPv6 Address:\t" . $ip_address_2 . "\n";
            }
            else
            {
                    print "Approved IPv4 Address:\t" . $ip_address_2 . "\n";
                    print "Approved IPv6 Address:\t" . $ip_address_1 . "\n";
            }
    }
    drawSeparator();


    #uncomment below lines to see response status and headers
    #print "Response status: " . $client->responseCode() . "\n";
    #foreach ($client->responseHeaders())
    #{
    #  print $_ . "=" . $client->responseHeader($_) . "\n";
    #}
```

Figure 24: Perl Script For Displaying "Active" IP Addresses for SpoofGuard

Once the above script is run with the **perl spoofguard_active_ips.pl** command at the cli, the below output is generated. As can be seen, the results match what is displayed in the GUI in Figure 23 confirming the results are correct.



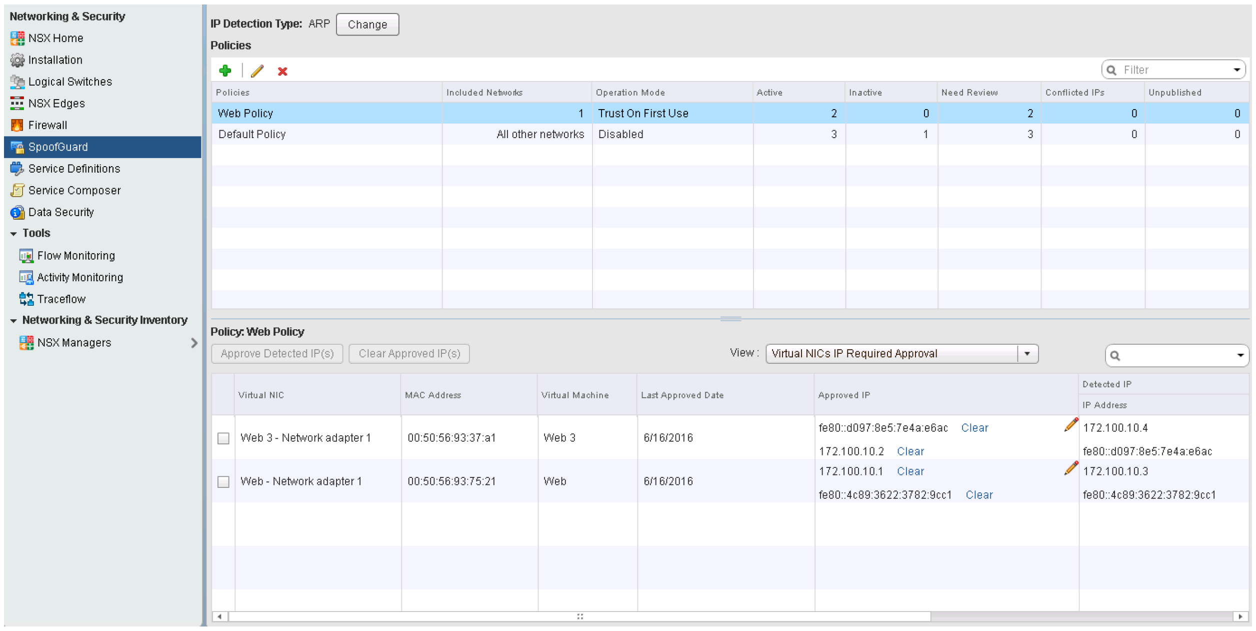
```
SpoofGuard Active Virtual NICs:

--------------------
Nic Name:                Web - Network adapter 1
Approved Mac Address:    00:50:56:93:75:21
Approved IPv4 Address:   172.100.10.1
Approved IPv6 Address:   fe80::4c89:3622:3782:9cc1
--------------------
Nic Name:                Web 3 - Network adapter 1
Approved Mac Address:    00:50:56:93:37:a1
Approved IPv4 Address:   172.100.10.2
Approved IPv6 Address:   fe80::d097:8e5:7e4a:e6ac
--------------------
```

Figure 25: Output of Perl Script Displaying "Active" IP Addresses for SpoofGuard

In this example, entries with the SpoofGuard state of **ACTIVE** are shown. To see entries with other SpoofGuard states, we can simply modify the key value pair at the end of the query. For example, to see entries with **REVIEW_PENDING** state, the **list=ACTIVE** key value pair at the end of the NSX REST API call can be changed to **list=REVIEW_PENDING.** In the code above, the **$nsx_rest_spoofguard_active_api** variable can be changed to **/api/4.0/services/spoofguard/spoofguardpolicy-5?list=REVIEW_PENDING.**

Note, for purposes of calling the NSX REST API, the default policy ID is always **spoofguardpolicy-1**. If custom policies have been created, the policy ID can be retrieved via REST API Get call with the following URL: **https://[NSX Manager IP Address]/api/4.0/services/spoofguard/policies/**.

In Figure 23, it can be seen there are no entries that need review for the Web Policy SpoofGuard policy as shown by the **0** under the **Need Review** column. For demonstration, the IP address of the **Web** VM with IP address **172.100.10.1** is changed to **172.100.10.3** and the IP address of **Web 3** VM with IP address **172.100.10.2** is changed to **172.100.10.4**. These IP address changes are detected by SpoofGuard. As shown in Figure 26 below, before the VMs are allowed to communicate using the new respective IP addresses, the changes must be approved. The column **Need Review** has also changed to **2** as expected. Selecting the **Virtual Nics IP Required Approval** option from the drop down box displays the entries needing review/approval. Note, the **Approved IPv4 Address** and **Detected IPv4 Address** for each of the two entries respectively is different.



IP Detection Type: ARP

Policies

Policies	Included Networks	Operation Mode	Active	Inactive	Need Review	Conflicted IPs	Unpublished
Web Policy	1	Trust On First Use	2	0	2	0	0
Default Policy	All other networks	Disabled	3	1	3	0	0

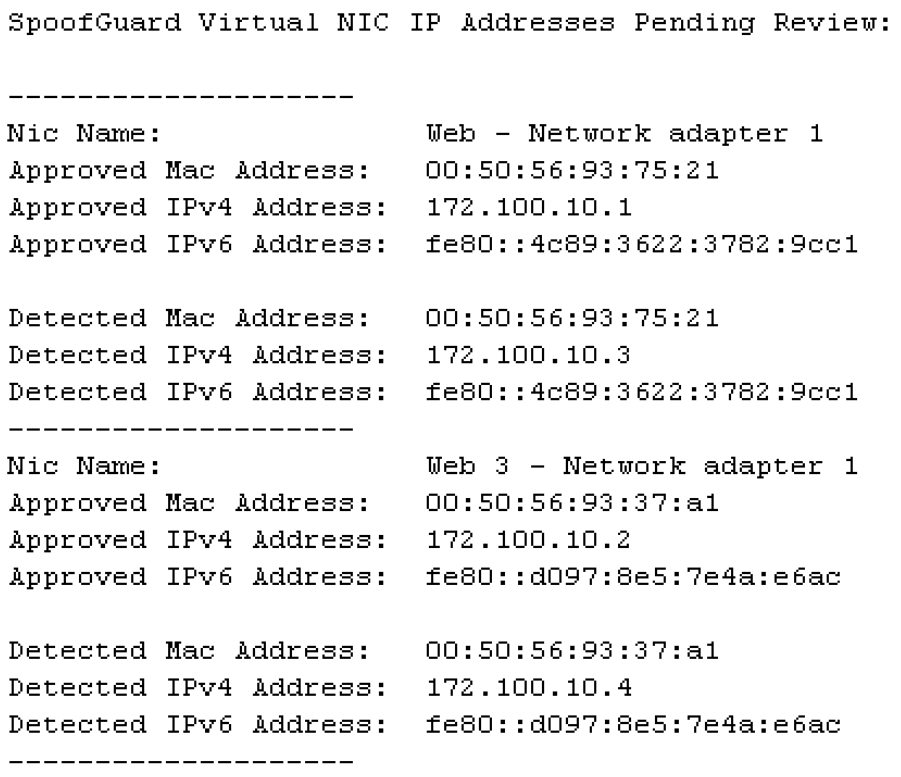
Policy: Web Policy

View : Virtual NICs IP Required Approval

Virtual NIC	MAC Address	Virtual Machine	Last Approved Date	Approved IP	Detected IP
					IP Address
Web 3 - Network adapter 1	00:50:56:93:37:a1	Web 3	6/16/2016	fe80::d097:8e5:7e4a:e6ac Clear 172.100.10.2 Clear	172.100.10.4 fe80::d097:8e5:7e4a:e6ac
Web - Network adapter 1	00:50:56:93:75:21	Web	6/16/2016	172.100.10.1 Clear fe80::4c89:3622:3782:9cc1 Clear	172.100.10.3 fe80::4c89:3622:3782:9cc1

Figure 26: Virtual NIC IP Address Changes Requiring Review/Approval

If in the Perl code, the **$nsx_rest_spoofguard_active_api** variable is changed to **/api/4.0/services/spoofguard/spoofguardpolicy-1?list=REVIEW_PENDING,** information can be displayed about IP Addresses that SpoofGuard has blocked and need review/approval**.** As can be seen in Figure 27 below, the correct results are displayed matching the results shown by the GUI in Figure 26. Note, additional fields and respective code was added to show the corresponding detected MAC and IP Addresses. These new fields correspond to the **detectedMacAddress** and **detectedIPAddress** data fields that can be obtained from the **$xmlRecord** variable within the script. The respective title/header for the results was also updated.



```
SpoofGuard Virtual NIC IP Addresses Pending Review:

--------------------
Nic Name:                Web - Network adapter 1
Approved Mac Address:    00:50:56:93:75:21
Approved IPv4 Address:   172.100.10.1
Approved IPv6 Address:   fe80::4c89:3622:3782:9cc1

Detected Mac Address:    00:50:56:93:75:21
Detected IPv4 Address:   172.100.10.3
Detected IPv6 Address:   fe80::4c89:3622:3782:9cc1
--------------------
Nic Name:                Web 3 - Network adapter 1
Approved Mac Address:    00:50:56:93:37:a1
Approved IPv4 Address:   172.100.10.2
Approved IPv6 Address:   fe80::d097:8e5:7e4a:e6ac

Detected Mac Address:    00:50:56:93:37:a1
Detected IPv4 Address:   172.100.10.4
Detected IPv6 Address:   fe80::d097:8e5:7e4a:e6ac
--------------------
```

Figure 17: Output of Perl Script – SpoofGuard IP Addresses Pending Review

Go

Go (also often referred to as Golang) is an open source, compiled, programming language developed by Google and many community contributors since 2007. It's distributed under a BSD-style license.

Go's mission is to "make easy to build simple, reliable and efficient software." It provides several packages with built-in functions that allow programmers to develop software quickly. For example, it provides HTTP and XML packages that can be used to interact with REST APIs very effectively. The Go language is rapidly gaining traction in the open source community and several popular projects (Docker, Terraform) are written in Go. Additional information about Go, as well as examples and tutorials can be found on the Go project's web site.

Go provides REST Client libraries that can be used to directly interact with the NSX REST API. There are also open source libraries that allow RAML to be parsed and used. This document is not meant to be a guide for the Go programming language, and a simple example is provided below that does not leverage RAML parsing.

Example: Creating NSX Services in Go

Services (or Applications) are used in NSX to define port numbers used in the Distributed Firewall rules. Once NSX is installed, a large number of built-in services for the most common use cases is already provided; however, users might be required to define new custom services.

Below, a simple program is shown written in Go that will create custom NSX Services using the NSX REST API, allowing the user to specify name, description, protocol (TCP, UDP, etc.) and value (a comma-separated list of ports and port ranges).

The program consists of code in a single **main** function that performs the following:

- Checks initial arguments to retrieve host name of the NSX Manager, credentials and an input file
- Parses the input file (semicolon separated list of entries) to retrieve the list of Services to create
- For each Service, an XML document is generated and sent over the NSX Manager API using the HTTP POST method
- Parses the return code, and, if successful, returns the NSX object ID of the created Service

The NSX REST API Guide, under section **Add Service**, specifies that all services must be created by executing an HTTP POST to the following URL:

```
https://nsxmgr-ip/api/2.0/services/application/scopeId
```

The guide also mentions that for all Services the **scopeId** parameter in the URL corresponds to the global scope (**globalroot-0**). The NSX API documentation also provides a detailed description of the XML structure that must be included in the HTTP request body to create the new Service:

```
<application>
 <objectId/>
 <type>
  <typeName/>
 </type>
 <description>(application description)</description>
 <name>(application name)</name>
 <revision>0</revision>
 <objectTypeName/>
 <element>
  <applicationProtocol>(application protocol)</applicationProtocol>
  <value>(port value(s))</value>
 </element>
</application>
```

Figure 28: XML Structure to Insert in the Request Body

Some of the parameters are empty or fixed, and the relevant ones are highlighted in red:

description, **name**, **applicationProtocol** and **value**. All these parameters are strings of text. As can be seen, this program does not do any validation of the inputs: if some illegal argument is specified on the command line or in the input file, it will be passed directly to the NSX API. If errors are encountered, NSX will return respective errors and the program will exit. Best practice is to add validation routines to such programs.

The following figure shows how the semicolon separated input file looks (comments begin with the **#** character):

```
# name, description, applicationProtocol, value
Test1;Test TCP Protcol;TCP;444
Test2;Test UDP Protocol;UDP;4871
Test3;Test TCP Protocol Range;TCP;181,1000-2000
Test4;Test Protocol Range;UDP;391,1821-3291
Test5;Test FTP;FTP;2121
```

Figure 29: Protocols Definition Input File

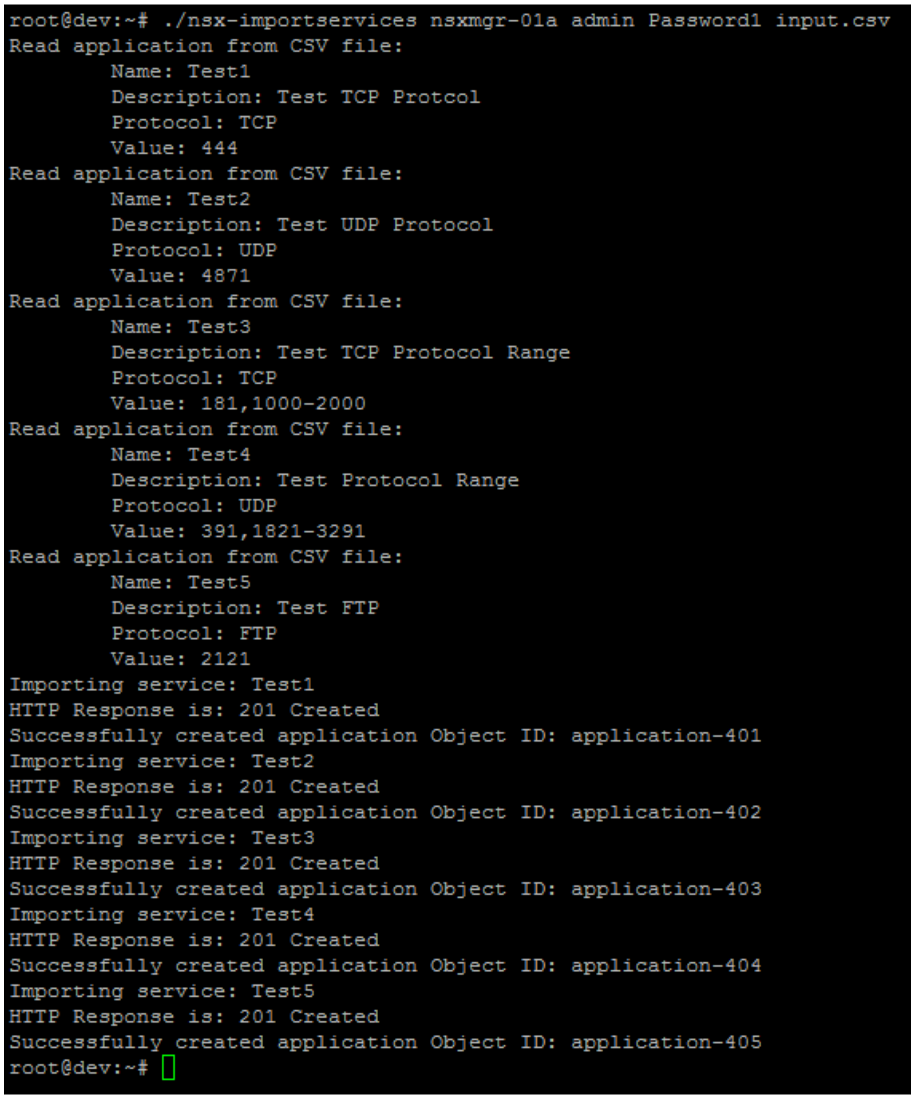
The program's Go code is presented below, and is compiled from the **nsx-importservices.go** source file. The reader is encouraged to refer to the comments in the code to better understand the program flow:

```go
package main

// Import required packages
import (
    "encoding/xml"
    "fmt"
    "os"
    "path/filepath"
    "net/http"
    "io/ioutil"
    "strings"
    "crypto/tls"
    "encoding/csv"
    "bytes"
)

// Define the NSX structure for the NSX service ("application") as a set of structs
// The desired XML format is as follows:
//<application>
// <objectId/>
// <type>
//  <typeName/>
// </type>
// <description>(application description)</description>
// <name>(application name)</name>
// <revision>0</revision>
// <objectTypeName/>
// <element>
//  <applicationProtocol>(application protocol)</applicationProtocol>
//  <value>(port value(s))</value>
// </element>
//</application>

type Application struct {
    XMLName xml.Name    `xml:"application"`
    ObjectId string `xml:"objectId"`
    Revision    int `xml:"revision"`
    Type    Type    `xml:"type"`
    Name    string  `xml:"name"`
    Description string  `xml:"description"`
    Element []Element   `xml:"element"`
}

type Type struct {
    XMLName xml.Name    `xml:"type"`
    TypeName    string  `xml:"typeName"`
}

type Scope struct {
    XMLName xml.Name    `xml:"scope"`
    Id  string  `xml:"id"`
    ObjectTypeName  string  `xml:"objectTypeName"`
    Name    string  `xml:"name"`
}

type Element struct {
    XMLName xml.Name    `xml:"element"`
    ApplicationProtocol string  `xml:"applicationProtocol"`
    Value   string  `xml:"value"`
}

func main() {

    // Check arguments and return error if incorrect
    // Note: there is no formal validation of the arguments in this example
    if(len(os.Args) != 5) {
        fmt.Printf("Syntax error\nUsage: %s [NSX Manager Address] [Username] [Password]
 [Input File]\n\n", os.Args[0])
        os.Exit(1)
    }
    nsxManager := os.Args[1]
    nsxUser := os.Args[2]
    nsxPassword := os.Args[3]
    inputFileName := os.Args[4]

    // Check the path and open the input file for read
    readFilePath, err := filepath.Abs(inputFileName)
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    csvfile, err := os.Open(readFilePath)
    if err != nil {
        fmt.Println(err)

        os.Exit(1)
    }
    // make sure the descriptor will be closed later
    defer csvfile.Close()

    // Read and parse the CSV file
    reader := csv.NewReader(csvfile)
    // Four fields are expected for each record, comments start with '#' and separator
is semicolon
    reader.FieldsPerRecord = 4
    reader.Comment = '#'
    reader.Comma = ';'
    rawCSVdata, err := reader.ReadAll()
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
     }


     // Parse the CSV data and add it a slice of Application types
     // Warning: there is no formal validation of the input in this example, all argumen
ts are passed to the NSX API
     var nsxServices []Application
     for _, chunk := range rawCSVdata {
         var x Application
         var e Element
         x.Name = chunk[0]
         x.Description = chunk[1]
         e.ApplicationProtocol = chunk[2]
         e.Value = chunk[3]
         x.Element = append(x.Element, e)
         x.Revision = 0

         fmt.Printf("Read application from CSV file:\n\tName: %s\n\tDescription: %s\n\tP
rotocol: %s\n\tValue: %s\n", chunk[0], chunk[1], chunk[2], chunk[3])
         nsxServices = append(nsxServices, x)
     }

     // Initialize the HTTPS client to skip SSL certificate verification
     tr := &http.Transport{ TLSClientConfig: &tls.Config{InsecureSkipVerify: true}}
     client := &http.Client{Transport: tr}

     // For each application read from the file, create an HTTPS POST request to the NSX
 Manager
     for _, srv := range nsxServices {

         // Encode the XML representation of this object in a buffer
         var buf bytes.Buffer
         xmlbuf := xml.NewEncoder(&buf)
         err = xmlbuf.Encode(&srv)
         if err != nil {
             os.Exit(1)
         }
         // Uncomment the following line to output the created XML body
         //fmt.Println("XML Body: " + buf.String())

         // Prepare the HTTPS POST request with the XML body
         fmt.Println("Importing service: " + srv.Name)
         req, err := http.NewRequest("POST", "https://" + nsxManager + "/api/2.0/service
s/application/globalroot-0", &buf )
         if err != nil {
             fmt.Println(err)
             os.Exit(1)
         }

         // Configure Basic authentication and Content-Type
         req.SetBasicAuth(nsxUser,nsxPassword)
         req.Header.Set("Content-Type", "application/xml")

         // Execute the HTTPS request
         resp, err := client.Do(req)
         if err != nil {
             fmt.Println(err)
             os.Exit(1)
         }
         defer resp.Body.Close()

         // Check the response type (expected 201 - Created)
         fmt.Println("HTTP Response is: " + resp.Status)
         if(resp.StatusCode != 201) {
             b, err := ioutil.ReadAll(resp.Body)
             if err != nil {
                 fmt.Println(err)
                 os.Exit(1)
             }
             fmt.Println("Error in executing query - got the following error:")
             fmt.Println(string(b))
             os.Exit(1)
         }

         // NSX returns the created application id as the last field of the Location header URL
         // Parse it and print it out
         loc, err := resp.Location()
         if err != nil {
             fmt.Println(err)
             os.Exit(1)
         }
         l := strings.Split(loc.String(),"/")
         fmt.Println("Successfully created application Object ID: " + l[len(l)-1])
     }
 }
```

Figure 30: Go Program to Create Custom NSX Services Leveraging the Relevant NSX REST API

In this example, we used XML structures to define the document hierarchy and leveraged the built-in XML encoder to generate the HTTP body. Users can also choose to build the HTTP body by simply concatenating strings with variables. There is no preferred option: the choice depends on the desired flexibility and reusability of the code.

The following picture shows the output of the program when run with the correct parameters:

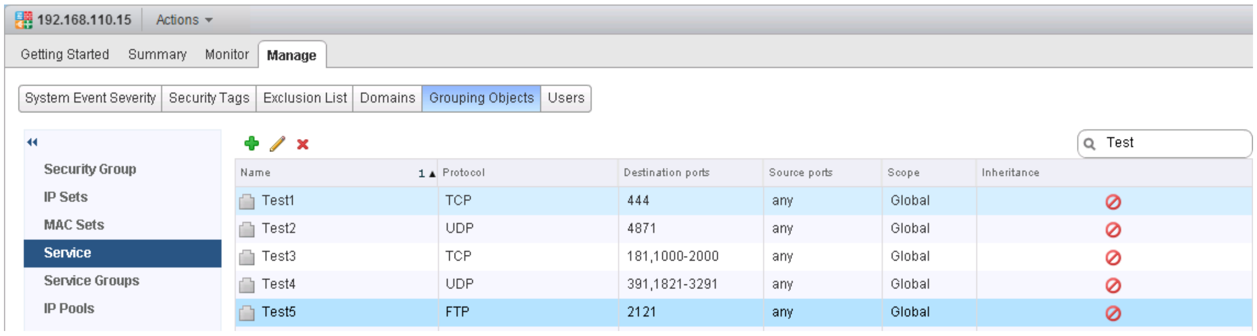


```
root@dev:~# ./nsx-importservices nsxmgr-01a admin Password1 input.csv
Read application from CSV file:
        Name: Test1
        Description: Test TCP Protcol
        Protocol: TCP
        Value: 444
Read application from CSV file:
        Name: Test2
        Description: Test UDP Protocol
        Protocol: UDP
        Value: 4871
Read application from CSV file:
        Name: Test3
        Description: Test TCP Protocol Range
        Protocol: TCP
        Value: 181,1000-2000
Read application from CSV file:
        Name: Test4
        Description: Test Protocol Range
        Protocol: UDP
        Value: 391,1821-3291
Read application from CSV file:
        Name: Test5
        Description: Test FTP
        Protocol: FTP
        Value: 2121
Importing service: Test1
HTTP Response is: 201 Created
Successfully created application Object ID: application-401
Importing service: Test2
HTTP Response is: 201 Created
Successfully created application Object ID: application-402
Importing service: Test3
HTTP Response is: 201 Created
Successfully created application Object ID: application-403
Importing service: Test4
HTTP Response is: 201 Created
Successfully created application Object ID: application-404
Importing service: Test5
HTTP Response is: 201 Created
Successfully created application Object ID: application-405
root@dev:~# 
```

Figure 31: Output After a Successful Run of the Go Program

After a successful run, the program outputs that the services were correctly created (HTTP Post returned **201 Created**). In the example shown above, the object IDs of the created services are ***application-401***, ***application-402***, ***application-403***, ***application-404*** and ***application-405***: these handlers can be used for subsequent CRUD operations.

Finally, it's possible to connect to the NSX Manager UI and observe that the new services were created according to the information specified in the input file; this is shown below.



Name	Protocol	Destination ports	Source ports	Scope	Inheritance
Test1	TCP	444	any	Global	⊘
Test2	UDP	4871	any	Global	⊘
Test3	TCP	181,1000-2000	any	Global	⊘
Test4	UDP	391,1821-3291	any	Global	⊘
Test5	FTP	2121	any	Global	⊘

Figure 32: NSX Services Created Using a Go program – NSX GUI verification

Automation Tools with NSX REST API

The previous sections documented how to consume the NSX REST API using clients such as a web browser, cURL, or via different programming languages. Administrators and engineers use such methods to build scripts that avoid the manual execution of repeatable tasks, and these methods are very effective. However, sometimes these methods can produce challenges in terms of management and maintenance.

To streamline the automation of tasks in a more structured manner, products, such as orchestrators and configuration management tools can be leveraged. With these solutions, users can define workflows or playbooks that include a list of tasks which will be automatically executed when some event occurs. Orchestrators and configuration management tools are often modular and highly configurable, allowing customers and partners to add additional capabilities to the product.

This document is not focused on discussing how these technologies work or how they differ from each other, but presents examples on how to leverage the NSX REST API using the respective tools. Two example products discussed in more detail below are VMware vRealize Orchestrator, a solution adopted by many VMware customers, and Ansible, an open source platform written in Python.

vRealize Orchestrator

VMware vRealize Orchestrator (vRO) is a powerful solution that simplifies the automation of complex IT tasks and integrates with several VMware and 3rd party components to extend service delivery and operational management. Given its flexibility and integration capabilities, many VMware customers use it for their automation strategy. It's packaged with vCenter Server, and, at the time of writing, does not require additional licenses.

This document is not meant to be a comprehensive guide on vRealize Orchestrator, but a few concepts below are important to mention:

- vRO is modular software and can be extended through Java plugins. VMware provides plugins to integrate it with many solutions, including vCenter and NSX
- vRO also includes a number of general purpose plugins that allow users to integrate with other solutions through several technologies, including AMQP, SSH, SQL, JDBC, PowerShell and HTTP REST APIs
- VMware Solution Partners ship plugins that can be imported in vRO and used to create workflows that involve the integration with 3rd party components. Available plugins can be downloaded from VMware's Solution Exchange web site.
- Users can build simple or complex workflows using vRO UI by simply concatenating different components. vRO also ships with an integrated Javascript parser that can be used to build the required logic and is easy to learn
- vRO workflows can be invoked manually, through APIs, by vRA, and AMQP message bus. The workflows can also be scheduled to run at specific times.

For additional details on vRealize Orchestrator please refer to the VMware vRealize Orchestrator web site. Figure 33 below displays a vRO workflow being built and leveraging NSX REST API.

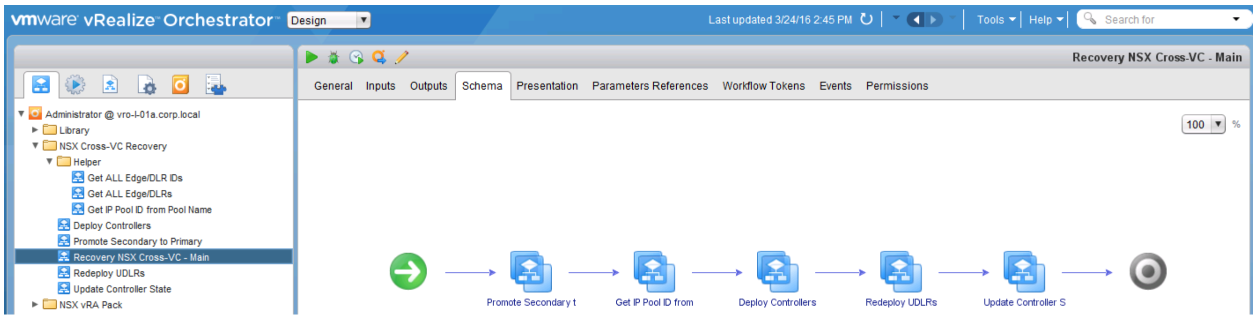


Figure 33: Building a vRO Workflow

vRO also provides a HTTP REST API client allowing for NSX REST API to be called directly from the tool; a NSX vRO Plugin is also available.

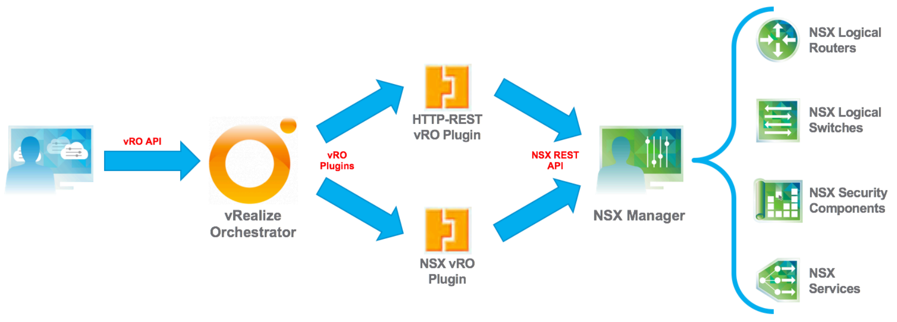
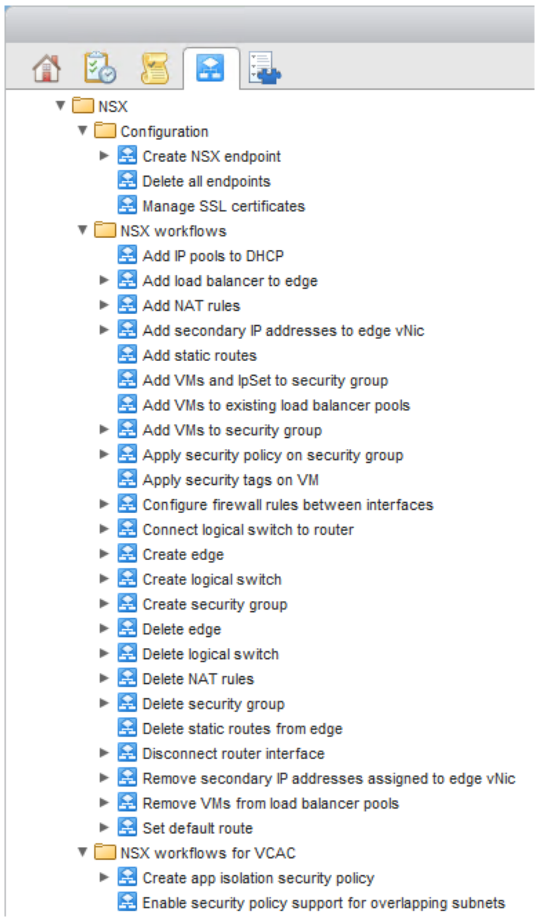


Figure 34: vRO Leveraging NSX REST API

As mentioned, VMware ships an NSX plugin that includes a number of built-in workflows that allow basic consumption of NSX features without requiring knowledge of the NSX API. The below screenshot shows the existing workflows available in the NSX vRO plugin at the time of writing.



- NSX
  - Configuration
    - Create NSX endpoint
    - Delete all endpoints
    - Manage SSL certificates
  - NSX workflows
    - Add IP pools to DHCP
    - Add load balancer to edge
    - Add NAT rules
    - Add secondary IP addresses to edge vNic
    - Add static routes
    - Add VMs and IpSet to security group
    - Add VMs to existing load balancer pools
    - Add VMs to security group
    - Apply security policy on security group
    - Apply security tags on VM
    - Configure firewall rules between interfaces
    - Connect logical switch to router
    - Create edge
    - Create logical switch
    - Create security group
    - Delete edge
    - Delete logical switch
    - Delete NAT rules
    - Delete security group
    - Delete static routes from edge
    - Disconnect router interface
    - Remove secondary IP addresses assigned to edge vNic
    - Remove VMs from load balancer pools
    - Set default route
  - NSX workflows for VCAC
    - Create app isolation security policy
    - Enable security policy support for overlapping subnets

Figure 35: NSX vRO Plugin Workflows

More capabilities will be added over time as the initial focus of the NSX plugin is meant to enable the NSX integration with vRealize Automation. More information on this topic can be found in the
vRealize Automation section of this document.

While several basic NSX operations are already included in the plugin, users might be required to automate tasks that are not natively present: extensibility can easily be done as vRO ships with a powerful HTTP REST API plugin that administrators can leverage to build custom integrations with NSX.

Two examples are provided here on how to create vRealize Orchestrator workflows using the HTTP REST API plugin and JavaScript to achieve the following:

- Create an IP Set in NSX
- Enable or Disable HA on an NSX Edge

These examples require basic knowledge of vRO, but are easy enough to follow even for beginners. Please refer to the vRealize Orchestrator page on the VMware Technology Network website for additional information and resources.

Example: Create an IP Set in NSX Using vRealize Orchestrator

IP Sets can be included within NSX Security Groups or used natively in the Distributed Firewall (DFW). IP Sets can be used to describe objects on the network that are not present within the vCenter environment (i.e. an external physical server).

We'll create a vRealize Orchestrator workflow that will create a new IP Set on the NSX Manager, allowing the user to specify name, description and value (a comma-separated list of IP addresses).

Before creating the workflow, we'll register our NSX Manager as an endpoint in our vRealize Orchestrator inventory: this is a common practice that will allow users to not have to specify the NSX Manager IP address and credentials every time, but just refer to the NSX Manager endpoint as an object (of type **REST:RestHost**).

To create the endpoint just run the **Add a REST host** workflow in the **Library → HTTP-REST → Configuration** folder and specify the required NSX Manager information (URL, credentials, timeout and certificate), as shown in the below figures.

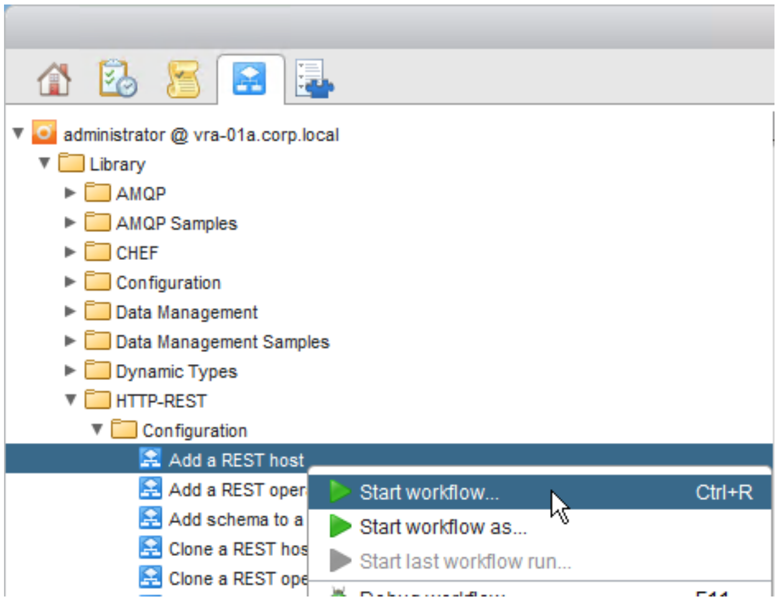
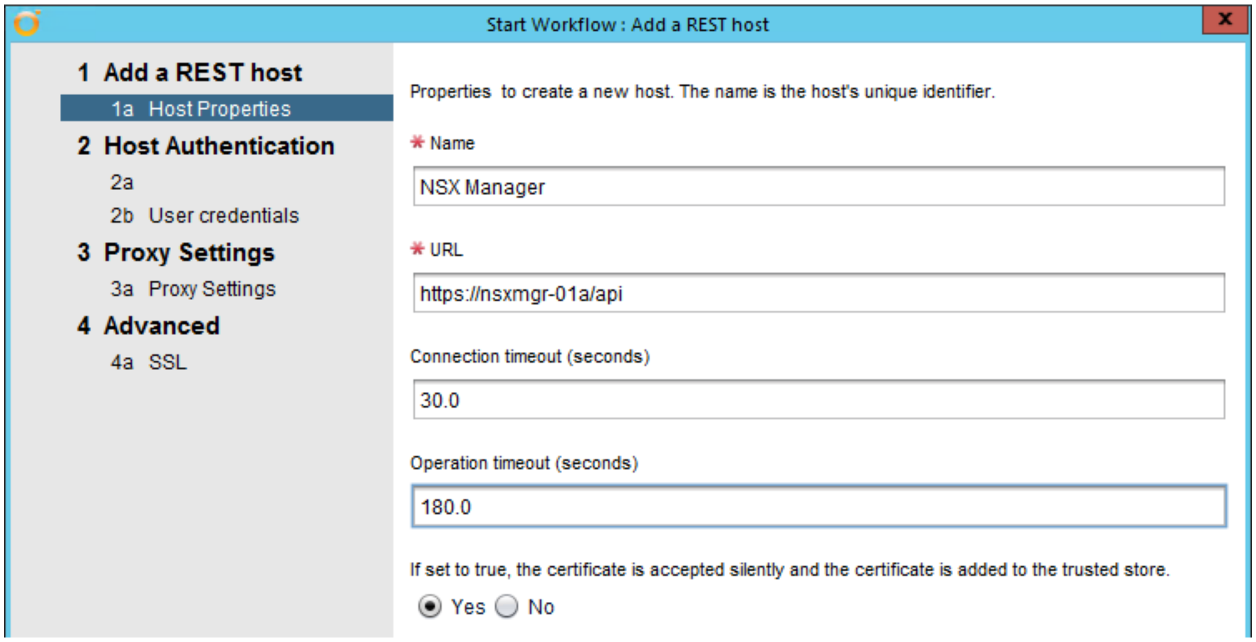


Figure 36: Run the "Add a REST host" Workflow in vRO



Properties to create a new host. The name is the host's unique identifier.

* Name

NSX Manager

* URL

https://nsxmgr-01a/api

Connection timeout (seconds)

30.0

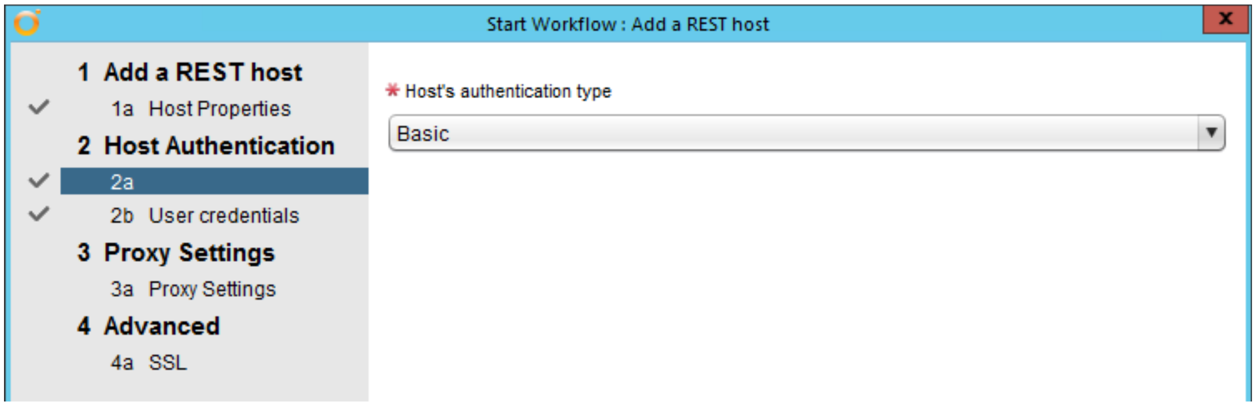
Operation timeout (seconds)

180.0

If set to true, the certificate is accepted silently and the certificate is added to the trusted store.

[✓ Yes] No

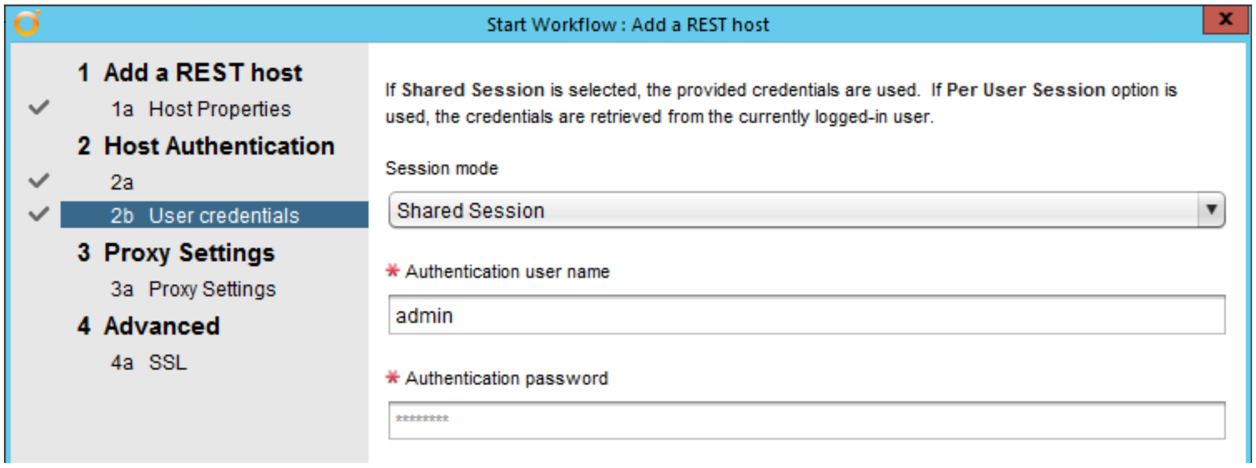
Figure 37:: Specify NSX Manager IP Address, Timeout and Certificate in vRO


Start Workflow : Add a REST host

1 Add a REST host
✓ 1a Host Properties
2 Host Authentication
✓ 2a
✓ 2b User credentials
3 Proxy Settings
3a Proxy Settings
4 Advanced
4a SSL

* Host's authentication type: Basic

Figure 38: Specify NSX Manager Basic Authentication in vRO


Start Workflow : Add a REST host

1 Add a REST host
✓ 1a Host Properties
2 Host Authentication
✓ 2a
✓ 2b User credentials
3 Proxy Settings
3a Proxy Settings
4 Advanced
4a SSL

If **Shared Session** is selected, the provided credentials are used. If **Per User Session** option is used, the credentials are retrieved from the currently logged-in user.

Session mode: Shared Session

* Authentication user name: admin

* Authentication password: ********

Figure 39: Specify NSX Manager Credentials in vRO

Please note that the **Operation timeout (seconds)** parameter is set to 180 seconds as some tasks on NSX are blocking (they do not return until the operation in the backend is completed) and the default value (60 seconds) could lead to timeout errors.

Check whether the workflow has been executed correctly and if the endpoint is shown in the HTTP-REST section of the vRO inventory:

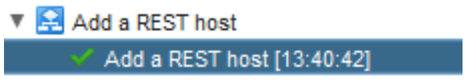


Figure 40: Verify vRO Workflow Completition

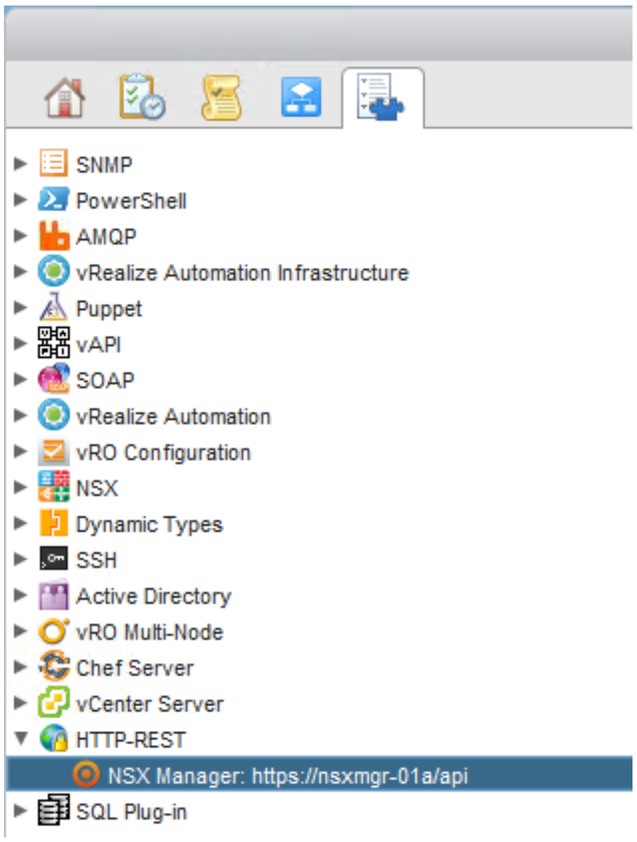


Figure 41: vRO Inventory with NSX as an HTTP-REST Endpoint

We can now prepare a vRealize Orchestrator workflow that creates a new IP Set in NSX. The workflow contains a single scriptable task, which includes JavaScript code that:

- Builds an XML document according to the NSX API specification (refer to the NSX API guide)
- Sends an HTTP POST request to the NSX Manager with the XML document as the body
- Parses the return code and, if successful, returns the NSX object ID of the created IP Set

The NSX API documentation, under section **Create an IPset**, specifies that all IP Sets must be created by executing an HTTP POST to the following URL:

```
https://nsxmgr-ip/api/2.0/services/ipset/scopeId
```

The guide also mentions that for all IP Sets the **scopeId** parameter in the URL corresponds to the global scope (**globalroot-0**).

The NSX API documentation also provides a detailed description of the XML document that must be included in the HTTP request body to create the new IP Set:

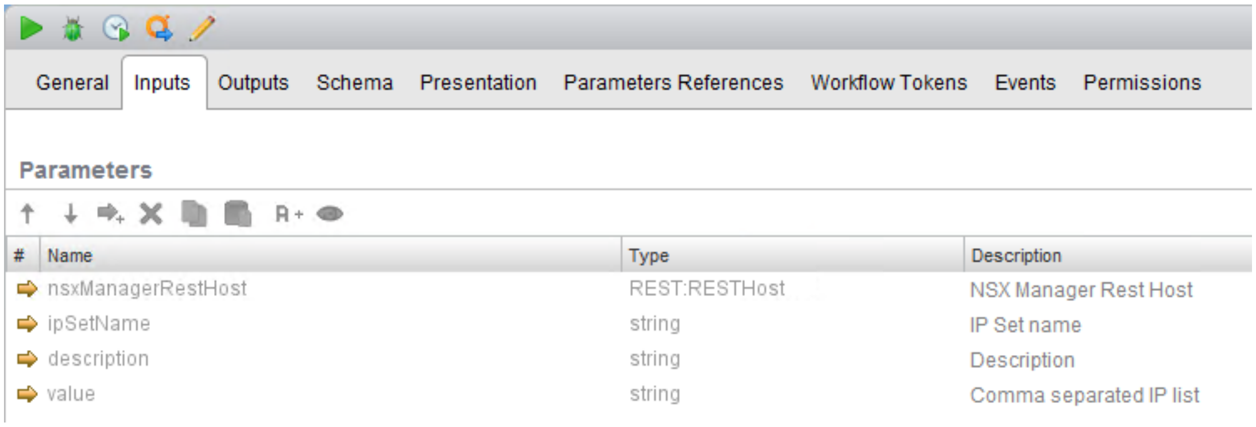
```
<ipset>
<objectId />
<type>
<typeName />
</type>
<description>
(IP Set description)
</description>
<name>(IP Set name)</name>
<revision>0</revision>
<objectTypeName />
<value>(comma separate list of IPs)</value>
</ipset>
```

Figure 42: XML Schema

Some of the parameters are empty or fixed, and, the relevant ones are highlighted in red: **description**, **name** and **value**. All these parameters are strings of text.

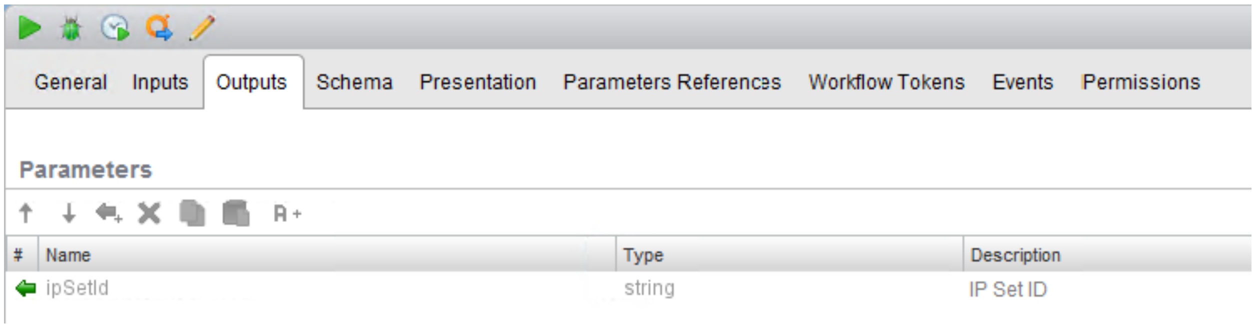
The vRealize Orchestrator workflow can be created interactively using the UI by dragging and dropping components in the canvas as well as binding input and output parameters. This document assumes that the reader has the basic knowledge to use vRO: please refer to the respective documentation for additional information.

The following figures show how the vRO workflow looks like (inputs, outputs and schema):



#	Name	Type	Description
	nsxManagerRestHost	REST:RESTHost	NSX Manager Rest Host
	ipSetName	string	IP Set name
	description	string	Description
	value	string	Comma separated IP list

Figure 43: "Create NSX IP Set" vRO Workflow Inputs



#	Name	Type	Description
	ipSetId	string	IP Set ID

Figure 44: "Create NSX IP Set" vRO Workflow Outputs

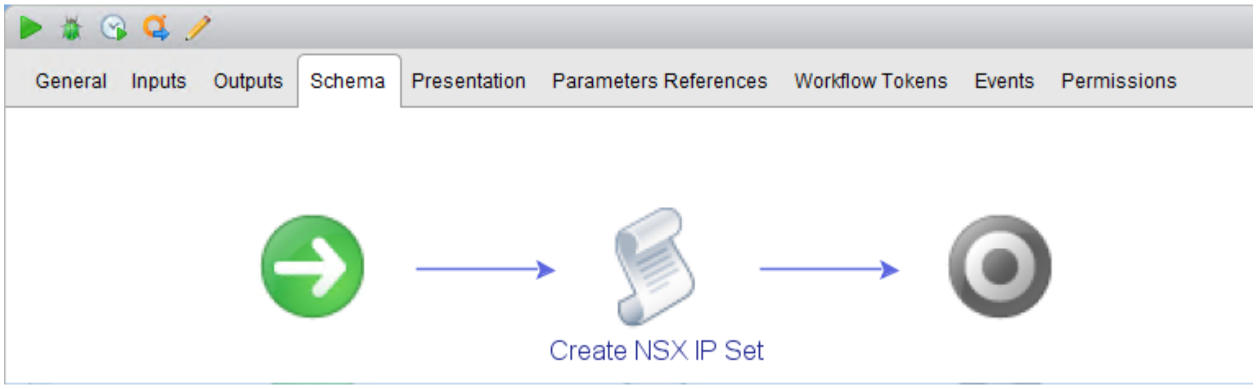
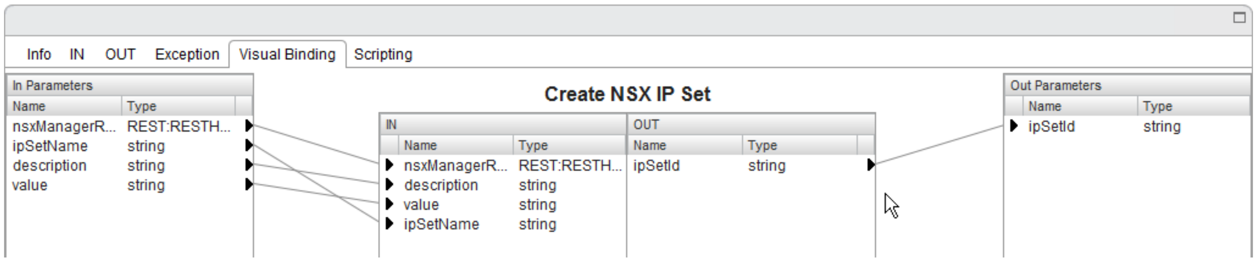


Figure 45: "Create NSX IP Set" vRO Workflow Schema

The scriptable task is bound to all the workflow inputs and outputs so that all variables are available in the JavaScript code. The visual binding of the task is shown in the below figure.



In Parameters

Name	Type
nsxManagerR...	REST:RESTH...
ipSetName	string
description	string
value	string

IN

Name	Type
nsxManagerR...	REST:RESTH...
description	string
value	string
ipSetName	string

OUT

Name	Type
ipSetId	string

Out Parameters

Name	Type
ipSetId	string

Figure 46: "Create NSX IP Set" vRO Workflow's Scriptable Task Visual Binding

The following JavaScript code is used in the **Scripting** section of the task. Please refer to the comments in the code to understand the logic/flow better.

```
// vRealize Orchestrator script: Create NSX IP Set
// This script uses the REST Plugin for vRO to create an IP Set in NSX
// Inputs are:
```

```
// - nsxManagerRestHost (type REST:RESTHost): specifies the NSX Endpoint to be used as REST API target
// - ipSetName (type string): specifies the name of the IP Set to be created
// - description (type string): specifies the description of the IP Set to be created
// - value (type string): specifies the value (comma separate IP list) of the IP Set to be created
// Outputs:
// - ipSetId (type string): NSX ID of the created IP set


// XML Schema for IP Set creation (from API guide) is:
//
//<ipset>
//<objectId />
//<type>
//<typeName />
//</type>
//<description>(IP Set description)</description>
//<name>(IP Set name)</name>
//<revision>0</revision>
//<objectTypeName />
//<value>(comma separate list of IPs)</value>
//</ipset>
//

// Prepare the HTTP body as an XML document with the required parameters
var xmlbody = new XML('<ipset />');
xmlbody.objectId = "";
xmlbody.type.typename = "";
xmlbody.description = description;
xmlbody.name = ipSetName;
xmlbody.revision = 0;
xmlbody.objectTypeName = "";
xmlbody.value = value;
// Uncomment the next line to dump the generated XML to the debug
//System.debug("Generated XML is: " + xmlbody);


// HTTP POST request is prepared using the URL: /2.0/services/ipset/scopeId
// Note that "/api" is not shown in the request URL as it's already set in the endpoint configuration
// scopeId is globalroot-0
var request = nsxManagerRestHost.createRequest("POST", "/2.0/services/ipset/globalroot-0", xmlbody.toString());
request.contentType = "application/xml";


// Execute the HTTP request
System.debug("Creating IP Set " + ipSetName);
System.debug("POST Request URL: " + request.fullUrl);
var response = request.execute();

// Evaluate the HTTP response
// We expect a HTTP 201 (Created), otherwise throw an exception
System.debug("GET Response Status Code: " + response.statusCode);
if (response.statusCode == 201) {
    System.debug("Successfully created IP Set " + ipSetName);
}
else {
    throw("Failed to create IP Set " + ipSetName);
}
```

```

// Return created IP Set id (it's the last element in a returned URI in the "Location" HTTP  header
// (it is also contained in the body of the respose)
ipSetId = response.getAllHeaders().get("Location").split('/').pop();
System.debug("Created IP Set ID is "  + ipSetId);
```

Figure 47: Javascript Code for vRO Script

The way the XML schema is created follows the ECMAScript for XML (E4X), which is an extension that adds native XML to JavaScript. This simplifies how XML information can be prepared and parsed to better interact with RESTful APIs.

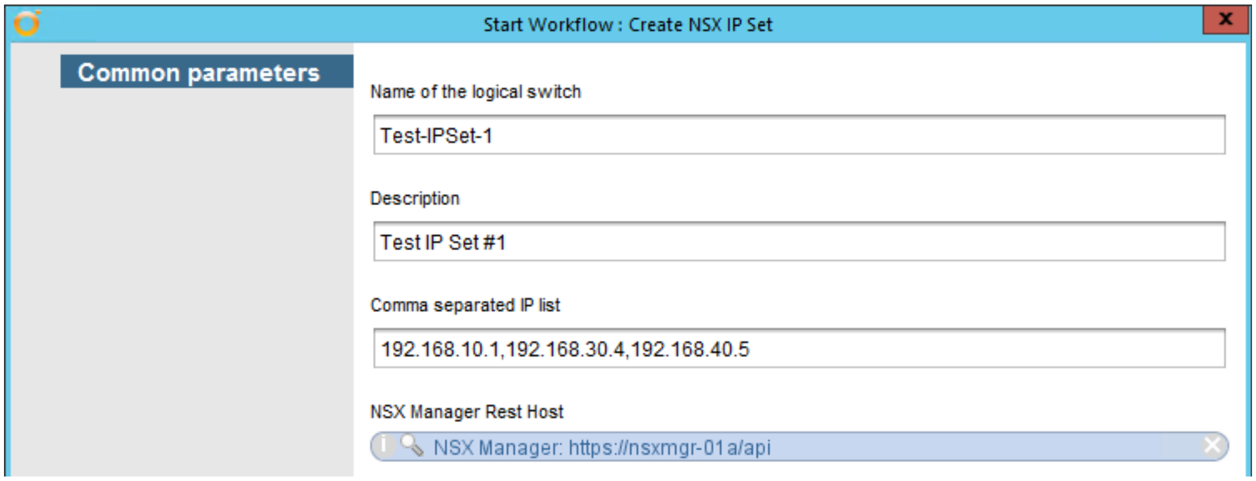
Users can also choose to build the HTTP body by simply leveraging JavaScript strings, as in the below code example (that achieves the same result as in the previous example, but without using the E4X syntax to prepare the XML document) – note that the string is now used directly in the **createRequest** function, without the **.toString()** conversion.

```
// Prepare the HTTP body as an string with the required parameters
stringbody='<ipset><objectId /><type><typeName /></type><description>' + description + '</description><name>' + ipSetName + '</name><revision>0</revision><objectTypeName /><value>' + value + '</value></ipset>';
// Uncomment the next line to dump the generated string to the debug
//System.debug("Generated string is: " + stringbody);

// HTTP POST request is prepared using the URL: /api/2.0/services/ipset/scopeId
// Note that "/api" is not shown in the request URL as it's already set in the endpoint configuration
// scopeId is globalroot-0
var request = nsxManagerRestHost.createRequest("POST", "/2.0/services/ipset/globalroot-0", stringbody);
request.contentType = "application/xml";
```

Figure 48: Alternative Usage of Javascript to Build HTTP Body

As an alternative, vRO also provides an XML library to implement the document structure if required.

Now that the workflow is created, users can run it in vRealize Orchestrator to create a new IP Set in NSX, specifying the parameter strings and the NSX Manager endpoint (selected from the available HTTP-REST object list in the inventory). A screenshot displaying this is shown below in Figure 49.



Name of the logical switch

Test-IPSet-1

Description

Test IP Set #1

Comma separated IP list

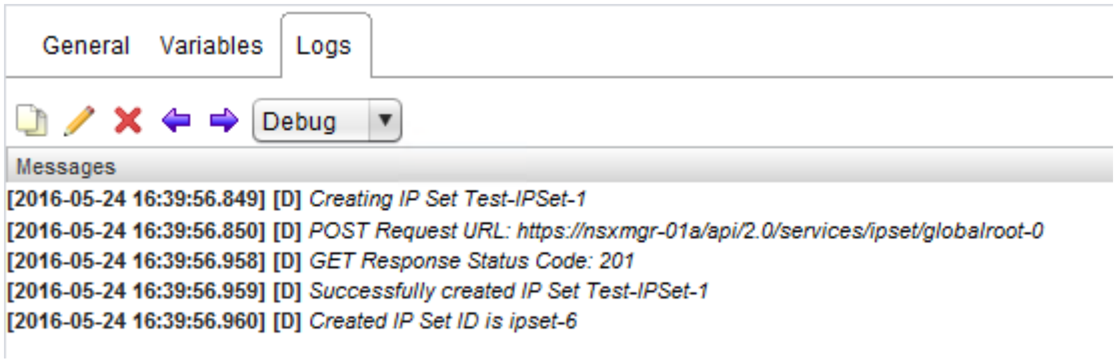
192.168.10.1,192.168.30.4,192.168.40.5

NSX Manager Rest Host

NSX Manager: https://nsxmgr-01a/api

Figure 49: Run "Create NSX IP Set" vRO Workflow

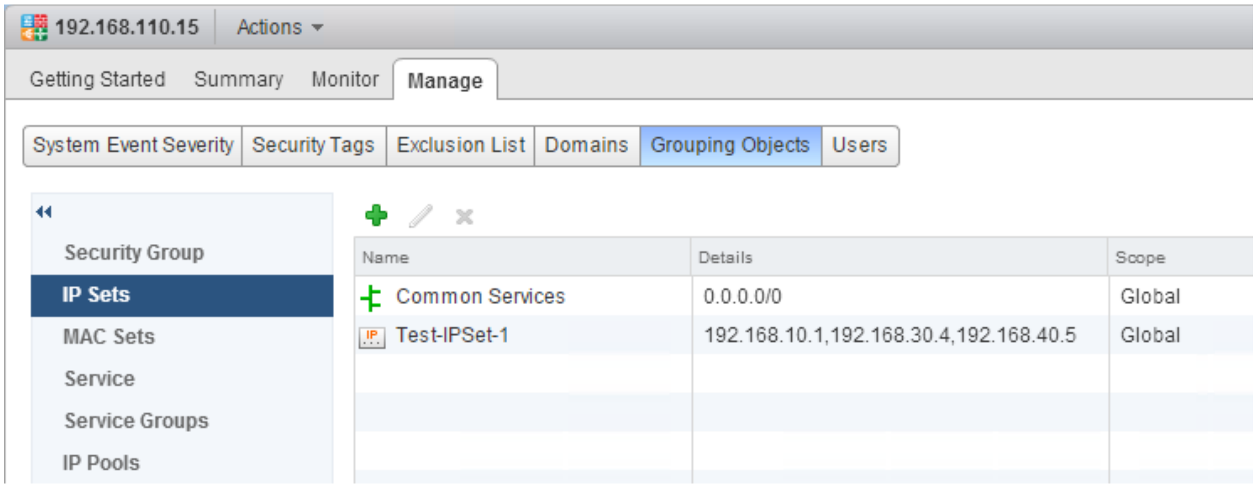
Once the workflow run is completed successfully, it’s possible to verify in the vRealize Orchestrator debug logs that the IP Set was successfully created (HTTP Post returned **201 – Created**). In the example shown in the following figure, the new object ID is ***ipset-6***; this handler can be used for subsequent CRUD operations.



```
[2016-05-24 16:39:56.849] [D] Creating IP Set Test-IPSet-1
[2016-05-24 16:39:56.850] [D] POST Request URL: https://nsxmgr-01a/api/2.0/services/ipset/globalroot-0
[2016-05-24 16:39:56.958] [D] GET Response Status Code: 201
[2016-05-24 16:39:56.959] [D] Successfully created IP Set Test-IPSet-1
[2016-05-24 16:39:56.960] [D] Created IP Set ID is ipset-6
```

Figure 50: "Create NSX IP Set" vRO Workflow Logs

Using the NSX GUI, it’s possible to verify that the new IP Set is created and it contains the expected addresses; this is validated below in Figure 51.



Name	Details	Scope
Common Services	0.0.0.0/0	Global
Test-IPSet-1	192.168.10.1,192.168.30.4,192.168.40.5	Global

Figure 51: IP Sets as They Appear in NSX GUI

This example demonstrated how to create vRealize Orchestrator workflows that leverage the NSX REST API via the HTTP-REST plugin.

Example: Enable or Disable HA on an NSX Edge Using vRO

NSX Edges can be configured to run in Active/Standby high availability (HA) mode; the feature can be enabled at deployment time or later. The vRealize Orchestrator plugin for NSX does not implement such methods, but, similar to the previous example, in this example, it will be implemented using the HTTP-REST module.

To avoid redundancy, this example omits the general initial details presented in the previous section, including the configuration of an HTTP-REST endpoint for the NSX Manager.

This workflow requires the user to specify the NSX Manager HTTP-REST endpoint, the Edge object ID as an input string, and a boolean that determines whether HA must be enabled or not. The difference from the previous example is that now the user prompts for a desired state (i.e. HA enabled or disabled), so we first need to query the current configuration to determine whether an action is needed.

The workflow contains a single scriptable task, which includes JavaScript code which performs the following:

- Queries the NSX API to retrieve the HA configuration of a specific Edge
- Parses the returned XML document to discover whether HA is currently enabled
- Determines if HA is already in the desired state (if so, nothing to do)

- If HA is not in the desired state, modifies the XML document to change configuration and sends an HTTP POST request to the NSX Manager with the updated body. Note that only a single field of the original XML document is modified, while all other HA parameters are preserved

- Checks the return code to determine if the operation was successful. The workflow does not return any output; an exception will be thrown in case of error

The NSX API documentation, under the section **Working with High Availability (HA)**, specifies that HA configuration must be managed through the following URL:

```
https://nsxmgr-ip/api/4.0/edges/edgeId/highavailability/config
```

The same URL is used for retrieving the existing configuration (via HTTP GET) or modifying it (via HTTP PUT).

Another difference from the previous example is that now an Edge object ID is required; this information is specified in the URL with the parameter **edgeId**. The workflow must complete the URL with the ID provided by the user as an input parameter.

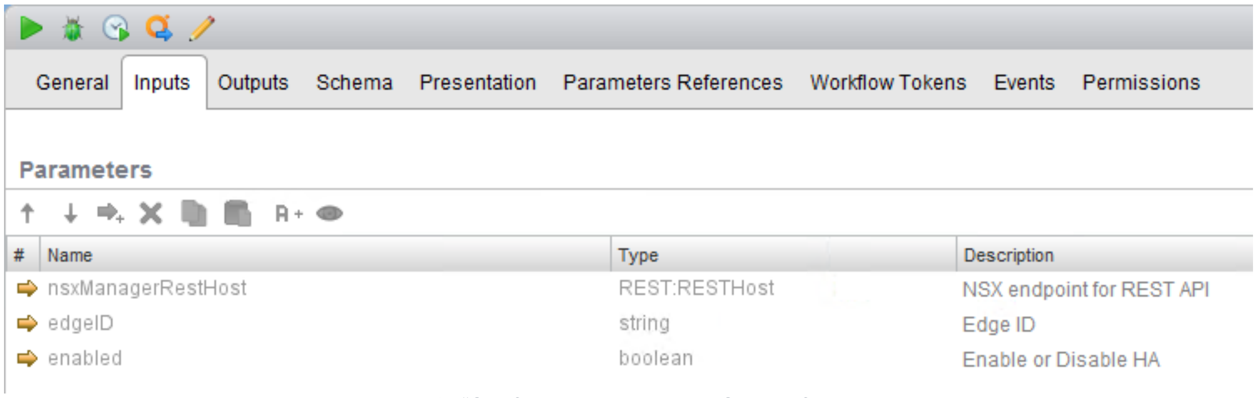
The NSX API documentation also provides a detailed description of the XML document that is returned by the HTTP GET request as shown below.

```
<highAvailability>
<vnic>(vNic index for HA)</vnic>
<ipAddresses>
<ipAddress>(First peer IP address)</ipAddress>
<ipAddress>(Second peer IP address)</ipAddress>
</ipAddresses>
<declareDeadTime>(Dead timer in seconds)</declareDeadTime>
<enabled>(enabled - true or false)<enabled>
</highAvailability>
```

Figure 52: XML Schema

For this specific scenario we just want to enable or disable HA, so we ignore all the parameters except the **<enabled> field**.

As this workflow has no outputs and the schema and scriptable task visual bindings are similar to the previous example, only the inputs are shown in the below figure.


General | Inputs | Outputs | Schema | Presentation | Parameters References | Workflow Tokens | Events | Permissions

Parameters

#	Name	Type	Description
	nsxManagerRestHost	REST:RESTHost	NSX endpoint for REST API
	edgeID	string	Edge ID
	enabled	boolean	Enable or Disable HA

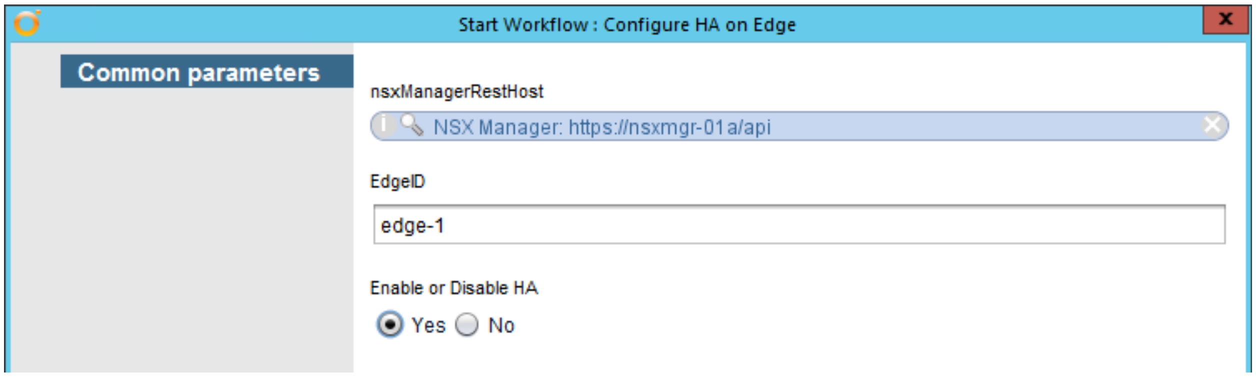
Figure 53: "Configure Edge HA" vRO Workflow Inputs

The following JavaScript code is used in the **Scripting** section of the task. Please refer to the comments in the code to understand the logic/flow better.

```
// vRealize Orchestrator script: Configure HA on NSX Edge
// This script uses the REST Plugin for vRO to enable HA on an NSX Edge
// It first retrieves the configuration of the NSX Edge and evaluates whether it's required to enable or disable HA
// If a change is required, perform the action
// Inputs are:
// - nsxManagerRestHost (type REST:RESTHost): specifies the NSX Endpoint to be used as REST API target
// - edgeID (type string): specifies the ID of the Edge (from NSX Manager)
// - enabled (type bool): specifies whether Edge HA should be enabled or disabled
// No outputs

// Retrieve the existing configuration
// HTTP GET request is prepared using the URL: /api/4.0/edges/edge-id/highavailability/config
// Note that "/api" is not shown in the request URL as it's already set in the endpoint configuration
var request = nsxManagerRestHost.createRequest("GET", "/4.0/edges/" + edgeID + "/highavailability/config");
request.contentType = "application/xml";

// Execute the HTTP request
System.debug("Querying current HA configuration for Edge " + edgeID);
System.debug("GET Request URL: " + request.fullUrl);
var response = request.execute();

// Evaluate the HTTP response
// We expect a HTTP 200 (OK), otherwise throw an exception
System.debug("GET Response Status Code: " + response.statusCode);
if (response.statusCode == 200) {
    System.debug("Response is success (200)!");
    // We can optionally output in the debug the entire XML document output (uncomment the next line)
    //System.debug("Response content is " + response.contentAsString);
}
else {
    throw("Failed to get HA configuration for Edge " + edgeID);
}

// We were able to retrieve the Edge configuration, let's now parse the returned XML
// Use the XML parser to parse the HTTP XML in the response
var document = XML(response.contentAsString);
// Uncomment the next line to dump the parsed XML to the debug
//System.debug("Edge HA configuration is " + document.toString());



// Determine if we need to do change something or if the HA state is already the desired one
// Initialize a boolean variable to false: if we need to do something will change to true later
var dosomething = false;
if(enabled) { // HA desired state is enabled
    if(document.enabled == "true") { // HA already enabled, nothing to do!
        System.debug("HA is already enabled on Edge " + edgeID);
    }
    else { // HA was not enabled, we must do something
        // Directly modify the XML document with the syntax below
        document.enabled = "true";
        dosomething = true;
    }
}
else { // HA desired state is disabled
    if(document.enabled == "true") { // HA was enabled, we must do something.
        // Directly modify the XML document with the syntax below
        document.enabled = "false";
        dosomething = true;
    }
    else { // is already disabled, nothing to do!
        System.debug("HA is already not enabled on Edge " + edgeID);
    }
}

// It's been determined that we need to do something, we need to submit the change to the NSX API
if(dosomething) {
    // Uncomment the next line to dump the modified XML to the debug
    //System.debug("Modified HA configuration is " + document.toString());
    // Conver the XML to a string that we will send in the HTTP body
    xmlbody=document.toString();
    // Uncomment the next line to dump the HTTP body we are about to send
    //System.debug("HTTP body being sent: " + xmlbody);
    // HTTP PUT request is prepared using the URL: /api/4.0/edges/edge-
id/highavailability/config with the required HTTP body
    // Note that "/api" is not shown in the request URL as it's already set in the endpoint configuration
    var request = nsxManagerRestHost.createRequest("PUT", "/4.0/edges/" + edgeID + "/highavailability/config",xmlbody);
    request.contentType = "application/xml";
    System.debug("Setting HA to " + enabled + " on Edge " + edgeID);

    // Execute the HTTP request
    System.debug("PUT Request URL: " + request.fullUrl);
    var response = request.execute();

    // Evaluate the HTTP response
    // We expect a HTTP 204 (No Content), otherwise throw an exception
    System.debug("PUT Response Status Code: " + response.statusCode);
    if (response.statusCode == 204) { // success!
        System.debug("HA successfully set to " + enabled + " on Edge " + edgeID);
    }
    else {
        throw("Failed to set HA to " + enabled + " on  Edge " + edgeID);
    }
}
```

Figure 54: Javscript code for vRO Workflow

Now that the workflow is created, users can run it in vRealize Orchestrator, specifying the Edge object ID and its desired HA state; a screenshot is shown below in Figure 55.



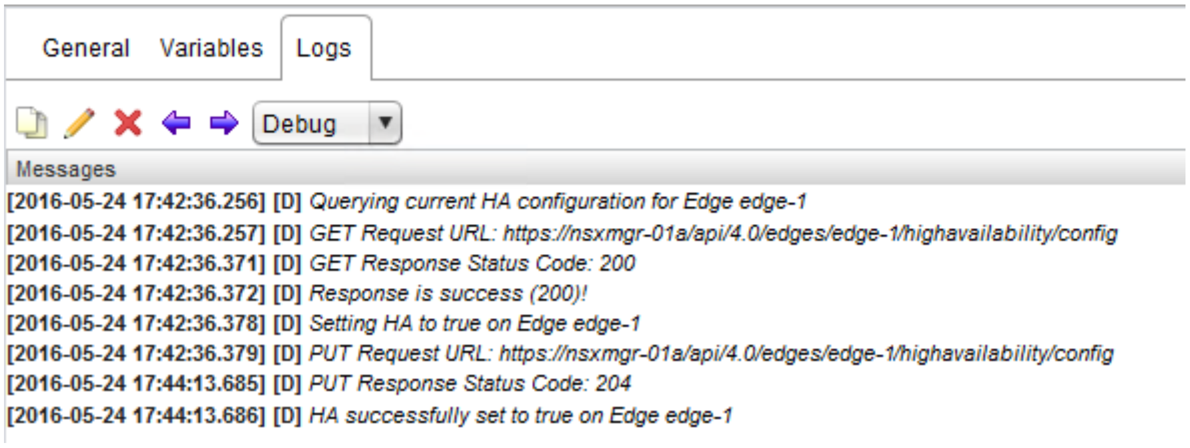
nsxManagerRestHost: NSX Manager: https://nsxmgr-01a/api

EdgeID: edge-1

Enable or Disable HA: [✓ Yes] No

Figure 55: Run "Configure Edge HA" vRO Workflow

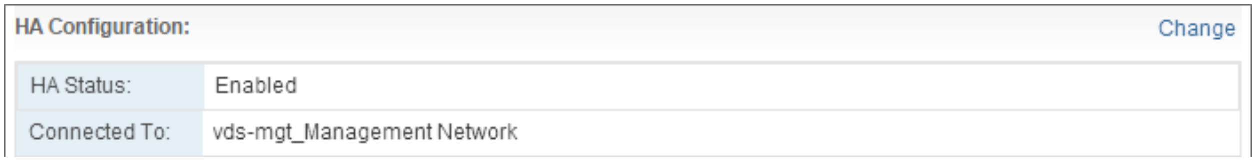
Once the workflow run is completed successful, it’s possible to verify in the vRealize Orchestrator debug logs that the configuration is changed as shown in Figure 56 below.



```
[2016-05-24 17:42:36.256] [D] Querying current HA configuration for Edge edge-1
[2016-05-24 17:42:36.257] [D] GET Request URL: https://nsxmgr-01a/api/4.0/edges/edge-1/highavailability/config
[2016-05-24 17:42:36.371] [D] GET Response Status Code: 200
[2016-05-24 17:42:36.372] [D] Response is success (200)!
[2016-05-24 17:42:36.378] [D] Setting HA to true on Edge edge-1
[2016-05-24 17:42:36.379] [D] PUT Request URL: https://nsxmgr-01a/api/4.0/edges/edge-1/highavailability/config
[2016-05-24 17:44:13.685] [D] PUT Response Status Code: 204
[2016-05-24 17:44:13.686] [D] HA successfully set to true on Edge edge-1
```

Figure 56: "Configure Edge HA" vRO Workflow Logs When an Action is Performed

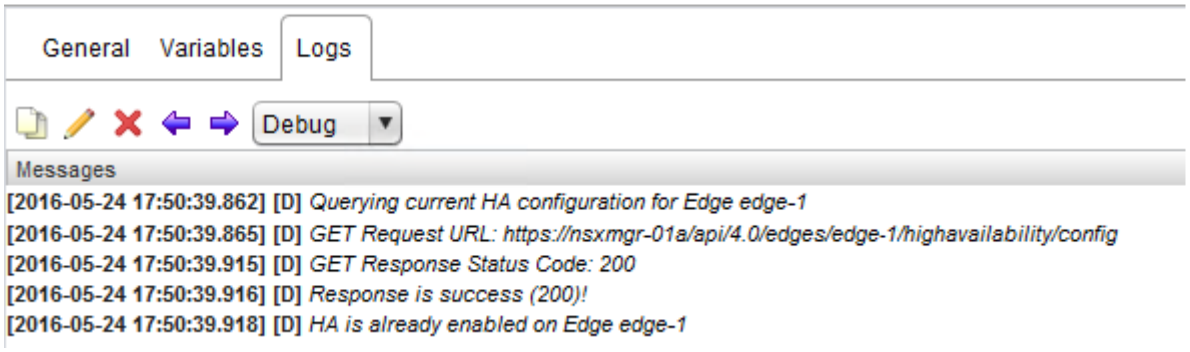
On the NSX UI, it can be verified that the HA for **edge-1** is now enabled as shown below.



HA Status:	Enabled
Connected To:	vds-mgt_Management Network

Figure 57: NSX Edge HA

As HA is already enabled, running the workflow again with the same parameters will return the below messages in the debug log and no action is performed.



```
[2016-05-24 17:50:39.862] [D] Querying current HA configuration for Edge edge-1
[2016-05-24 17:50:39.865] [D] GET Request URL: https://nsxmgr-01a/api/4.0/edges/edge-1/highavailability/config
[2016-05-24 17:50:39.915] [D] GET Response Status Code: 200
[2016-05-24 17:50:39.916] [D] Response is success (200)!
[2016-05-24 17:50:39.918] [D] HA is already enabled on Edge edge-1
```

Figure 58:: "Configure Edge HA" vRO Workflow Logs When No Action is Performed

This example again demonstrated how to create vRealize Orchestrator workflows that leverage the NSX REST API via the HTTP-REST plugin.

Ansible

Together with Puppet, Chef and Salt, Ansible is a popular configuration management tool; it can be used to describe the state of a system and provides tools to manage its lifecycle. It can also be used for provisioning new systems; this operation can really be seen as a (complex) change of state. Ansible can also provide and orchestrate these functions on a vast number of systems at the same time.

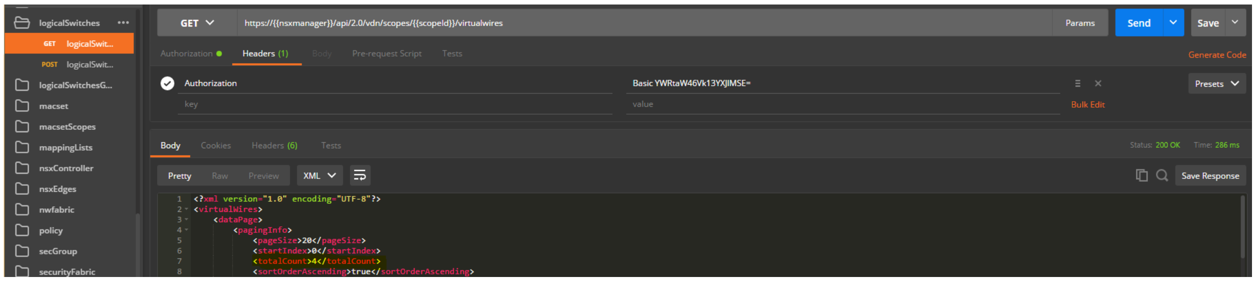
Given that state management is at the core of Ansible, idempotency is a critical characteristic: if the current state and the desired state are the same, no actions are carried out. So running twice the same Ansible playbook on the same system will result in the second run not producing any change.

To end on a light note, the name Ansible is taken from science fiction where it stands for a device that can transfer information with a speed greater than lightspeed.

Example: Using Ansible Playbook to Manage Logical Switch State

The goal of this example is to show how an Ansible playbook could be used to manage the state of a logical switch; it will be done leveraging the work done with RAML (http://raml.org/).

As always, the first thing to do is check the start state of the testing environment, specifically the number of NSX Logical Switches already available. We use Postman and leverage a collection automatically generated from the RAML file defining the NSX-v API.



GET https://{{nsxmanager}}/api/2.0/vdn/scopes/{{scopeId}}/virtualwires

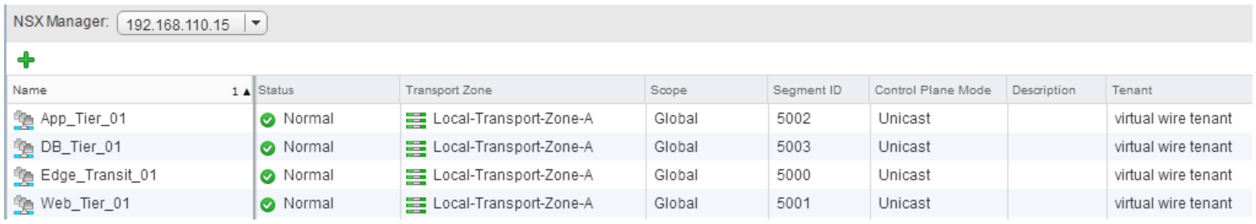
Authorization: Basic YWRtaW46Vk13YXJIMSE=

Status: 200 OK Time: 286 ms

```
<?xml version="1.0" encoding="UTF-8"?>
<virtualWires>
    <dataPage>
        <pagingInfo>
            <pageSize>20</pageSize>
            <startIndex>0</startIndex>
            <totalCount>4</totalCount>
            <sortOrderAscending>true</sortOrderAscending>
```

Figure 59: Using Postman to Verify the Number of Logical Switches Present

There are a total of four logical switches, which is confirmed by looking at the vSphere web client as shown in Figure 60 below.



NSX Manager: 192.168.110.15

Name	Status	Transport Zone	Scope	Segment ID	Control Plane Mode	Description	Tenant
App_Tier_01	✓ Normal	Local-Transport-Zone-A	Global	5002	Unicast		virtual wire tenant
DB_Tier_01	✓ Normal	Local-Transport-Zone-A	Global	5003	Unicast		virtual wire tenant
Edge_Transit_01	✓ Normal	Local-Transport-Zone-A	Global	5000	Unicast		virtual wire tenant
Web_Tier_01	✓ Normal	Local-Transport-Zone-A	Global	5001	Unicast		virtual wire tenant

Figure 60: Using NSX Manager GUI to Verify the Number of Logical Switches Present

Let's now have a look at the Ansible playbook that will be used to create a new NSX logical switch: **create_logicalswitch.yml**

```
 -emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ more create_logicalswitch.yml
---
- hosts: 127.0.0.1
  connection: local
  gather_facts: False
  vars_files:
     - answerfile.yml
  tasks:
  - name: nsx_logical_switch Operation
    nsx_logical_switch:
      nsxmanager_spec: "{{ nsxmanager_spec }}"
```

```
12.       state: present
13.       transportzone: "Local-Transport-Zone-A"
14.       name: "Ansible-LS"
15.       controlplanemode: "UNICAST_MODE"
16.       description: "Ansible Logical Switch"
17.     register: create_logical_switch
```

Figure 61: Ansible Playbook (create_logicalswitch.yml)

The first lines specify that this playbook will be run on the same machine where Ansible is installed, that facts gathering will not be performed, and that some parameters needed by the playbook are stored in the file named **answerfile.yml** (see below). Note that the path of the file specifying the RAML format of the NSX-v API is explicitly called out, together with the NSX Manager login credentials

```
 emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ more answerfile.yml
nsxmanager_spec:
  raml_file: '/home/emazza/Documents/AnsiblePlaybook/nsxraml/nsxvapi.raml'
  host: '10.152.65.157:21443'
  user: 'admin'
  password: 'VMware1!'
```

Figure 62: Parameter Specification (answerfile.yml)

Looking again at the Ansible playbook, it's important to note the following line

```
 nsx_logical_switch:
  nsxmanager_spec: "{{ nsxmanager_spec }}"
  state: present
  transportzone: "Local-Transport-Zone-A"
  name: "Ansible-LS"
```

Figure 63: "state: present" Line in Ansible Playbook

The code in Figure 63 highlights the specification of the desired state the system must be at the end of playbook run: a logical switch named **Ansible-LS** must be present in the NSX transport zone named **Local-Transport-Zone-A**. This is linked with the well-known idempotent characteristic of Ansible: it compares the current and desired state of the system and only executes changes when needed. So when the playbook will be called twice in a row, the second run will not produce any change to the system as the starting state already match the desired final state.

Below it is shown how this works at a high level. The playbook calls the function **nsx_logical_switch.py** (line 10 in Figure 61); this script is written in Python and stored under the library folder of the machine where Ansible is installed.

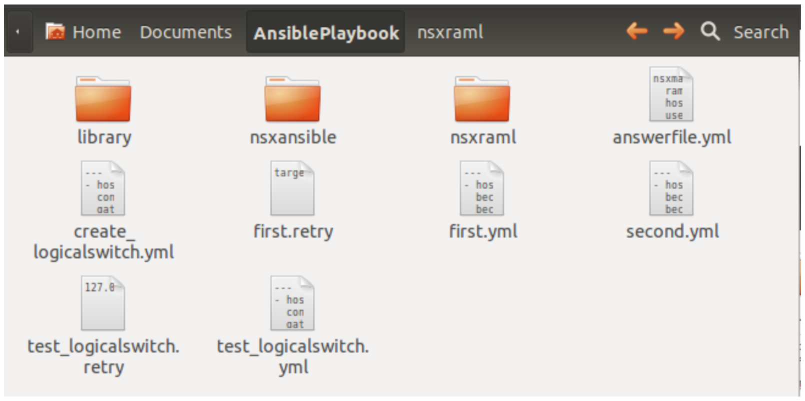


Figure 64: File Organization

Looking at the code, we can see the definition of the parameters that have been passed via the Ansible playbook (line 02 to line 11). At line 15 the **NsxClient** class is made available via importing the respective module via the **client.py** script; **NsxClient** is used to create a NSX client session object (line 15) which will be used to interact with the system programmatically.

```
1.  def main():
2.      module = AnsibleModule(
3.          argument_spec=dict(
4.              state=dict(default='present', choices=['present', 'absent']),
5.              nsxmanager_spec=dict(required=True, no_log=True),
6.              name=dict(required=True),
7.              description=dict(),
8.              transportzone=dict(required=True),
9.              controlplanemode=dict(default='UNICAST_MODE', choices=['UNICAST_MODE',
10.             'MULTICAST_MODE', 'HYBRID_MODE'])
11.         ),
12.         supports_check_mode=False
13.     )
14.
15.     from nsxramlclient.client import NsxClient
16.     client_session=NsxClient(module.params['nsxmanager_spec']['raml_file'],
17.     module.params['nsxmanager_spec']['host'], module.params['nsxmanager_spec']['user'],
18.     module.params['nsxmanager_spec']['password'])
```

Figure 65: nsx_logical_switch.py (snippet)

Looking now at the rest of the code, the further below steps can clearly be identified

```
from nsxramlclient.client import NsxClient
client_session=NsxClient(module.params['nsxmanager_spec']['raml_file'], module.params['
nsxmanager_spec']['host'], module.params['nsxmanager_spec']['user'],
module.params['nsxmanager_spec']['password'])

vdn_scope=retrieve_scope(client_session, module.params['transportzone'])
lswitch_id=get_lswitch_id(client_session, module.params['name'], vdn_scope)

if len(lswitch_id) is 0 and 'present' in module.params['state']:
```

```
9.      ls_ops_response=create_lswitch(client_session, module.params['name'],
10.     module.params['description'],  module.params['controlplanemode'], vdn_scope)
11.
12.     module.exit_json(changed=True, argument_spec=module.params,
13.     ls_ops_responsels_ops_response=ls_ops_response)
```

Figure 66: nsx_logical_switch.py (snippet)

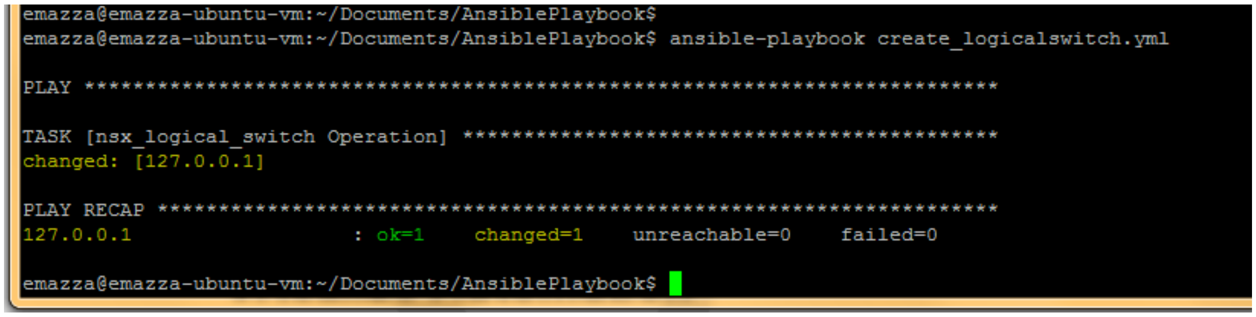
- Line 05: retrieval of the vdn scope corresponding to the transport zone passed as input
- Line 06: retrieval of the id of the logical switch belonging to the transport zone with name equal to the parameter **name**: defined in the playbook (**Ansible-LS** in this example)
- Line 08 to Line 13: if there is no such logical switch and the requested state is **present**, then the **create_lswitch** function will be called

Below the **create_lswitch** function that is defined in the same **nsx_logical_switch.py** file is looked at in more detail. This function basically creates a python dictionary representing a logical switch, fills it with the appropriate data, and passes it to the create method of the NSX client session object previously defined.

```
def create_lswitch(session, lswitchname, lswitchdesc, lswitchcpmode, scope):
    lswitch_create_dict = session.extract_resource_body_schema('logicalSwitches',
    'create')
    lswitch_create_dict['virtualWireCreateSpec']['controlPlaneMode'] = lswitchcpmode
    lswitch_create_dict['virtualWireCreateSpec']['name'] = lswitchname
    lswitch_create_dict['virtualWireCreateSpec']['description'] = lswitchdesc
    lswitch_create_dict['virtualWireCreateSpec']['tenantId'] = 'Unused'
    return  session.create('logicalSwitches', uri_parameters={'scopeId': scope},
    request_body_dict=lswitch_create_dict)
```

Figure 67: "create_lswitch" function

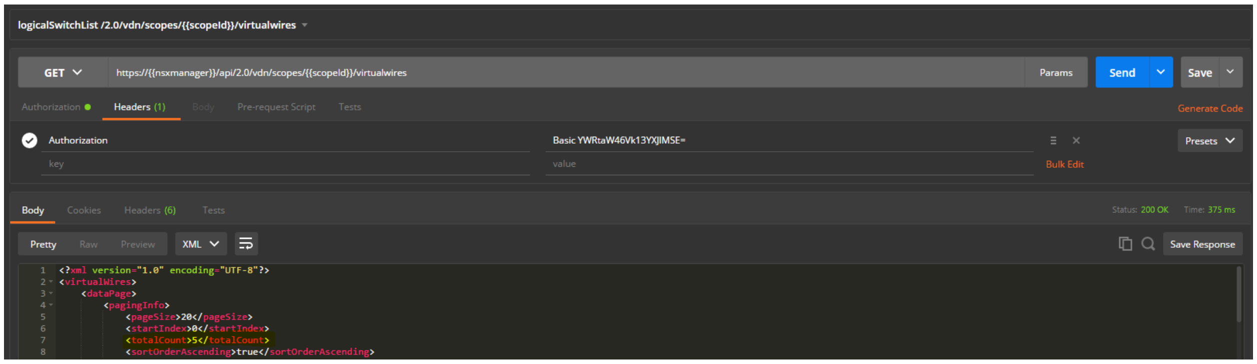
The below screenshot shows the respective Ansible playbook run output.



```
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ ansible-playbook create_logicalswitch.yml

PLAY ***************************************************************************

TASK [nsx_logical_switch Operation] ********************************************
changed: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=1    changed=1    unreachable=0    failed=0

emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
```

Figure 68: Ansible Playbook Run Output

The output is showing that the operation completed correctly and that one change to the state of the system has been made (remember that this logical switch did not exist before). Another look at Postman output will confirm this: there are now five logical switches, one of which has been recently created by the Ansible playbook



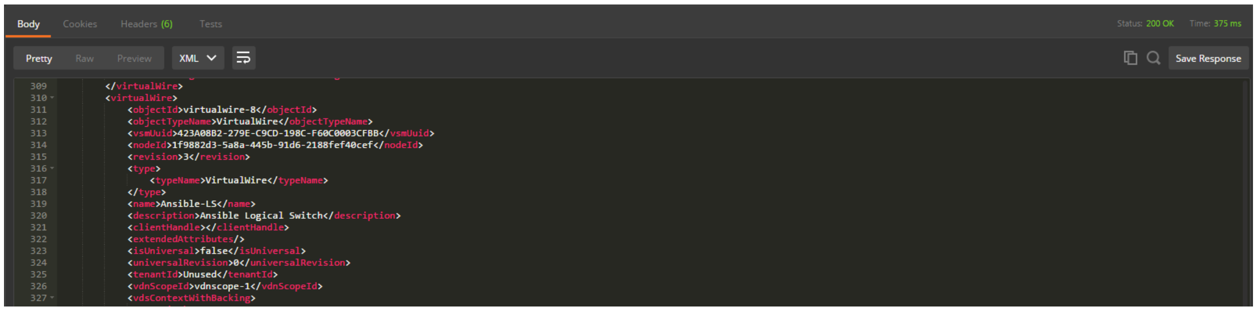
logicalSwitchList /2.0/vdn/scopes/{{scopeId}}/virtualwires

GET https://{{nsxmanager}}/api/2.0/vdn/scopes/{{scopeId}}/virtualwires

Authorization: Basic YWRtaW46Vk13YXJIMSE=

Status: 200 OK Time: 375 ms

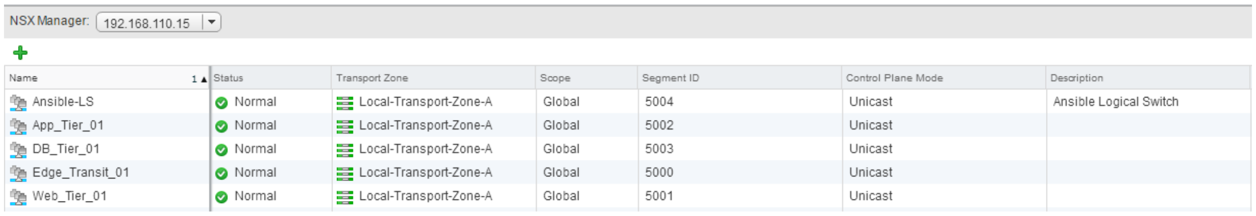
```
<?xml version="1.0" encoding="UTF-8"?>
<virtualWires>
    <dataPage>
        <pagingInfo>
            <pageSize>20</pageSize>
            <startIndex>0</startIndex>
            <totalCount>5</totalCount>
            <sortOrderAscending>true</sortOrderAscending>
```



Status: 200 OK Time: 375 ms

```
        </virtualWire>
        <virtualWire>
            <objectId>virtualwire-8</objectId>
            <objectTypeName>VirtualWire</objectTypeName>
            <vsmUuid>423A08B2-279E-C9CD-198C-F60C0003CFBB</vsmUuid>
            <nodeId>1f9882d3-5a8a-445b-91d6-2188fef40cef</nodeId>
            <revision>3</revision>
            <type>
                <typeName>VirtualWire</typeName>
            </type>
            <name>Ansible-LS</name>
            <description>Ansible Logical Switch</description>
            <clientHandle></clientHandle>
            <extendedAttributes/>
            <isUniversal>false</isUniversal>
            <universalRevision>0</universalRevision>
            <tenantId>Unused</tenantId>
            <vdnScopeId>vdnscope-1</vdnScopeId>
            <vdsContextWithBacking>
```

Figure 69: Verification via Postman NSX REST API Call Output

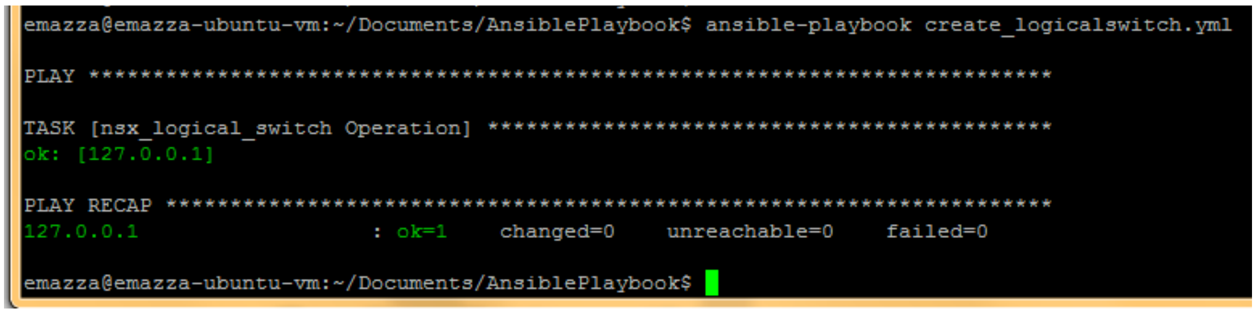


NSX Manager: 192.168.110.15

Name	Status	Transport Zone	Scope	Segment ID	Control Plane Mode	Description
Ansible-LS	Normal	Local-Transport-Zone-A	Global	5004	Unicast	Ansible Logical Switch
App_Tier_01	Normal	Local-Transport-Zone-A	Global	5002	Unicast	
DB_Tier_01	Normal	Local-Transport-Zone-A	Global	5003	Unicast	
Edge_Transit_01	Normal	Local-Transport-Zone-A	Global	5000	Unicast	
Web_Tier_01	Normal	Local-Transport-Zone-A	Global	5001	Unicast	

Figure 70: Verification of Ansible Playbook's Run Results

Running the same playbook again will not cause any change in the system as the current state is already the final desired state. Figure 71 below shows this in practice: the playbook runs correctly (**ok=1**) but no changes have been made (**changed=0**).



```
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ ansible-playbook create_logicalswitch.yml

PLAY ***************************************************************************

TASK [nsx_logical_switch Operation] ********************************************
ok: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=1    changed=0    unreachable=0    failed=0

emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
```

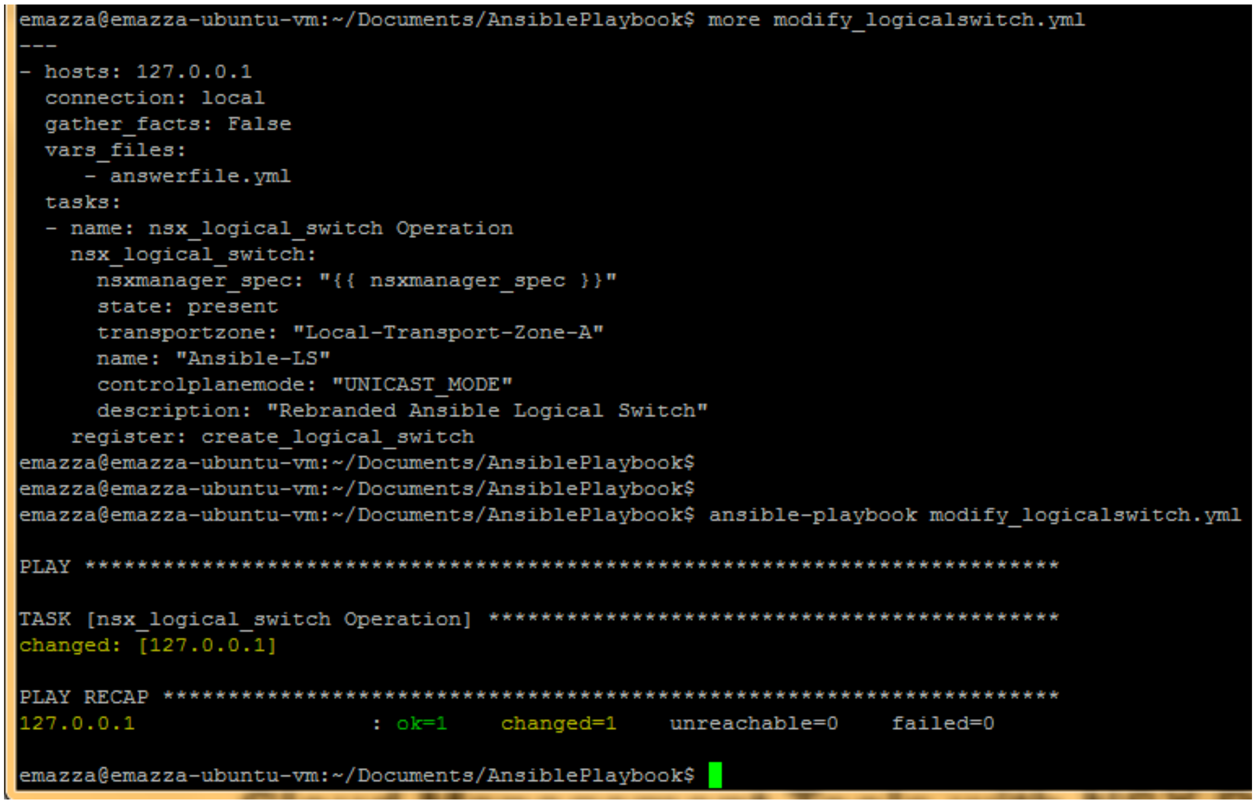
Figure 71: Ansible Playbook Run Output

We can look at how this is managed at the code level in the **nsx_logical_switch.py** file. At line 07 the Ansible playbook is required to create a logical switch with the name of a logical switch that already exists. From line 08 to line 25 we cycle through the other attributes of this logical switch to understand if their values match the ones set in the playbook and mark accordingly a change request flag. Finally, from line 26 to line 32 we deal with the appropriate course of action based on the final value of the change request flag.

```
1.   if len(lswitch_id) is 0 and 'present' in module.params['state']:
2.      ls_ops_response=create_lswitch(client_session, module.params['name'],
3.      module.params['description'], module.params['controlplanemode'], vdn_scope)
4.
5.      module.exit_json(changed=True, argument_spec=module.params,
6.      ls_ops_responsels_ops_response=ls_ops_response)
7.  elif len(lswitch_id) is not 0 and 'present' in module.params['state']:
8.      lswitch_details=get_lswitch_details(client_session,lswitch_id[0])
9.      change_required=False
10.     for lswitch_detail_key, lswitch_detail_value in
11.     lswitch_details['virtualWire'].iteritems():
12.         if lswitch_detail_key == 'name'
13.         and lswitch_detail_value != module.params['name']:
14.             lswitch_details['virtualWire']['name']=
15.             module.params['nsxmanager_spec']['name']
16.             change_required=True
17.         elif lswitch_detail_key == 'description'
18.         and lswitch_detail_value != module.params['description']:
19.             lswitch_details['virtualWire']['description']=module.params['description']

20.             change_required=True
21.         elif lswitch_detail_key == 'controlPlaneMode'
22.         and lswitch_detail_value != module.params['controlplanemode']:
23.             lswitch_details['virtualWire']['controlPlaneMode']=
24.             module.params['controlplanemode']
25.             change_required=True
26.     if change_required:
27.         ls_ops_response=change_lswitch_details(client_session,lswitch_id[0],
28.         lswitch_details)
29.         module.exit_json(changed=True, argument_spec=module.params,
30.         ls_ops_responsels_ops_response=ls_ops_response)
31.     else:
32.         module.exit_json(changed=False, argument_spec=module.params)
```

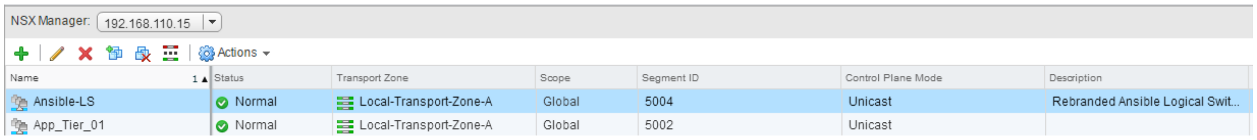
Figure 72: Idempotency in nsx_logical_switch.py

Let's see this in action and request a final state which will result in a change of the description of the already existing logical switch.



```
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ more modify_logicalswitch.yml
---
- hosts: 127.0.0.1
  connection: local
  gather_facts: False
  vars_files:
     - answerfile.yml
  tasks:
  - name: nsx_logical_switch Operation
    nsx_logical_switch:
      nsxmanager_spec: "{{ nsxmanager_spec }}"
      state: present
      transportzone: "Local-Transport-Zone-A"
      name: "Ansible-LS"
      controlplanemode: "UNICAST_MODE"
      description: "Rebranded Ansible Logical Switch"
    register: create_logical_switch
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ ansible-playbook modify_logicalswitch.yml

PLAY ***************************************************************************

TASK [nsx_logical_switch Operation] ********************************************
changed: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=1    changed=1    unreachable=0    failed=0

emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
```

Figure 73: Ansible Playbook Run Output



NSX Manager: 192.168.110.15

Actions

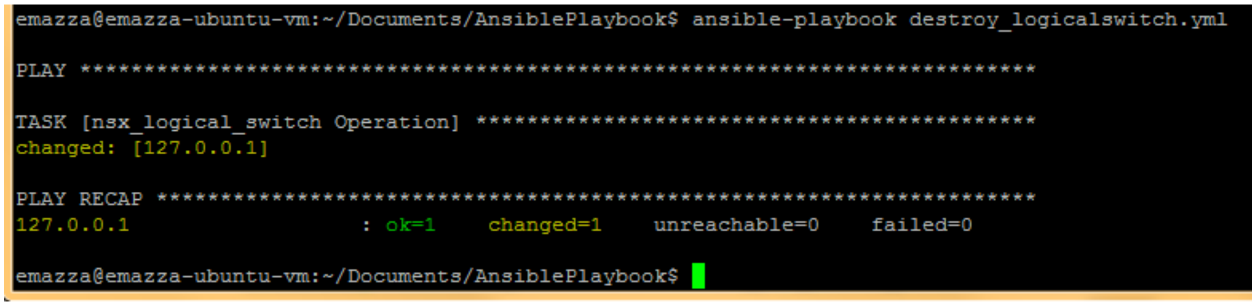
Name	Status	Transport Zone	Scope	Segment ID	Control Plane Mode	Description
Ansible-LS	Normal	Local-Transport-Zone-A	Global	5004	Unicast	Rebranded Ansible Logical Swit...
App_Tier_01	Normal	Local-Transport-Zone-A	Global	5002	Unicast	

Figure 74: Checking name change using NSX Manager GUI

Finally, let's request a final state such that this logical switch must not exist anymore in the system.

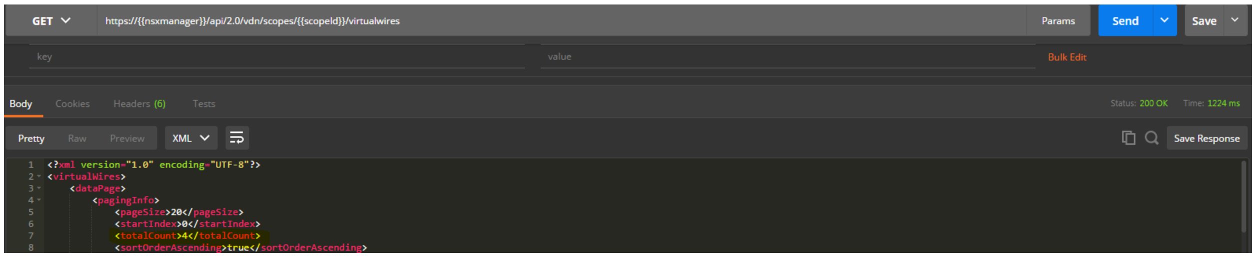
```
 emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ more destroy_logicalswitch.yml
---
- hosts: 127.0.0.1
  connection: local
  gather_facts: False
  vars_files:
     - answerfile.yml
  tasks:
  - name: nsx_logical_switch Operation
    nsx_logical_switch:
      nsxmanager_spec: "{{ nsxmanager_spec }}"
      state: absent
      transportzone: "Local-Transport-Zone-A"
      name: "Ansible-LS"
      controlplanemode: "UNICAST_MODE"
      description: "Rebranded Ansible Logical Switch"
    register: create_logical_switch
```

Figure 75: Ansible Playbook (destroy_logicalswitch.yml)



```
emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$ ansible-playbook destroy_logicalswitch.yml

PLAY ***************************************************************************

TASK [nsx_logical_switch Operation] *******************************************
changed: [127.0.0.1]

PLAY RECAP *********************************************************************
127.0.0.1                  : ok=1    changed=1    unreachable=0    failed=0

emazza@emazza-ubuntu-vm:~/Documents/AnsiblePlaybook$
```

Figure 76: Ansible Playbook Run Output

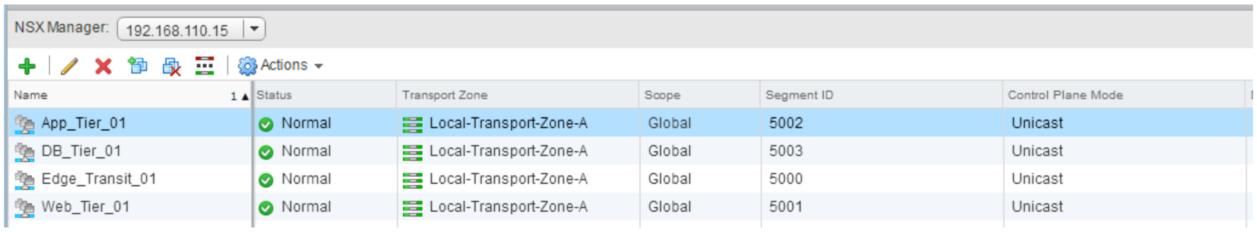


GET https://{{nsxmanager}}/api/2.0/vdn/scopes/{{scopeId}}/virtualwires

Status: 200 OK Time: 1224 ms

```
<?xml version="1.0" encoding="UTF-8"?>
<virtualWires>
    <dataPage>
        <pagingInfo>
            <pageSize>20</pageSize>
            <startIndex>0</startIndex>
            <totalCount>4</totalCount>
            <sortOrderAscending>true</sortOrderAscending>
```

Figure 77: Using Postman to Verify the Number of Logical Switches Present



NSX Manager: 192.168.110.15

Name	Status	Transport Zone	Scope	Segment ID	Control Plane Mode
App_Tier_01	Normal	Local-Transport-Zone-A	Global	5002	Unicast
DB_Tier_01	Normal	Local-Transport-Zone-A	Global	5003	Unicast
Edge_Transit_01	Normal	Local-Transport-Zone-A	Global	5000	Unicast
Web_Tier_01	Normal	Local-Transport-Zone-A	Global	5001	Unicast

Figure 78: Using NSX Manager GUI to Verify the Number of Logical Switches Present

Cloud Management Platforms with NSX REST API

Many organizations are building private cloud solutions to provide their users with tools that automatically deploy infrastructure services. Cloud architects have quickly realized that automating the provisioning and destruction of Virtual Machines (VMs) is not sufficient to satisfy business requirements: too often configuration changes in network and security are also required to provide the desired service, and, with manual provisioning of such components, the benefits of the entire cloud initiative are limited.

For example, when manually deploying production environments, the process often follows a chain of events that involve operations by several teams (VI, network, security, storage, load balancing, applications, etc.). This model not only slows down the deployment, but is also prone to human errors that can lead longer wait times or misconfigurations in security policies thus resulting in insecure environments. A common example is the removal of firewall policies related to decommissioned applications, which is a rarely performed task.

Moreover, as organizations are transitioning towards Micro-segmentation, the requirement of granular security policies deployed alongside the applications is becoming important, which again is not easy to perform via manual operations. Lastly, many companies are developing applications in-house and their dev and test environments, while not requiring the same levels of availability and security of production, are critical to deliver new features and are therefore considered revenue generating. In such environments, network and security configurations (such as NAT, Load Balancer and Firewall) are often required and must be automated.

For the above reasons, modern IaaS clouds provide automated and integrated consumption of virtual machines, network constructs (such as Logical Switches and Load Balancers), security policies, middleware and applications. Depending on the scope of the initiative (production, dev, test/QA, etc.) and the organizational structure, private cloud solutions can offer different levels of governance and consumption models (i.e. whether giving users access to service catalogs or directly to CMP APIs, implementing approval workflows and strict resource management, etc.).

While describing the differences among several Cloud Management Platform (CMP) solutions is out of the scope of this document, it is important to mention that using NSX as the underlying network and security platform is extremely beneficial as all possible governance and consumption models can apply to it. With NSX it’s possible to automate all the configuration actions by leveraging the NSX REST API: several Cloud Management Platforms (CMPs) already provide out-of-the-box integrations to consume the services provided by NSX, such as Logical Switching, Logical Routing, Distributed Firewall and Logical Load Balancing.

This section describes how one of the most widely used CMP solutions, VMware vRealize Automation, leverages the NSX API to automate the consumption of services.

Other solutions can already integrate with NSX out-of-the-box and it’s important to understand that, since most CMPs are modular, it’s possible to extend their capabilities to consume NSX services either via the REST API or through vRealize Orchestrator. The other examples provided in this document should be useful to understand how to implement such extensions using common programming languages.

vRealize Automation

VMware vRealize Automation (vRA) is a cloud automation solution that automates the delivery of custom infrastructure, application, and IT services across a multi-vendor hybrid cloud infrastructure. Details on vRA are available on VMware's vRA web site, but, in short, it provides an automated way of deploying and managing applications through a service catalog and natively integrates with NSX.

The latest version of vRA at the time of writing is 7.0. It allows native consumption of both pre-created and on-demand NSX network and security services through the concept of a blueprint: cloud architects can design entire application stacks (that include VMs, pre-existing or on-demand networks, load balancers, security groups, policies and tags) and publish them in a service catalog. Cloud consumers can then deploy, manage and destroy applications by leveraging the service catalog, as shown in the picture below.

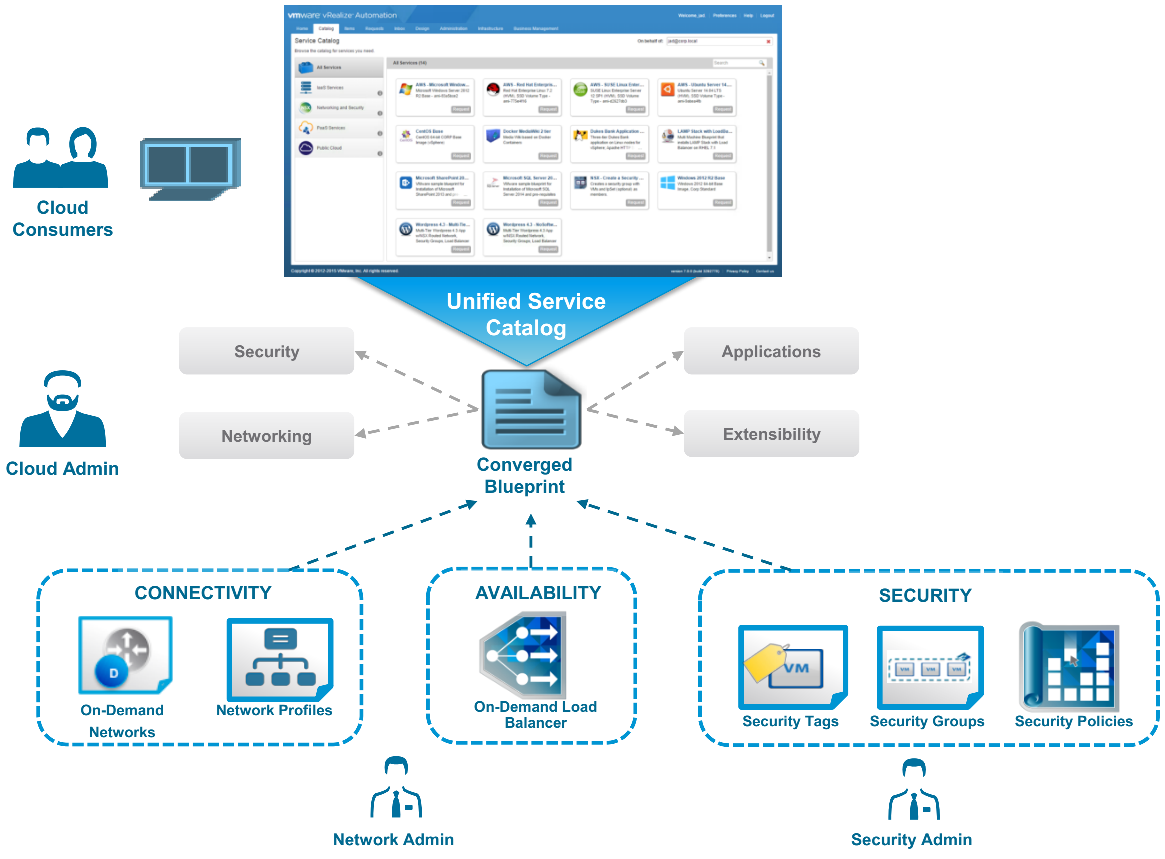


Figure 79: Cloud Consumption Model with vRA

This document is not meant to be a comprehensive guide on how vRA can consume NSX services, but the most significant aspects of the consumption layer from the network and security perspective are presented below.

In regards to Networking, vRA can natively consume the following services based on NSX:

- External networks: vRA can deploy VMs on to existing NSX Logical Switches. This is very useful for production applications that get deployed on pre-existing, shared networks.

- NAT networks: when overlapping addressing is required (very often for dev and test environments), vRA can create new NSX Logical Switches and NSX Edges and define NAT rules to allow reusability of address space across different deployments. It’s possible to define 1:1 and 1:Many network profiles, depending on the required external access. Once the application is decommissioned, vRA removes the created network objects alongside VMs.

- Routed networks: some organizations require cloud users to create on-demand routed networks. As part of a blueprint, cloud architects can configure vRA to create new NSX Logical Switches and attach them to an existing NSX Distributed Logical Router during the provisioning process. vRA also manages the IP addressing accordingly to avoid overlapping. As in the NAT case, once the application is decommissioned, dynamically created NSX Logical Switches are also removed.

- Load balancing: vRA can create an NSX Edge at application deployment time and configure it as a load balancer for the required services. Both in-line and one-arm topologies are supported, depending on the network profile.

- DHCP: if required, vRA can configure DHCP services on an NSX Edge when it configures 1:Many NAT networks.

vRA can also natively consume NSX Service Composer capabilities, allowing security admins to define a security posture that is consumed by the cloud platform. vRA leverages the following NSX security capabilities:

- Pre-created NSX Security Groups: vRA can discover existing Security Groups that exist in NSX and add VMs to them during the provisioning of the application. Once the application owner deletes the VMs, they disappear from the Security Groups, reducing the risk of stale firewall rules.

- On-demand NSX Security Groups: vRA is also capable of creating new NSX Security Groups for each VM tier (i.e. web, app or DB), and attaching existing Security Policies to them. When users decommission the applications, the on-demand Security Groups are also removed by vRA.

- Pre-created NSX Security Tags: vRA can discover and attach existing NSX Security Tags to the VMs it creates, allowing a tag-based consumption of security.

- App Isolation: to simplify security automation, vRA also provides an option (App Isolation) that can be enabled at the blueprint level. When selected, vRA will automatically generate NSX Security Groups and Security Policies to block all communications across different application deployments, unless explicitly permitted by other rules. This is extremely useful to provide Micro-segmentation without the need of complex application scoping.

The below picture shows a graphical representation of two applications deployed by vRA that include security, load balancing and App Isolation with NSX.

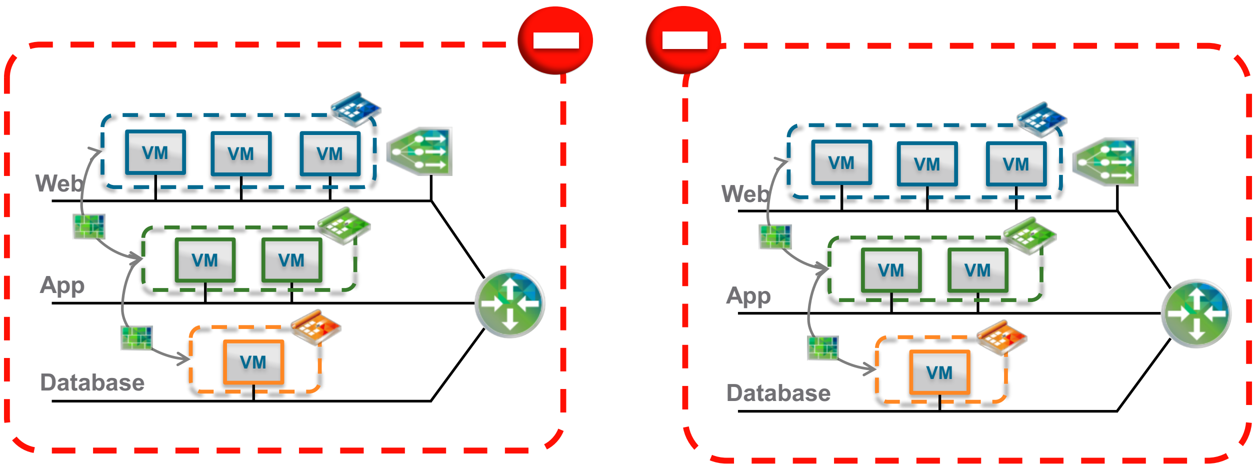


Figure 80: vRealize Automation Topology Example with App Isolation

All the NSX capabilities that vRA can consume are configured using the Converged Blueprint Designer (CBD): it provides a canvas in the UI where users can drag and drop different components (VMs, networks, security groups, load balancers, etc.) and build the required application topology. Blueprints can also be exported in a YAML file that users can be modify and re-upload (blueprint-as-code).

The below picture shows an example of the Converged Blueprint designer, where a 2-tier application (Web and DB) that includes two on-demand routed networks, a load balancer and an on-demand security group for the web tier are being utilized.

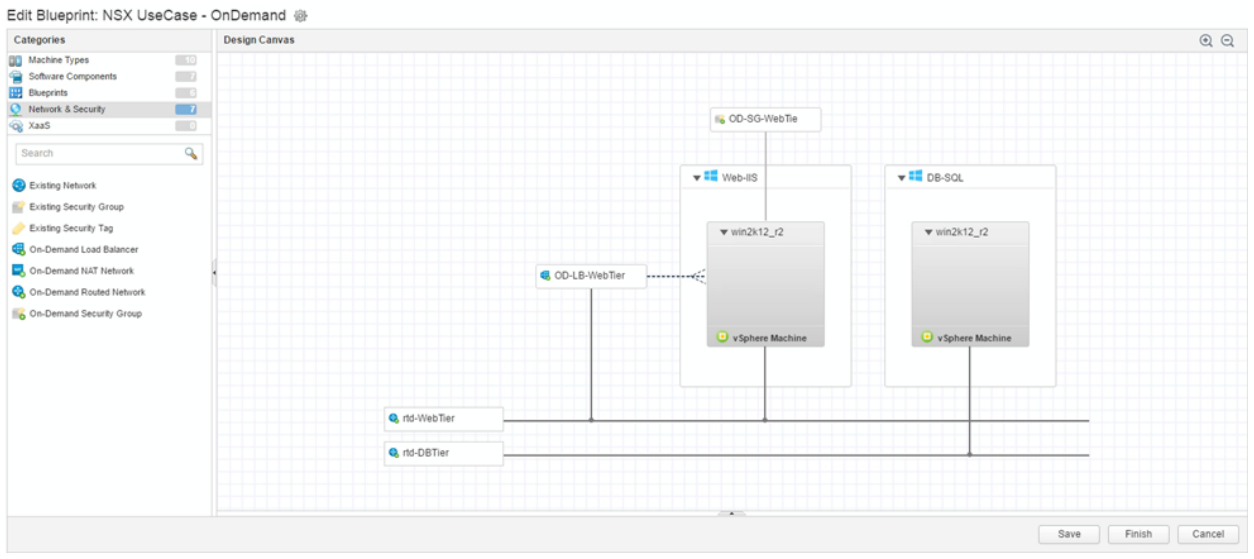


Figure 81: vRealize Automation Converged Blueprint Designer

As mentioned, vRealize Automation can leverage the NSX REST API to achieve the required configurations. To add some detail about the implementation, it is relevant to mention that the actual consumption model is implemented through vRealize Orchestrator (vRO), using the NSX vRO Plugin, as shown in the Figure 82 below.

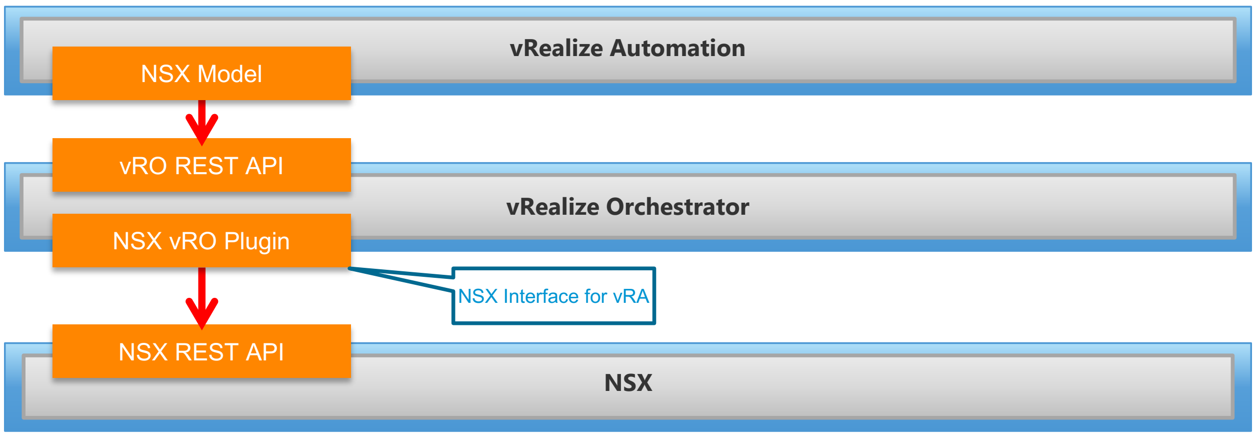


Figure 82: vRealize Automation Consumption of NSX Through vRealize Orchestrator

Among the several design reasons that drive this choice, the most relevant is the ability of extending the native consumption model. NSX is a platform that provides a comprehensive set of capabilities and not all of them are automated by the native vRealize Automation integration, as the out-of-the-box focus is set on the most common use cases. The ability to extend the default behavior and enable additional use cases is a great capability that can be leveraged through the different vRA extensibility options, that all leverage vRO as the underlying orchestration layer.

A vRO plugin for vRA allows for purpose-built custom automation of NSX objects.

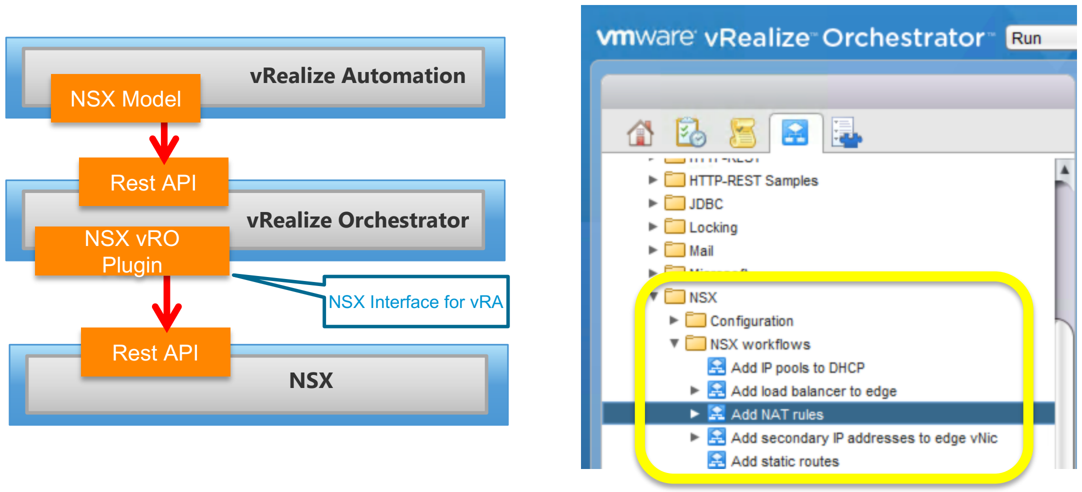


Figure 83: NSX and vRA – Extensibility using vRO

While more features are being added in the native integration, some examples on use cases that require extensibility and are simple to enable are the following:

- Change the size or the Syslog targets of an NSX Edge deployed by vRA
- Enable a persistence algorithm on an NSX Load Balancer configured by vRA
- Define Day 2 operations to add additional NAT rules for an application

These and many other use cases can be enabled by leveraging two vRA extensibility features:

- EBS (Event Broker Service): allows cloud administrators to subscribe vRO workflows to specific events, effectively modifying the behavior of the native integration. For example, one could subscribe a workflow that will change the size of the NSX Edge to the event that is generated upon deployment: this way, as soon as the end user triggers the deployment of an application, vRA will create an NSX Edge and call the workflow to change its size.

- XaaS (anything as-a-service): vRO workflows can be published in vRA and invoked directly from the service catalog by the end users, allowing direct consumption. XaaS can also be used to embed vRO workflows directly within a converged blueprint, allowing potentially unlimited extensibility use cases. For example, a Day 2 operation that adds additional NAT rules on an existing NSX Edge could be defined using XaaS and published in the service catalog, allowing end users to invoke it against a previously deployed application.

As both extensibility approaches leverage vRealize Orchestrator as the underlying layer, the same content (vRO workflows and actions) can be reused across both options.

Additional details about the NSX plugin for vRO and how it’s possible to consume the NSX API via the REST plugin are provided in the section related to vRealize Orchestrator.

References

NSX REST API

VMware vRealize Orchestrator Documentation