TECHNICAL WHITE PAPER – SEPTEMBER 2019

NSX Policy API: Getting Started Guide

Version 1.0

vmware®

Table of contents

Overview

NSX-T release 2.4 introduced a new object model to simplify and automate network and security configurations through outcome driven statements. The resulting new Policy APIs reduces the number of configuration steps by allowing users to describe the desired end-goal while letting the system figure out how best to achieve it. The Policy API model can be used to create the entire intent in one go in an order-independent prescriptive manner. This document provides a quick guide to understand the new Policy API model, covers the consumption and talks about the hierarchical API it provides. The Policy API should not be confused with a Security Policy (in a DFW context).

Policy APIs provide a simplified data model and allow for consumption using an intent-based approach. It uses a declarative API model and can be used to create the entire intent in one go without caring about ordering or having to make multiple API calls. The intent is expressed using the parent/child relationship between the objects. The data model guarantees that a parent object is created before its child object making it order independent. The model also allows for a user defined Object ID to be specified whilst creating an object. Each object can be referenced by providing the full path of the object hierarchy.

The APIs operate on Policy Objects and are provided under the hierarchical API endpoint:

/policy/api/v1/infra/

A high-level overview of the object hierarchy is shown below:

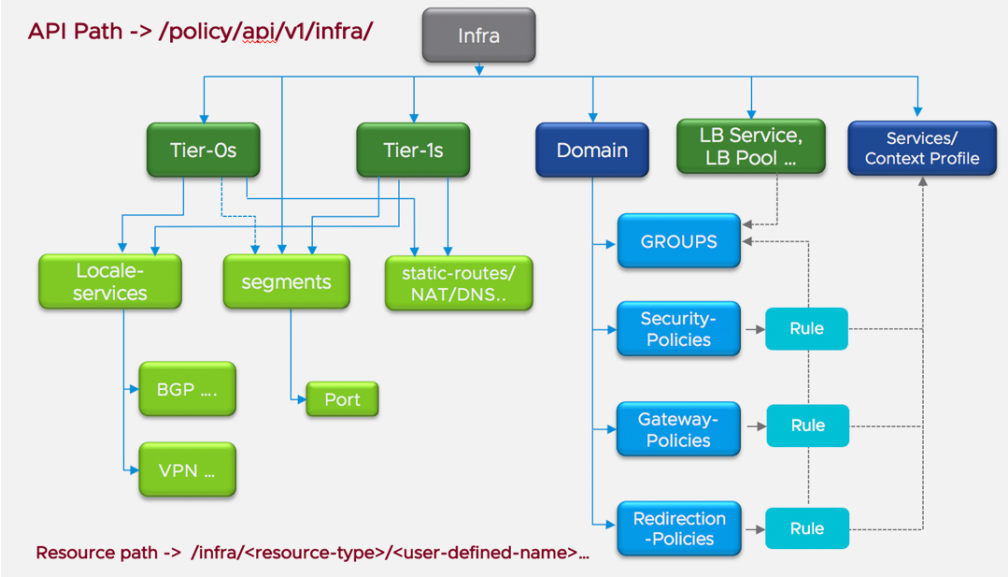

Figure 1.1: High-level object hierarchy diagram

As seen in the diagram, referring to specific object follows the tree structure. For example, referring to a Rule would be through:

```
/infra/domain/<domain-id>/security-policies/<security-policies>/rules/<rule-id>
```

Note that its not the API path but the path to the resource. The API path would be

```
/policy/api/v1/infra/domain/<domain-id>/security-policies/<security-policies>/rules/<rule-id>
```

Usage

From NSX-T 2.4 release, users interact with the NSX Manager using the Simplified UI. The traditional objects will be available under the Advanced UI.

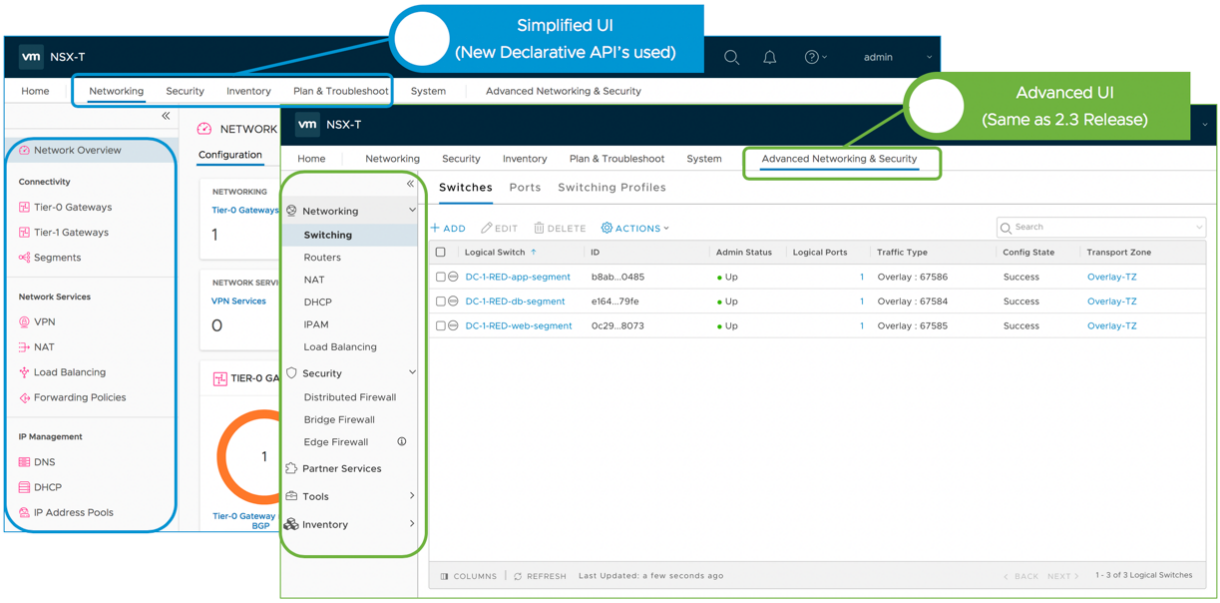

Figure 2.1: Simplified and Advanced UI

The Simplified UI is a new interface introduced from NSX-T 2.4 that uses the declarative API Policy model. This interface should be used to manage NSX-T.

The objects created in NSX-T 2.3 and before will be exposed in the Advanced UI section.

There is a very clear separation of function that is defined between objects created with Policy APIs vs objects created using traditional MP APIs:

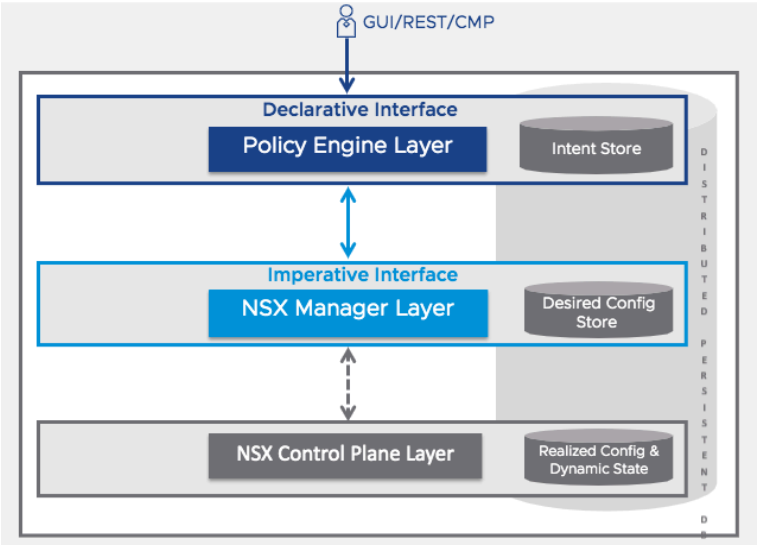

Figure 2.2: Separation of functions

Objects created using Policy API will be visible as the Management Plane (MP) objects after realization. A call to Policy API defines an intended state and realization happens in MP. Note that objects created using MP API are not visible to Policy API and cannot be seen in the Simplified UI.

Today, Policy objects refer to logical constructions like Tier-0/Tier-1 Gateways, Segments, Distributed Firewall, Gateway Firewalls, Load Balancers, DHCP, NAT etc. They should not be confused with system objects like Transport Zones, Transport Nodes, Edge Clusters.

Naming Conventions

With the introduction of Policy APIs, the new objects are referred with a new naming scheme. The table below lists the new Policy constructs and their corresponding MP constructs. The MP constructs are still valid and are used when using the traditional (MP) APIs

Existing Construct	Policy Construct	Definition
Logical Switch	Segment	A network entity equivalent to Logical Switch
T1 Logical Router	Tier-1 Gateway	This is equivalent to the T1 Router and allows the topology to scale out. Multiple Tier-1 Gateways talk to the Tier-0 Gateway
T0 Logical Router	Tier-0 Gateway	Equivalent to the T0 Logical Router and allows Tier-1s to talk to the outside world
NSGroups, IP Sets, MAC Sets	Group	Grouping construct to statically or dynamically group different objects. This could be inventory entities like IPs, VMs, MACs etc
Firewall Section	Security Policy	A section to encompass various security policies. Each Security Policy will have a set of Firewall Rules
Firewall Rule	Rule	A structure to encompass various firewall policies
Edge Firewall	Gateway Firewall	Tier-0/Tier-1 Edge Firewall capabilities for North-South connectivity

Policy API Data Model

Basic Structure

Policy APIs are built as regular REST APIs. They support the traditional GET/PUT/DELETE calls. Note that there are no POST calls. Where applicable, the calls accept a JSON body formatted in a specific way (as defined by the schema in the API Guide) and return a code to indicate success or failure. However, they also implement PATCH API calls. Some of the similarities and differences are discussed more in the Operations section below.

The next change which has a big impact is the ability to use User Defined IDs. These can be alpha numeric and is also used as the display_name if it is not specified. This allows for objects to be identified and searched for easily. It allows for the Object ID/Object Name to be from the URI. Consider a simple API call:

```
PATCH /policy/api/v1/tier-0s/MyTier0
```

In the above, the "MyTier0" is used as the Object ID and the display name and the fields need not be specified in the request body. The below *empty* request body is completely valid to create a new Tier-0 Gateway (with default values for its members)

```
PATCH /policy/api/v1/tier-0s/MyTier0

{ }
```

It is valid because the resource type and Object ID/Object Name is inferred from the URI itself. With these two fields known, rest of the fields use the default values.

There are 2 objects that needs a special callout; *Infra* and *Domain*. As we see in *Figure 1.1*, Infra is the root object. It is a system owned object and cannot be deleted or modified. It has to be passed while using the hierarchical api (/policy/api/v1/infra) and has all other objects as its child objects including the "default" domain object. The "default" domain object is the parent object of certain members like SecurityPolicy and like the Infra object, cannot be deleted. It is used only in the context of the hierarchical API while creating, modifying or deleting its child objects.

It is important to note that the "marked_for_delete" flag should not be set to "true" for Infra and "default" Domain objects. In the case of the "default" Domain object, the child members like SecurityPolicy or Groups have the "marked_for_delete" flag which has to be set to "true" for deletion. More on the "marked_for_delete" flag in the next section. Reasons: System created objects can't be deleted by user.

The Policy API model has certain other features that make it order independent and declarative in nature. Some of the salient properties are described below

Parent-Child hierarchy

The API model follow a parent-child hierarchical tree structure. Based on the object model described above Figure 1.1. The complete list of child objects is listed in the API Guide in the schema definition. Multiple objects can be defined as a nested parent-child tree.

Example:

```
    {
      "resource_type": "ChildTier1",
      "Tier1": {
        "resource_type": "Tier-1",
        "id": "MyTier",
        "children": [
          {
            "resource_type": "ChildSegment",
            "Segment": {
              "resource_type": "Segment",
              "type": "DISCONNECTED",
              "connectivity_path": "/infra/tier-1s/MyTier1",
              "transport_zone_path": "/infra/sites/default/enforcement-points/default/transport-
zones/664ba01c-815d-48ba-a7e0-8ff1d928db50",
              "id": "MySegment",
              "children": []
            },
          }
        ]
      }
    }
```

The example above shows a Tier-1 Router with a Segment as its child object. Only a few fields are shown for simplicity.

Custom Object ID

The Policy API model allows for user defined Object IDs. The object ID is also used as a display name in case one is not provided. Not only does this make objects easily identifiable, it also makes it extremely easy to create an object path. In the example above, referring to the Segment is as easy as forming the path "*/infra/tier-1s/MyTier1/segments/MySegment*"

Hierarchical API

The object model includes a powerful hierarchical API endpoint **/policy/api/v1/infra**. This endpoint is the root object and provides a means to interact with multiple objects (peers as well as children). This API endpoint can also be used to fetch the entire config ("*GET /policy/api/v1/infra?filter=Type-*"). CRUD (Create/Read/Update/Delete) operations can be performed on this endpoint by defining the desired end state as the request body. Using the hierarchical API gives you the ability to create an entire topology using one PATCH API call. The desired outcome can be specified in a human-readable JSON and submitted as the request body and sent via the PATCH call to the */policy/api/v1/infra* endpoint.

Below are some of the properties that are important when working with hierarchical API:

Wrapper Objects

All objects are nested inside an intermediary node of a specific object type. These wrappers have a property named "*resource_type*" containing the actual object configuration.

Lets look at the simple example again:

```
{
  "resource_type": "Infra",
  "id": "infra",
  "children": [
    {
      "resource_type": "ChildTier1",
      "Tier1": {
        "resource_type": "MyTier1",
        "id": "Tier-1",
        "children": [
          {
            "resource_type": "ChildSegment",
            "Segment": {
              "resource_type": "Segment",
              "type": "DISCONNECTED",
              "connectivity_path": "/infra/tier-1s/MyTier1",
              "transport_zone_path": "/infra/sites/default/enforcement-points/default/transport-
zones/664ba01c-815d-48ba-a7e0-8ff1d928db50",
              "id": "MySegment",
              "children": []
            },
          }
        ]
      }
    }
  ]
```

```
}
```

In the above snippet, the child objects of the infra node is not the actual Tier1 or Segment. Instead, they are nested inside an intermediary node of type *ChildTier1* and *ChildSegment*. These wrappers contain the actual object configuration.

Marked for Delete Flag

The wrapper object has a special purpose field *marked_for_delete*. Setting it to "true" deletes the parent object and all its child objects. This can be useful while deleting an entire topology.

In the example below, the "marked_for_delete" flag is set to "true". So submitting the below JSON request body along with a PATCH /policy/api/v1/infra/ call deletes the Tier-1 object and the associated Segment.

```
{
  "resource_type": "Infra",
  "id": "infra",
  "children": [
    {
      "resource_type": "ChildTier1",
      "marked_for_delete": "true",
      "Tier1": {
        "resource_type": "MyTier1",
        "id": "Tier-1",
        "children": [
          {
            "resource_type": "ChildSegment",
            "marked_for_delete": "false",
            "Segment": {
              "resource_type": "Segment",
              "type": "DISCONNECTED",
              "connectivity_path": "/infra/tier-1s/MyTier1",
              "transport_zone_path": "/infra/sites/default/enforcement-points/default/transport-
zones/664ba01c-815d-48ba-a7e0-8ff1d928db50",
              "id": "MySegment",
              "children": []
            },
          }
        ]
      }
    }
  ]
}
```

In the above example, the "marked_for_delete" flag is set to "false" for Segment child object. Since the parent Tier1 object as the flag set to "true", both the Tier1 and the Segment will be deleted.

The “marked_for_delete” flag can be added to any object – except “Infra” and “default” domain. These two are system default objects and cannot be deleted. The flag is not a mandatory field and default value is “false”. It can be added when needed or can always be present for completeness. When dealing with Domain object, the “marked_for_delete” should be used on its child objects (example: SecurityPolicy). In cases when “marked_for_delete” has a different value between the parent and child, the parent value takes precedence.

Note that this field is honored only when used under the /policy/api/v1/infra hierarchical API endpoint. Setting the ‘marked_for_delete’ to ‘true’ while referring to a specific object path does not delete the object. In such a case (deleting at a specific object path), the DELETE API has to be used.

Resource Type

The resource type serves dual purpose within the context of the hierarchical API. Its type varies when used within the wrapper object and when it is part of the inner object. As seen below the values for *resource_type* change based on its position.

```
{
  "resource_type": "Infra",
  .......
   “children”: [
  {
   "resource_type": "ChildTier1",
   "Tier1": {
      "resource_type": "Tier1",
      "id": "Tier-1-GW-PROD",
      "tier0_path":"/infra/tier-0s/Tier-0-GW-BM-01",
      "route_advertisement_types": [
               "TIER1_CONNECTED"
       ],
       "children": [
          {
            ... Define Segment WEB as child Segment object ...
          },
       .......
    }
  ]
}
```

In the parent wrapper object, it represents an object of type *ChildPolicyConfigResource* where as when inside the child object, it represents a *string* and based on the object being defined, the value is set – reserved string “Tier1” in the above example.

For normal operations, the resource type is of *ChildPolicyConfigResource*. The complete list of supported values are listed in the API Guide. When used this way, normal CRUD operations can be performed on the parent object and all its child objects. There are cases when we need to protect the parent object but allow CRUD operations to the child objects. In such a case, a special resource type *ChildResourceReference* can be used. This references a parent resource but protects it from the CRUD operations.

In the example below, the highlighted section defines a parent resource ChildTier1 of type *ChildPolicyConfigResource* and normal CRUD operations can be performed on it.

```
{
```

```
  "resource_type": "Infra",
  .......
   "children": [
  {
   "resource_type": "ChildTier1",
   "Tier1": {
      "resource_type": "Tier1",
      "id": "Tier-1-GW-PROD",
      "tier0_path":"/infra/tier-0s/Tier-0-GW-BM-01",
      "route_advertisement_types": [
               "TIER1_CONNECTED"
       ],
       "children": [
          {
            ... Define Segment WEB as child Segment object ...
          },
        .......
    }
  ]
}
```

In the example blow, the special resource type "ChildResourceReference" is used, along with the field "target_type" identifying the object, which protects the parent object from CRUD operations:

```
{
  "resource_type": "Infra",
  .......
   "children": [
  {
   "resource_type": "ChildResourceReference",
   "id": "Tier-1-GW-PROD",
   "target_type": "Tier1",
   "children": [
     {
        ... Define Segment WEB as child Segment object ...
     },
      .......
  ]
}
```

Concurrency Control

In order to prevent one client from overwriting another client's updates, NSX-T employs a technique called optimistic concurrency control.

All REST payloads contain a property named "*_revision*". This is an integer that is incremented each time an existing resource is updated. Clients must provide this property in PUT requests and it must match the current *_revision* or the update will be rejected. This guards against the following situation:

- Client 1 reads resource A.
- Client 2 reads resource A.
- Client 1 replaces the display_name property of resource A and does a PUT to replace the resource.
- Client 2 replaces a different property of resource A and attempts to perform a PUT operation.
- Without optimistic concurrency control, Client 2's update would overwrite Client 1's update to the display_name property. Instead, Client 2 receives a 409 Conflict error. To recover, Client 2 must fetch the resource again, apply the change, and perform a PUT.

Policy APIs also accept PATCH REST calls, which by default does not require that the *_revision* property be provided. A client can request that the *_request* property be checked when it is performing a PATCH in the /infra path. To do this, the client should pass the query parameter *enforce_revision_check*. Example.

```
PATCH /policy/api/v1/infra?enforce_revision_check=true
```

Operations

This section talks about various API methods and their role in operations.

Creation (PATCH/PUT)

The Policy APIs support the use of a PATCH call to interact with the objects. When a PATCH call is used, a new object is created if one doesn't exist and if the object already exists, then it updates the object. Objects are updated as per the provided JSON request body regardless of its current configuration.

The PUT call can also be used to create an object. For a successful edit of an object using PUT calls, the *_revision* field has to specified and the number has to match the existing revision (typically got using a GET before the PUT).

The following table shows the behavior when different options are used together:

HTTP Method	_revision passed	ID already exists	Output
PUT	No	No	Create Object
PUT	No	Yes	Error – Object Exists
PUT	Yes	Yes	Update Object
PUT	Yes	No	Error – Object Not Found
PATCH	Ignore	Yes	Update/Merge
PATCH	Ignore	No	Create

Below are different options to create the Tier-0 Gateway:

Example 1: Direct reference to the object with PATCH:

```
PATCH /policy/api/v1/infra/tier-0s/MyTier0

{
    "resource_type": "Tier0",
    "id": "MyTier0",
    "transit_subnets": [ "10.2.3.0/24" ],
     "ha_mode": "ACTIVE_STANDBY"
}
```

Note the reference to the object path in the URI

Example 2: Direct reference to the object with PUT:

```
PUT /policy/api/v1/infra/tier-0s/MyTier0

{
    "resource_type": "Tier0",
    "id": "MyTier0",
    "transit_subnets": [ "10.2.3.0/24" ],
     "ha_mode": "ACTIVE_STANDBY"
}
```

Note that "_revision" is not sent in the request body. This tells the PUT operation to create the object. However, it has to be sent for any future edit operations. A GET call is required to get the current revision number which is then used in the PUT call

Also, since Policy APIs allow user defined IDs, just specifying the name of the object in the URI is sufficient. The "id" need not be sent again. In both PATCH or PUT, the request body can just be this:

```
{
    "transit_subnets": [ "10.2.3.0/24" ],
     "ha_mode": "ACTIVE_STANDBY"
}
```

There is no "resource_type" or "id" as both are identified by the URI below. The "tier-0s" tell that it's a Tier-0 resource and the "MyTier0" define its display_name/ID.

```
/policy/api/v1/infra/tier-0s/MyTier0
```

Example 3: Creation using Hierarchical API:

```
PATCH /policy/api/v1/infra
```

```
{
    "resource_type": "Infra",
    "children": [{
            "resource_type": "ChildTier0",
            "marked_for_delete": "false",
            "Tier0": {
                "resource_type": "Tier0",
                "id": "MyTier0",
                "transit_subnets": [ "10.2.3.0/24" ],
                "ha_mode": "ACTIVE_STANDBY"
            }
    }]
}
```

Note that the URI only refers to the infra object. In this case, the "ID" and "resource_type" has to be provided as the URI is generic.

Deletion (DELETE/PATCH)

Deletion of objects can be done by referencing the object path directly or by using the hierarchical API. In the case of the hierarchical API, the PATCH call with the *marked_for_delete* set to 'true' deletes the object(s).

Example: Delete an object by direct reference:

```
DELETE /policy/api/v1/infra/tier-0s/MyTier0
```

Example: Deletion using hierarchical API endpoint with PATCH:

```
PATCH /policy/api/v1/infra/
{
    "resource_type": "Infra",
    "children": [{
            "resource_type": "ChildTier0",
            "marked_for_delete": "true",
            "Tier0": {
                "resource_type": "Tier0",
                "id": "MyTier0",
                "transit_subnets": [ "10.2.3.0/24" ],
                "ha_mode": "ACTIVE_STANDBY"
            }
    }]
}
```

Retrieving Configs (GET)

Object configuration can be retrieved using the standard GET operations. The configuration can be retrieved on a specific object by providing the direct object path or via the hierarchical API. When using the hierarchical API, getting the entire configuration requires a special filter to the URI.

Example to retrieve Tier-0 config using direct object path reference:

```
GET /policy/api/v1/infra/Tier-0s/MyTier0
Response:
{
  "transit_subnets": [
    "10.2.3.0/24"
  ],
  "internal_transit_subnets": [
    "169.254.0.0/28"
  ],
  "ha_mode": "ACTIVE_STANDBY",
  "failover_mode": "NON_PREEMPTIVE",
  "ipv6_profile_paths": [
    "/infra/ipv6-ndra-profiles/default",
    "/infra/ipv6-dad-profiles/default"
  ],
  "force_whitelisting": false,
  "default_rule_logging": false,
  "disable_firewall": false,
  "resource_type": "Tier0",
  "id": "MyTier0",
  "display_name": "MyTier0",
  "path": "/infra/tier-0s/MyTier0",
  "relative_path": "MyTier0",
  "parent_path": "/infra/tier-0s/MyTier0",
  "marked_for_delete": false,
  "_system_owned": false,
  "_create_user": "admin",
  "_create_time": 1568679488993,
  "_last_modified_user": "admin",
  "_last_modified_time": 1568679488993,
  "_protection": "NOT_PROTECTED",
  "_revision": 0
}
```

Example to retrieve Tier-0 config using hierarchical API:

```
GET /policy/api/v1/infra?filter=type-Tier0
```

```
{
  "resource_type": "Infra",
  "id": "infra",
  "display_name": "infra",
  "path": "/infra",
  "relative_path": "infra",
  "children": [
    {
      "Tier0": {
        "transit_subnets": [
          "10.2.3.0/24"
        ],
        "internal_transit_subnets": [
          "169.254.0.0/28"
        ],
        "ha_mode": "ACTIVE_STANDBY",
        "failover_mode": "NON_PREEMPTIVE",
        "ipv6_profile_paths": [
          "/infra/ipv6-ndra-profiles/default",
          "/infra/ipv6-dad-profiles/default"
        ],
        "force_whitelisting": false,
        "default_rule_logging": false,
        "disable_firewall": false,
        "resource_type": "Tier0",
        "id": "MyTier0",
        "display_name": "MyTier0",
        "path": "/infra/tier-0s/MyTier0",
        "relative_path": "MyTier0",
        "parent_path": "/infra/tier-0s/MyTier0",
        "children": [],
        "marked_for_delete": false,
        "_system_owned": false,
        "_create_user": "admin",
        "_create_time": 1568679488993,
        "_last_modified_user": "admin",
        "_last_modified_time": 1568679488993,
        "_protection": "NOT_PROTECTED",
        "_revision": 0
      },
      "resource_type": "ChildTier0",
```

```
      "marked_for_delete": false,
      "_protection": "NOT_PROTECTED"
    }
  ],
  "marked_for_delete": false,
  "connectivity_strategy": "BLACKLIST",
  "_system_owned": false,
  "_create_user": "system",
  "_create_time": 1567727520685,
  "_last_modified_user": "system",
  "_last_modified_time": 1567727520685,
  "_protection": "NOT_PROTECTED",
  "_revision": 0
}
```

Note the change in the URI and that in this case information about the parent object “infra” is also returned.

The retrieve the entire configuration of the system:

```
GET /policy/api/v1/infra?filter=Type-
```

The above returns all the configuration including the system generated ones. Expect it to take a few seconds depending on the size of the system configuration.

Some of the other filter examples are:

```
Retrieve all Tier-1s:
GET /policy/api/v1/infra?filter=Type-Tier1

Retrieve all Tier-1s and Segments (both flexible and fixed):
GET /policy/api/v1/infra?filter=Type-Tier1|Segment

To retrieve basic Tier-0 config:
GET /policy/api/v1/infra?filter=Type-Tier0|LocaleServices|Bgp|Tier0Interface|PolicyNat|PrefixList

Retrieve basic Tier-1 config:
GET /policy/api/v1/infra?filter=Type-Tier1|LocaleServices|PolicyNat

Retrieve whole intent tree except Services, LB, DHCP and MAC:
GET /policy/api/v1/infra?filter=Type--(?i)^(?!(?:Service|LB|Dhcp|Mac)).*$
```

Importing API Spec

You can download the OpenAPI specifications for the NSX-T Policy APIs at the following URIs:

```
GET /api/v1/spec/openapi/nsx_policy_api.yaml
GET /api/v1/spec/openapi/nsx_policy_api.json
```

The API spec can then be imported into tools like Postman or SDKs can be generated using tools like swagger-codegen.

Realization

When an API to create an object in Policy is called (through PATCH or PUT), a successful 200 return code ensures that the system knows about the intent. Actual realization happens in the system and can take longer. This could be due to the number of objects being created using the hierarchical API. In such a case, alarms are generated, and the following APIs can be used to check them:

```
GET /policy/api/v1/infra/realized-state/alarms
```

Every configured item will have an "intent path" or just "path". Examples are:

```
/infra/domains/default/groups/WEB
/infra/ip-pools/TEP-pool
```

The following 2 APIs can be used to validate if the intent has been realized on the configured endpoint:

```
Realization status: Used to retrieve the overall realization status of the specified intent path
GET /policy/api/v1/infra/realized-state/status?intent_path=<intent_path>

Realized entities: Used to retrieve the realized objects on the endpoint for the specified intent
GET /policy/api/v1/infra/realized-state/realized-entities?intent_path=<intent_path>
```

The status of the Policy service itself can be retrieved using the API:

```
GET /api/v1/node/services/policy/status
```

FAQs

Some frequently asked questions:

Q. Can I use the 'marked_for_delete' flag to delete objects by referring to them using the full path (Example: /policy/api/v1/infra/tier-0s/MyTier0)

A. No. The 'marked_for_delete' flag is treated as a read-only field in this case. The field is honored only when used with the hierarchical API endpint (/policy/api/v1/infra)

Q. How can I retrieve a full config from the NSX Manager?

A. GET /policy/api/v1/infra?filter=Type-

Q. Can I use the full config retrieved to do a backup/restore workflow?

A. Yes. However, retrieving the full config also returns the system owned objects. These objects have to be removed from the JSON request body before sending it through the PATCH request. All system owned objects can be identified with:

"_system_owned": true,

Building a Full Topology

This section describes building of a complete example Topology using the Policy API model. Consider a typical 3-tier topology:

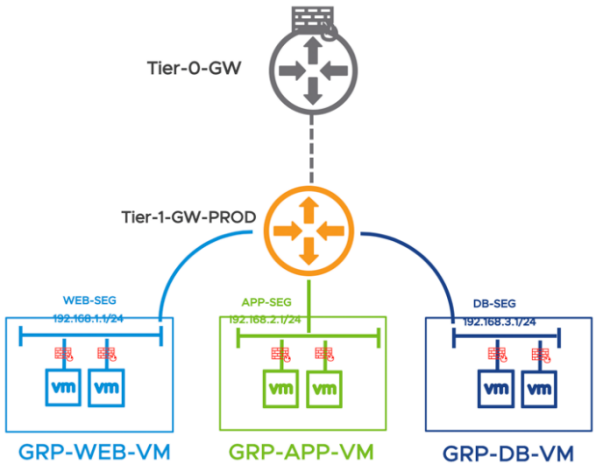

Figure 8.1: Typical 3-tier topology

The above topology consists of an existing Tier-0 gateway. This provides the North-South connectivity. To this is connected a Tier-1 Gateway. There are 3 Segments (Web, App and DB) connected to the Tier-1 Gateway. To satisfy the need for micro-segmentation, there are DFW rules specified on each Segment Port. Dynamic Groups are also created to group VMs with similar tags and these dynamic groups are used in the DFW rules.

To build such a topology, the first step is to create the Tier-1 Gateway.

```
PATCH (or PUT) /policy/api/v1/infra/tier-1s/Tier-1-GW-PROD

{

    "resource_type": "Tier-1",

    "id": "Tier-1-GW-PROD",

    "tier0_path": "/infra/tier-0s/Tier-0-GW",

    "route_advertisement_types": [

        "TIER1_CONNECTED"

    ]

}
```

Next, create a Segment.

```
PATCH (or PUT) /policy/api/v1/infra/segments/WEB-SEG
```

```
{
    “resource_type”: “Segment”,
    “id”: “WEB-SEG”,
    “subnets”: [
        {
            “gateway_address”: “192.168.1.1/24”
        }
    ],
    “connectivity_path”: “/infra/tier-1s/Tier-1-GW-PROD”,
    “transport_zone_path”: “/infra/sites/default/enforcement-points/default/transport-zones/{{tz-uuid}}
}
```

However, the above 2 steps can be combined into a single API call using the hierarchical API:

```
PATCH /policy/api/v1/infra
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildTier1",
      "marked_for_delete": false,
      "Tier1": {
         ...  Tier-1 object parameters ...  ,
        "children": [
          {
            "resource_type": "ChildSegment",
             "marked_for_delete": false,
            "Segment": {
             ...  Segment WEB-SEG object parameters ...
            }
          }
        ]
      }
    }
  ]
```

```
}
```

Note that in case, a fixed segment is being created.

Extending this, we can now create the Tier-1 Gateway and the 3 Segments together to get the following topology at once!

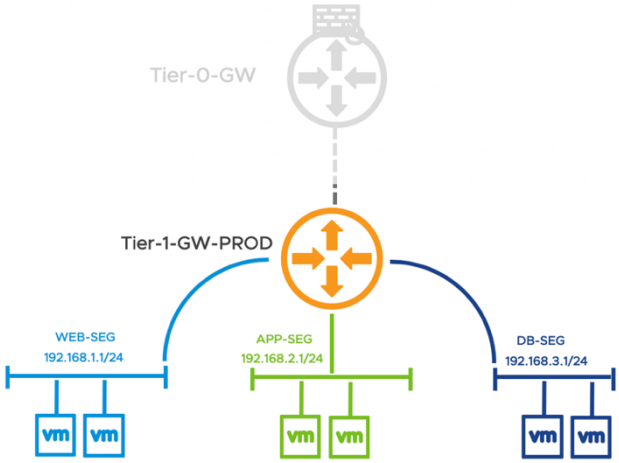

Figure 8.2: Create the Tier-1 and Segments together

```
PATCH /policy/api/v1/infra

{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildTier1",
      "marked_for_delete": false,
      "Tier1": {
         ...  Tier-1 object parameters ...  ,
        "children": [
          {
              "resource_type": "ChildSegment",
              "Segment": {
                  ... WEB-SEG object parameters ...
              }
          },
          {
              "resource_type": "ChildSegment",
```

```
                "Segment": {
                    ... APP-SEG object parameters ...
                }
            },
            {
                "resource_type": "ChildSegment",
                "Segment": {
                    ... DB-SEG object parameters ...
                }
            }
        ]
      }
    }
  ]
}
```

Now, let's look at creating a dynamic security group based on the VM tags. Of course, this can be done by creating a Group using the direct API:

```
PATCH /policy/api/v1/infra/domains/default/groups/GRP-WEB-VM
```

But, doing it through hierarchical API is more interesting. At

```
PATCH /policy/api/v1/infra/
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildDomain",
      "Domain": {
        "id": "default",
        "resource_type": "Domain",
        "children": [
          {
            "resource_type": "ChildGroup",
            "marked_for_delete": "false",
```

```
            "Group": {
              "resource_type": "Group",
              "id": "GRP-WEB-VM",
              "expression": [
                {
                  "member_type": "VirtualMachine",
                  "value": "vmw-webvm",
                  "key": "Tag",
                  "operator": "EQUALS",
                  "resource_type": "Condition"
                }
              ]
            }
          }
      ]
    }
  }
 ]
}
```

Like with Segments, we can create all 3 Groups at once to get the following topology:

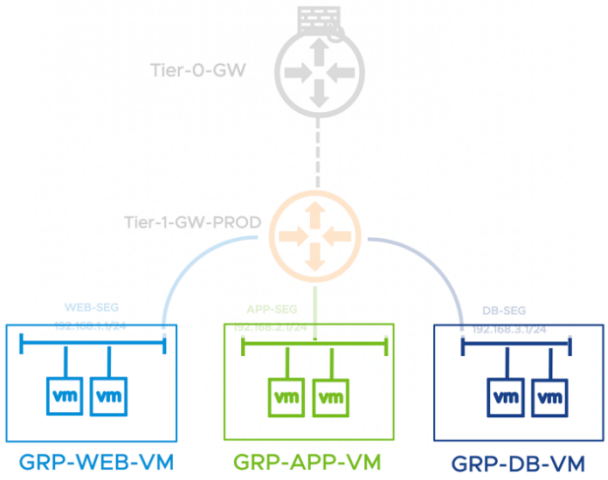

Figure 8.3: Creating dynamic Groups

```
PATCH /policy/api/v1/infra

{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildDomain",
      "Domain": {
        "id": "default",
        "resource_type": "Domain",
        "children": [
          {
             …… Define GROUP WEB VM as ChildGroup object……
          },
          {
            …… Define GROUP APP VM as ChildGroup object……
          },
          {
            …… Define GROUP DB VM as ChildGroup object……
          },
       ]
    }
   }
```

```
  ]
}
```

The next step is to define DFW rules. Again, we can do this via the hierarchical API:

```
PATCH /policy/api/v1/infra/
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildDomain",
      "Domain": {
        "id": "default",
        "resource_type": "Domain",
        "children": [
          {
            "resource_type": "ChildSecurityPolicy",
            "marked_for_delete": "false",
            "SecurityPolicy": {
              "id": "VMW-APP-Policy",
              "resource_type": "SecurityPolicy",
              "rules": [
                {
                  "resource_type": "Rule",
                  "display_name": "any-to-web",
                  "sequence_number": 1,
                  "source_groups": [
                    "ANY"
                  ],
                  "destination_groups": [
                    "/infra/domains/default/groups/GRP-WEB-VM"
                  ],
                  "services": [
                    "/infra/services/HTTPS"
                  ],
```

```
                    "action": "ALLOW"
                }
            ]
            }
    ...
}
```

And different DFW rules can be defined at once too!

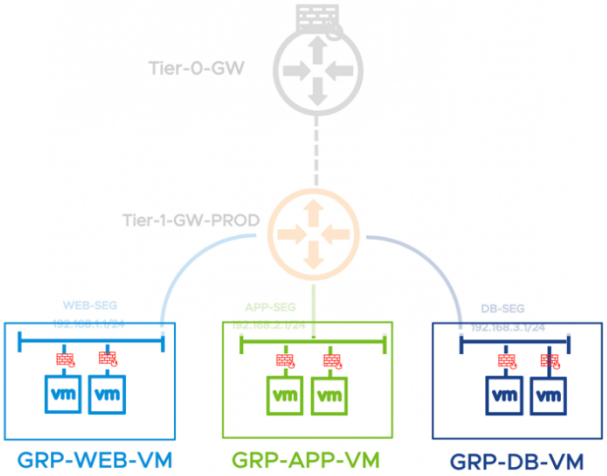

Figure 8.4: Security Policies

```
PATCH /policy/api/v1/infra/
{
  "resource_type": "Infra",
  "children": [
    {
      "resource_type": "ChildDomain",
      "Domain": {
        "id": "default",
        "resource_type": "Domain",
        "children": [
          {
            "resource_type": "ChildSecurityPolicy",
            "SecurityPolicy": {
              "id": "VMW-APP-Policy",
```

```
            "resource_type": "SecurityPolicy",
            "rules": [
              {
                ……..  Rule-1  object - Any to WEB allow HTTPS  ……..
              },
              {
                ……..  Rule-2  object - WEB to APP allow HTTP  ……..
              },
              {
                ……..  Rule-3  object - APP to DB allow MySQL  ……..
              }
              ]
            }
      ...
}
```

Although the above workflow gives us the topology we desire, putting all of it together, gives us the the entire topology – Tier-1, Segments, Groups and Security Policies in just one PATCH call!

```
PATCH /policy/api/v1/infra/
{
    "resource_type": "infra",
    "children": [
        {
            "resource_type": "ChildTier1",
            "Tier1": {
                ... Define Tier-1 Gateway object parameters ...
                "children": [
                    {    ... Define Segment WEB as child Segment object ...    },
                    {    ... Define Segment APP as child Segment object ...    },
                    {    ... Define Segment DB as child Segment object ...    },
                ]
            }
        },
        {
```

```
            "resource_type": "ChildDomain",
            "Domain": {
                "id": "default",
               "resource_type": "Domain",
                "children": [
                   {    ... Define GROUP WEB VM as ChildGroup object ...  },
                   {    ... Define GROUP APP VM as ChildGroup object ...    },
                   {    ... Define GROUP DB VM as ChildGroup object ...    },
                  {    ... Define Security Policy as ChildSecurityPolicy object ...   }
                ]
            }
        }
    ]
}
```

Just like creating the entire topology with one API call, the whole topology can be deleted by adding the "marked_for_delete" tag to the parent objects (Tier-1, Groups, Security Policy) and running the exact same PATCH API. Note that the "default" Domain is a system domain and cannot be deleted. Deleting its children (Groups, Security Policy) in the example above has to be done by putting the "marked_for_delete" in each child object.

Conclusion

Policy APIs gives an order independent method to define the intent of the entire topology in just one API call making operations very easy. Please refer to the API Guide for detailed object definitions and schemas.

Since the data model aims to capture every intent, the API is primarily a reflection of the data model itself. In general, query strings and operation inputs are not needed (For example, there are no APIs of the form /policy/api/v1/infra/…/action=xyz)

VMware, Inc. 3401 Hillview Avenue Palo Alto CA 94304 USA Tel 877-486-9273 Fax 650-427-5001 www.vmware.com.