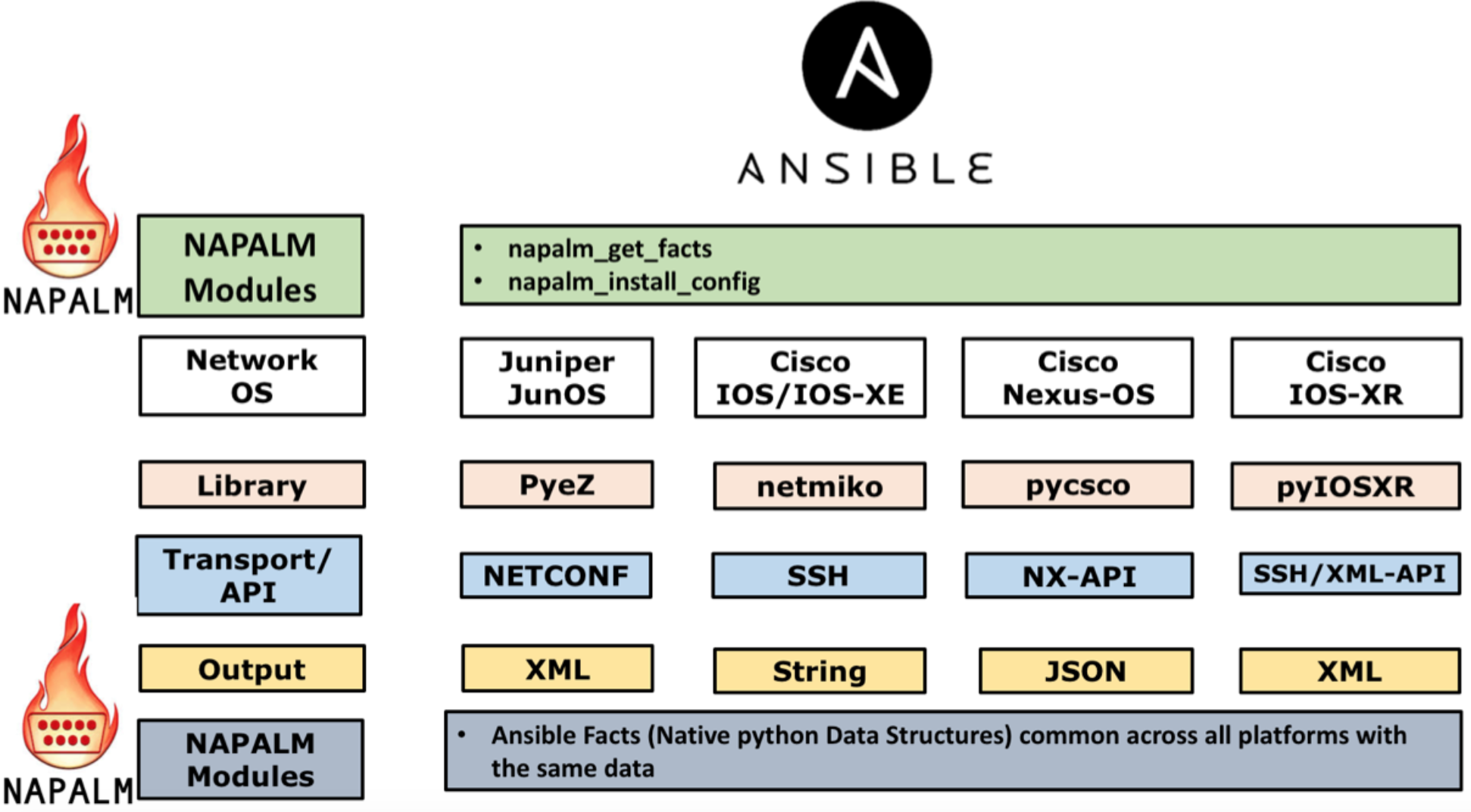
**Chapter Introduction**

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) as the name implies is a multi-vendor python library to interact with different vendor equipment and it provides a consistent method to interact with all these devices irrespective of the vendor equipment which is managed.

As we have shown in all the previous chapters how to interact with different network devices using ansible, however for each vendor OS we use a different ansible module which support this OS and also we saw that the data returned from each vendor OS is completely different. Although that writing playbook for multi-vendor devices is still possible however this require multiple different modules and we need to account for the different data structures returned by these devices. This is the main point that NAPALM tries to address. NAPALM tries to provide a similar ansible module to interact with multiple vendor OS and the data returned by NAPALM from these different vendor OS is normalized and is consistent.

NAPALM interact with each device according to the most common API supported by this node and the API which is widely adopted by the community. The below diagram outline how NAPALM interact with the most common Network devices and the libraries used in NAPALM to interact with these APIs on the devices



Since NAPALM tries to provide a similar and consistent method to interact with network equipment, it supports a specific vendor devices and also it support only the major and most common tasks that is carried on these devices like device configuration, retrieving Operational state for interfaces , BGP and LLDP and many other. For more information regarding the supported devices as well as the supported methods when interacting with these devices please check the below links

<https://napalm.readthedocs.io/en/latest/support/index.html>  
  
In this chapter, we will outline how to automate a multi-vendor Network using NAPALM and ansible.  
We wil outline how to manage the configuration on these different vendor OS as well as how to retrieve operational state from these devices.We will base our illustration based on the below sample network diagram of a basic SP network.



Below are the Software releases that this chapter is based on

* Ansible Machine Running Ubuntu 16.04
* Ansible 2.8
* Juniper vMX running JunOS 14.1R8 and JunOS 17.1R1 Release
* Cisco IOS-XR 6.1.2

The main recipes covered in this chapter is shown below

* Installing AWS SDK
* Authenticating to your AWS Account
* Building Ansible Inventory.
* Deploying VPCs.
* Deploying Subnets.
* Deploying IGW.
* Adjusting Routing Table.
* Deployment Validation using Ansible.
* Decommissioning Resources using Ansible

**Install AWS SDK**

In this recipe we outline how to the required python libraries needed to start interacting with AWS orchestration system using ansible. This step is mandatory since the python library must be installed on the ansible control machine in order for all the ansible AWS modules to work.

**Getting Ready**

You need to have sudo access on the machine in order to install the aws python library as well as have python already installed and python PIP package which we will use to install the aws package.

**How to do it..**

* We can test with any Ansible aws module to check if we required the python library or not

$ ansible localhost -m aws\_az\_facts  
  
localhost | FAILED! => {

"changed": false,

"msg": "boto3 required for this module"

}

* Install the boto3 package as shown below

$ sudo pip3 install boto3

**How it works..**

The python SDK library to interact with AWS Orchestration system API is **boto3** for python3, this library is mandatory to be present on the ansible control machine since all the ansible aws modules rely on this python package. We can check if this package is already installed or not on the system using the first step that we have outlined with running any aws module (**aws\_az\_facts** for example) using the ansible module. In case the boto3 library is not present we will get the error message that boto3 is not installed.

We install the boto3 package using the python-pip program using the pip3 command which will install the boto3 package and all the dependencies needed to install and run the package correctly. At this stage we have all the requirements to run all the Ansible AWS modules

**Building Ansible Inventory**

In this recipe, we will outline how to build an Ansible inventory to describe the infrastructure network setup that we will build across the AWS public. cloud This is a mandatory step in order to define all our VPCs across all the regions that will deploy our infrastructure in, this will allow us to group our infrastructure effectivity and group our variables according to these groupings.

**How to do it..**

* Create a new folder ch7\_aws and create the hosts file as shown below

$ cat hosts  
  
[us]

us\_prod\_vpc

[eu]

eu\_prod\_vpc

[prod\_vpcs]

us\_prod\_vpc

eu\_prod\_vpc

* Create the ansible.cfg file inside the ch7\_aws with the below contents

$ cat ansible.cfg  
  
[defaults]

inventory=./hosts

vault\_password\_file=~/.ansible\_vault\_passwd

gathering=explicit

transport=local

retry\_files\_enabled=False

action\_warnings=False

**How it works..**

We created the hosts ansible inventory file and we declare our VPCs as nodes in our inventory similar to how we define a network node. The only exception is that a VPC doesn’t have an management IP address so we don’t specify the ansible\_host argument for those VPCs.

We create the following groups in our inventory file which are:

* + **US** group which group all the VPCs in United States
  + **EU** group which group all the VPCs in Europe.
  + **prod\_vpcs** which group all our production VPCs.

We also define the ansible.cfg file which all the configuration options that we used in all the previous recipes. We specify the vault password file which will has the encryption password that we will use to encrypt all our passwords to authenticate to the AWS orchestration system.

**Authenticating to your AWS Account**

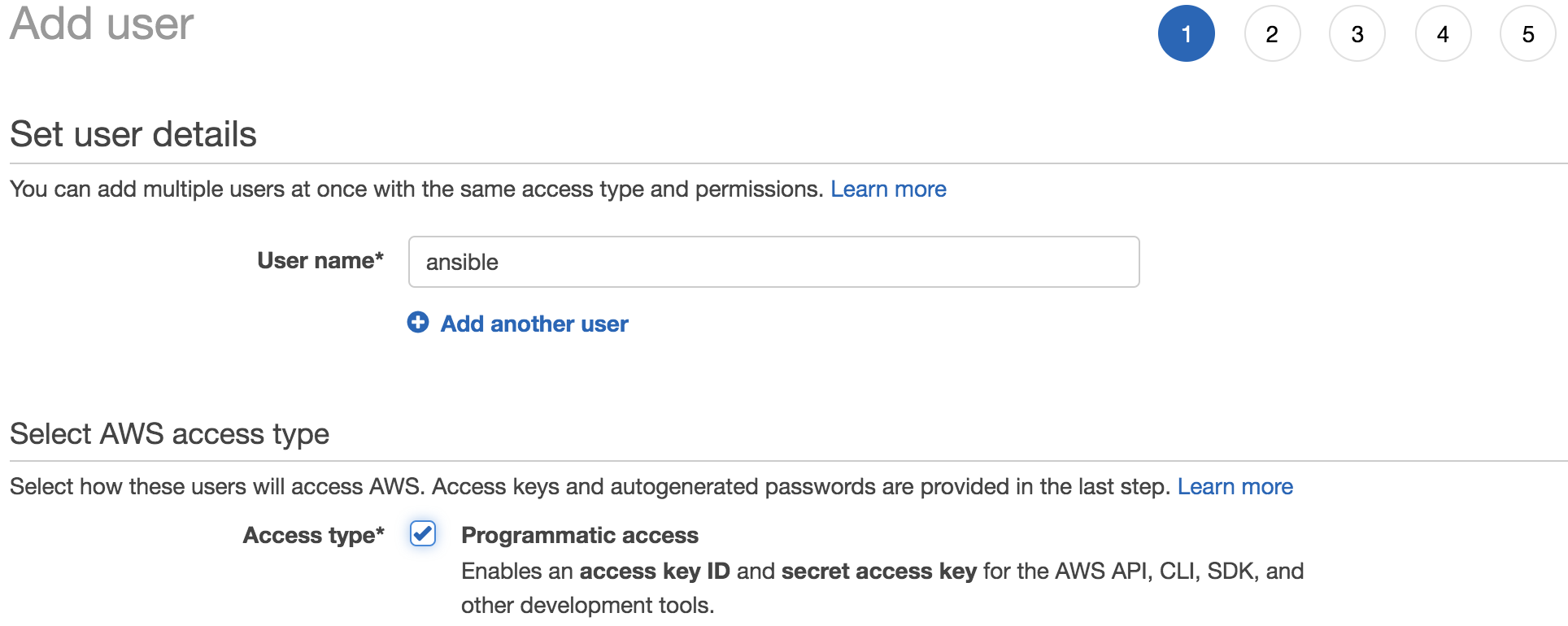
In this recipe, we will outline how to create the required credentials to programmatically authenticate to our AWS account from AWS and how to secure these credentials using ansible vault. This is a mandatory step in order to be able to run any ansible module in all the next recipes.    
  
https://955645556619.signin.aws.amazon.com/console

**Getting Ready**

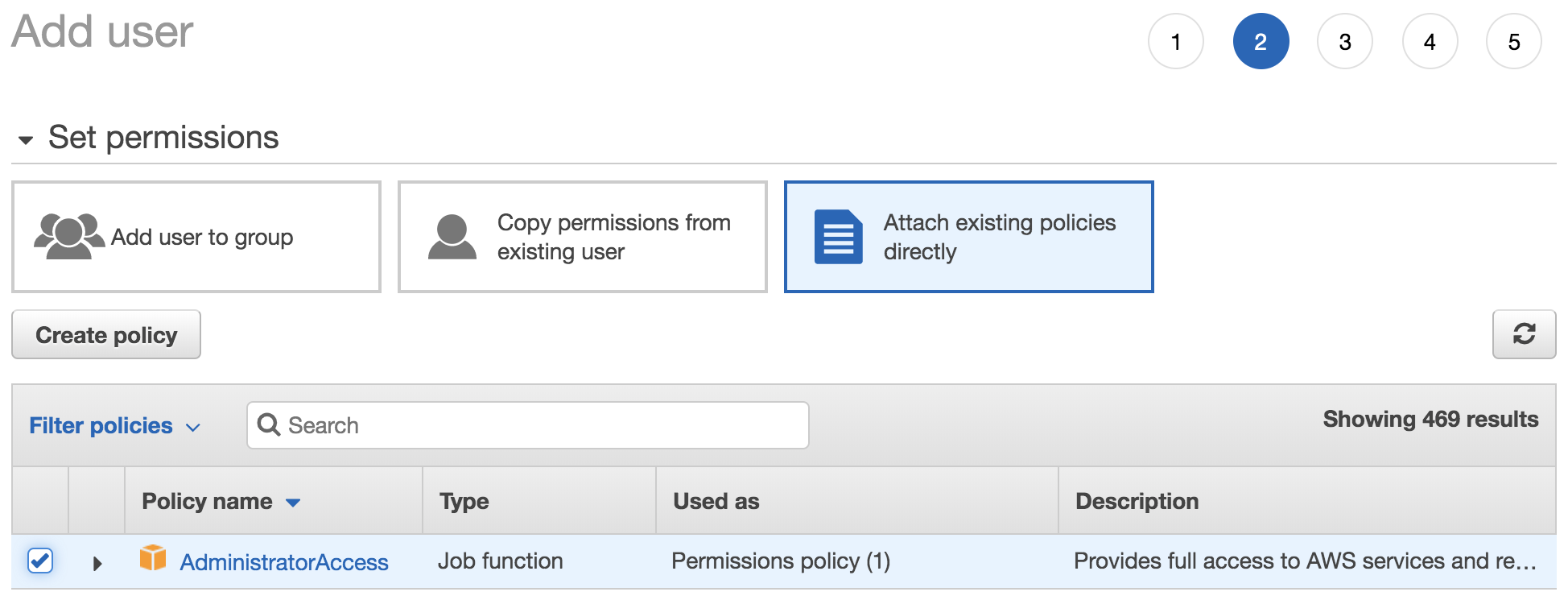
The Ansible controller must have internet access and the ansible inventory must be setup as outlined in the previous recipe. Also, the user performing these steps must be have programmatic access to the AWS console enabled by the administrator/root user for the AWS Account.

**How to do it..**

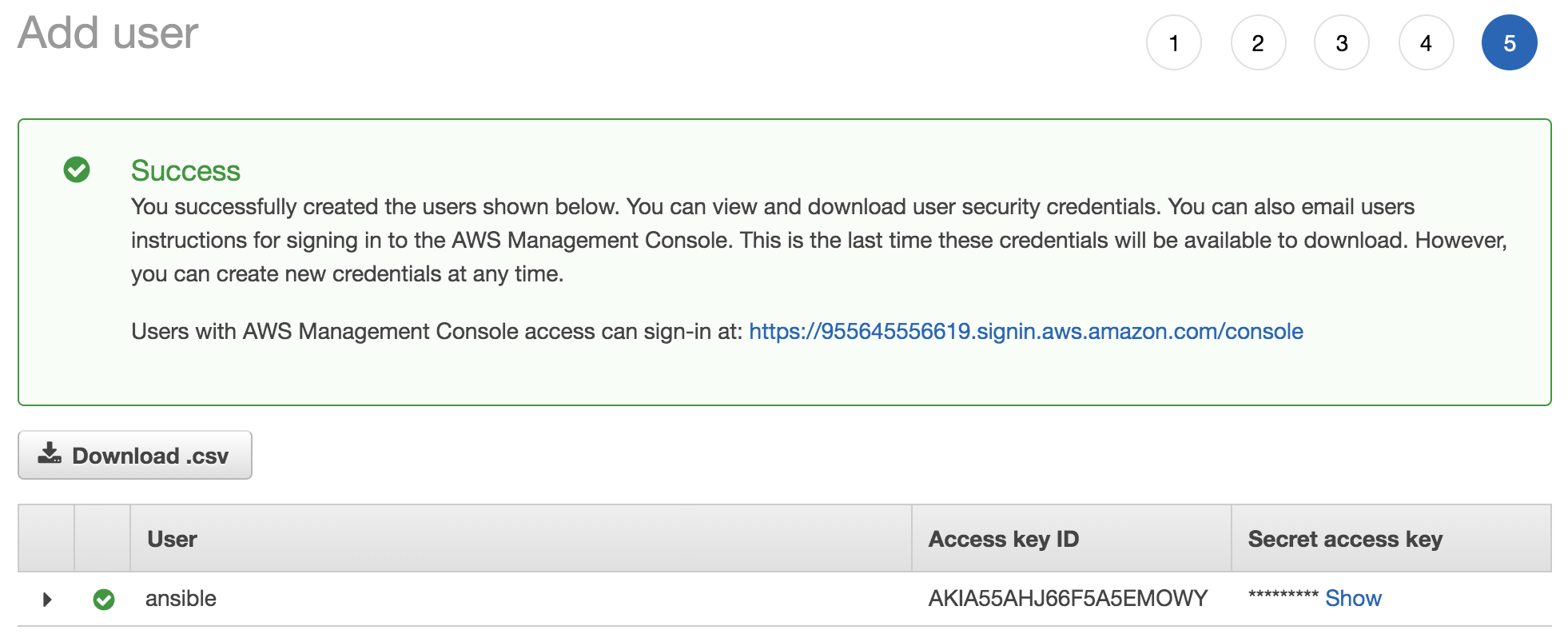
* An Admin User on the AWS Account, need to Create a new user using IAM with programmatic access as shown below.



* Assigned the correct IAM Policy to this new user which allows him to create all the Networking Resources that it should manage (or a Full Access Policy for Simplicity)



* Finish Creating the User and on the last page the Add User wizard will display the Access Key ID and Secret Access key in a csv file to download. These parameters will be used to authenticate to the AWS API for this account.



* Encrypt the access key ID and secret access key using ansible vault as shown below

$ ansible-vault encrypt\_string <ACCESS\_KEY\_ID> --name aws\_access\_key

$ ansible-vault encrypt\_string <SECRET\_ACCESS\_KEY> --name aws\_secret\_key\_id

* Create the group\_vars inside the ch7\_aws and create the all.yml file inside the group\_vars. Populate the all.yml file with the passwords encrypted using the ansible-vault in the previous step.

ansible\_connection: local

aws\_access\_key: !vault |

$ANSIBLE\_VAULT;1.1;AES256

37623631653336633662666138353639653365323637323665353731386661343164393664333434

3430306562623532366137663835636138613633633835660a656235363130303035383965663464

39326130613433643861653933623032393735376466333861326537646563643736356632303435

6631326531666461310a353461396431303765393830636432363430323438373635383132336462

37356163643662623633643965386465656563613533613938666232343063396261

aws\_secret\_key\_id: !vault |

$ANSIBLE\_VAULT;1.1;AES256

38353165666437393262303035646531666139623963643066623066346633393964633438626539

6266623937343036376266373463623266316462613139660a336664353564623531393332613433

34336363393962666633363630393631376135656666623862373966643935386665363733376133

6236326462326566320a653364336464363963623136363362666632396133613863376166343135

37373839316430643337353564373062333232656136393564613132333065316638383739326238

3530386534303033636463626664346234653136353534633265

**How it works..**

The first step is to ensure that a user account has programmatic access to the AWS console through the API. In order for a user to authenticate to the AWS API he must be assigned two passwords which are generated by AWS during user creation or when requesting to change these passwords. These two passwords are access and secret access key. These two passwords are only visible and available upon creation and AWS provide them in a CSV file that you download. Further, we need to ensure that this user has the correct IAM permission to create the needed resources (VPC, Subnets, Routing Tables, etc..). So, in our example this new user is assigned and Administrator Policy which gives him full access over the AWS account to create any resource (EC2 instances, VPCs, Subnets, etc..). The steps that we have outlined to create a new user is optional if a user is already present with programmatic access and the required IAM privileges , we just demonstrated this for completeness.

As we have the secrets generated by AWS for this account in the CSV file we take these password and encrypt them using ansible vault and store them in the **all.yml** file under group\_vars so that we use these credentials when we are creating all the resources for our two VPCs. We use the ansible-vault encrypt\_string command to generate the **aws\_access\_key** using the access\_key\_id data we get from the CSV file and we generate the **aws\_secret\_key\_id** using the secret\_access\_key data that we get from the CSV file. These variables will be encrypted using the vault password file that we have declared in the ansible.cfg file this file has the encryption password that we use to encrypt all these password.

In the next recipe we will outline how to use these encrypted variables that we have created to authenticate to the AWS console when creating the VPCs.

**Deploying VPCs using Ansible**

In this recipe we will outline how to deploy AWS VPCs using Ansible. AWS VPCs are the foundational networking construct and it can be thought as a virtual Data centre within the cloud that the Administrator create within his AWS account. In order to start building any other infrastructure related services within AWS, the VPC must be created first, we will outline how to describe all the required VPCs and how to automate their build using Ansible.

**Getting Ready**

The AWS control machine must be connected to the Internet and the AWS credentials to connect to the AWS API for this account must be in place as outlined in the previous recipe.

**How to do it..**

* Create ***us.yml*** and ***eu.yml*** file under the group\_vars directory as shown below

$ cat group\_vars/eu.yml  
aws\_region: eu-west-1

$ cat group\_vars/eu.yml  
aws\_region: us-east-1

* Create the eu\_prod\_vpc.yml file and us\_prod\_vpc.yml file under host\_vars directory and populate them as shown below.

$ cat host\_vars/eu\_prod\_vpc.yml  
  
vpc\_name: EU\_Prod\_Public\_VPC

vpc\_cidr: 10.3.0.0/16  
vpc\_tags:

role: prod

region: eu

$ cat host\_vars/us\_prod\_vpc.yml  
vpc\_name: US\_Prod\_Public\_VPC

vpc\_cidr: 10.1.0.0/16  
vpc\_tags:

role: prod

region: US

* Create a new playbook pb\_aws\_net\_build.yml and populate it as shown below

$ cat pb\_aws\_net\_build.yml  
  
- name: Create all AWS Networks

hosts: prod\_vpcs

environment:

AWS\_ACCESS\_KEY: "{{ aws\_access\_key }}"

AWS\_SECRET\_KEY: "{{ aws\_secret\_key\_id }}"

tasks:

- name: Create New VPC

ec2\_vpc\_net:

cidr\_block: "{{ vpc\_cidr }}"

region: "{{ aws\_region }}"

name: "{{ vpc\_name }}"

state: "{{ vpc\_state | default('present') }}"

tags: "{{ vpc\_tags }}"

register: create\_vpc

notify: Create VPC  
  
 handlers:

- name: Create VPC

debug: var=create\_vpc

**How it works..**

AWS has global presence across the globe and it segregate each part of its infrastructure in each part of the world into regions. An AWS Region is the collection of AWS facilities in a part of the world and each region in AWS is considered to be an isolated fault domain with its own Orchestration and management systems. So when we are creating a VPC we need to specify which region we will create this VPC so we need to describe this information in our ansible variables. In our case we specify the aws region for all our VPCs in US to be us-east-1 and all our VPCs in Europe (EU) to be in eu-west-1. This is achieved by defining the ***aws\_region*** variable under the eu.yml and us.yml files under the group\_vars directory.

This logic of AWS region is critical for most of the services in AWS which are region specific and all the networking constructs that we will built are all region specific. Declaring the aws\_region with Ansible is a must since when we specify the aws\_region in all the modules in order to initiate the correct API call to the correct API endpoint in the designated region since the API endpoint for each region has a different FQDN. For more information regarding the AWS endpoints for all the services in AWS in all the regions please check the below link.  
<https://docs.aws.amazon.com/general/latest/gr/rande.html>

We declare the variables for each VPC under the host\_vars directory and we create a yml file for each of our VPCs. We specify the VPC name, VPC prefix and the tags that should be assigned to the VPC. Finally, we create the ansible playbook to build our infrastructure and we use a new option within the playbook which is the **environment**. This option create a temporarily environment variables ( AWS\_ACCESS\_KEY and AWS\_SECRET\_KEY) during playbook execution. These environment variables have their value set equal to the aws\_access\_key and aws\_secret\_key\_id variables that we have defined in the all.yml file. This make these Environment variables present during the playbook execution.

We create the VPCs on AWS cloud using the ansible module ec2\_vpc\_net and we specify the AWS region where this VPC will be deployed using the region attribute. We define its IP prefix, name and any associated tags. All this information is derived from the variables that we have defined in the host\_vars file for this VPC.

As the module create the VPC it returns the all the information for the VPC that was created and we save this information in a new variable called create\_vpc and we assigne a handeler for this task , so as when this task changes the VPC state we run the task “Create VPC” in the handelers section to output what have changed using the debug module.

Below is a snippet of the VPC creation when we run this task for the

"create\_vpc": {

"vpc": {

"cidr\_block": "10.1.0.0/16",

< -- Output Omitted for brevity -->

"dhcp\_options\_id": "dopt-b983c8c2",

"id": "vpc-0d179be0eb66847f3",

"instance\_tenancy": "default",

"is\_default": false,

"owner\_id": "955645556619",

"state": "available",

"tags": {

"Name": "US\_Prod\_Public\_VPC",

"region": "US",

"role": "prod"

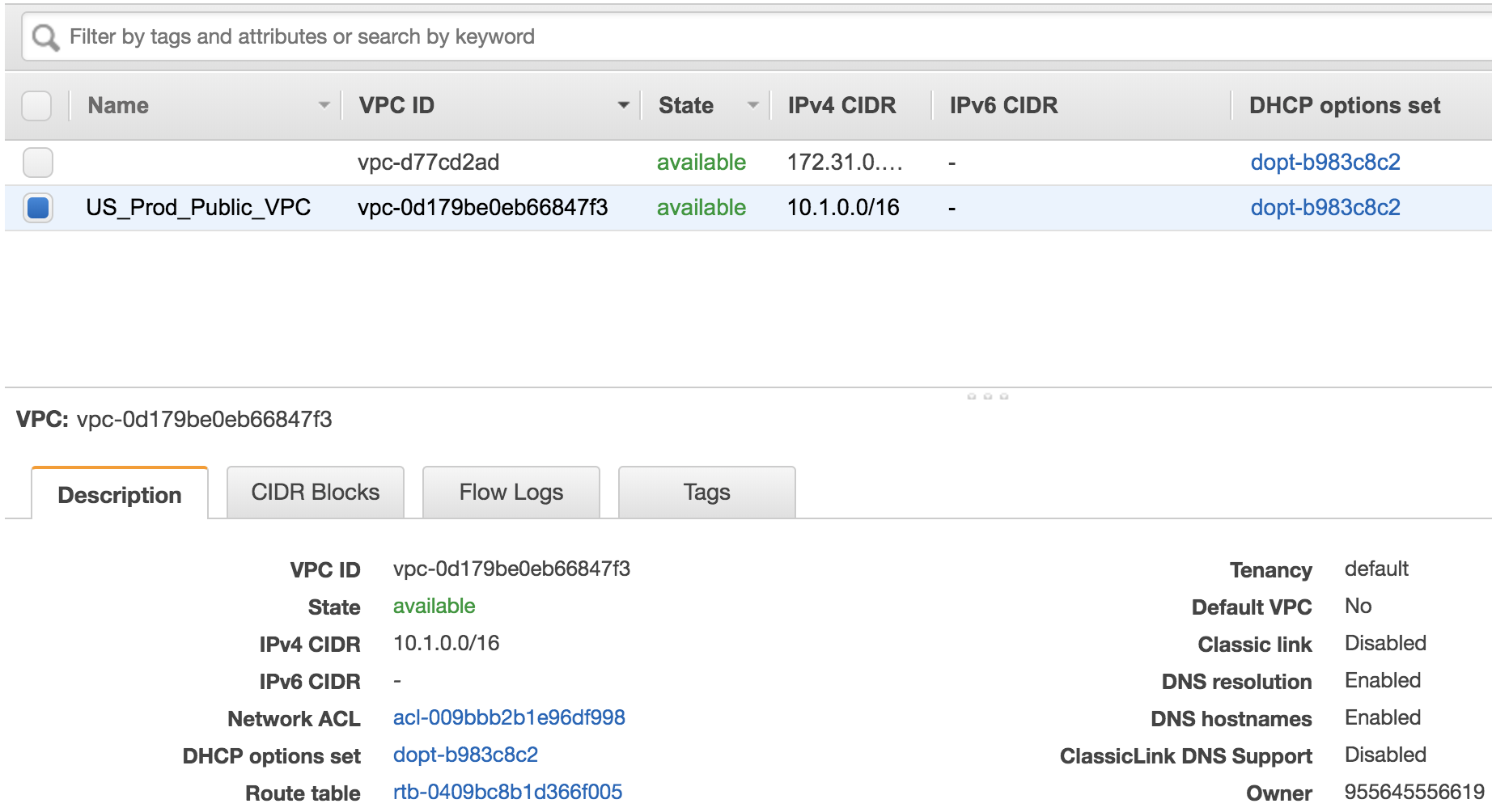
}

}

}

}

The below diagram outlines the VPC created on AWS from the console



**See Also..**

For more information regarding the ec2\_vpc\_net module and the other parameters available within this module please check the below URL.

<https://docs.ansible.com/ansible/latest/modules/ec2_vpc_net_module.html#ec2-vpc-net-module>

**Deploying Subnets using Ansible**

In this recipe, we will outline how to deploy Subnets within our AWS VPCs using Ansible. Subnets is a fundamental networking construct within AWS in order to provide more resiliency for application deployed on AWS Cloud. Subnets provide the arability to be mapped to different availability zones within an AWS region to provide a more redundant setup for the network infrastructure.

**Getting Ready**

The Ansible Control machine must have internet reachability and the VPCs must be already provisioned as per the previous recipe.

**How to do it..**

* Update the eu\_prod\_vpc.yml file with the subnets data as shown below. Same is done for us\_prod\_vpc.yml to include all the subnets data.

$ cat host\_vars/eu\_prod\_vpc.yml  
  
< -- Output Omitted for brevity -->  
  
vpc\_subnets:

eu-prod-public-a:

cidr: 10.3.1.0/24

az: "{{ aws\_region }}a"

tags: "{{ vpc\_tags }}"

public: true

eu-prod-public-b:

cidr: 10.3.2.0/24

az: "{{ aws\_region}}b"

tags: "{{ vpc\_tags }}"

public: true

* Update the playbook pb\_aws\_net\_build.yml and populate it with the new task to build the subnets

$ cat pb\_aws\_net\_build.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: "set fact: VPC ID"

set\_fact:

vpc\_id: "{{ create\_vpc.vpc.id }}"

tags: [vpc,subnet,igw,rt]

- name: create VPC subnets

ec2\_vpc\_subnet:

region: "{{ aws\_region }}"

vpc\_id: "{{ vpc\_id }}"

cidr: "{{ item.value.cidr }}"

az: "{{ item.value.az }}"

tags: "{{item.value.tags | combine({ 'Name': item.key })}}"

with\_dict: "{{ vpc\_subnets }}"

register: create\_vpc\_subnets

**How it works..**

Within an AWS region the availability zones are the construct which provides resiliency for the physical infrastructure within an AWS region. In order to use the availability zones effecitently we need to map our infrastructure within a VPC to be allocated to different availability zones within the region and this is accombilies using AWS Subnets.

In our sample deployment we use two subnets spread across two availability zones in order to provide high avaialbity to our network setup. We declare the subnets that we will deploy within each VPC using the vpc\_subnets variable. This variables include the CIDR that we will use within each subnet (which must be a subset of the VPC CIDR) along with the Availability Zone that we want this Subnet to be attached to and finally the Tags that we want to assign to this subnet. We build the Availability Zone name using the AWS region plus the suffix (‘a’,’b’,’c’,etc..). This is the naming convenstion that AWS uses to name the availability zones within a region.

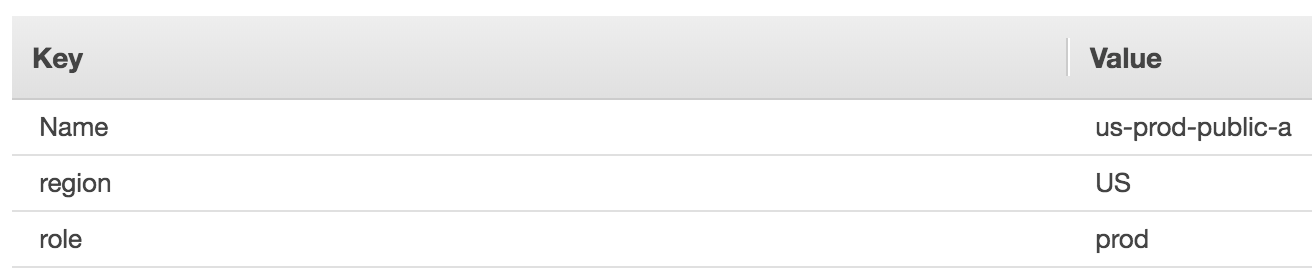
In order to create the subnets, we need to specify these subnets, belong to which VPC. In order to do thin, we need to specify the vpc-id parameter during the API call to create the subnet. This vpc-id is a unique identifier that AWS assign to the VPC during the VPC creation time. We get this value from the VPC creation task that was executed first to create the VPC and we saved the output of this task to the variable vpc\_create. We this this variable to retrive the ID of the VPC and assign it to the vpc-id variable using set\_fact module.

Finally we build the subnets using the ec2\_vpc\_subnet module to create the needed subnets within each VPC and we loop over the vpc\_subnets data structure in order to build all the required subnets.

Below snippet outline the Subnets are correctly provisioned on the AWS cloud in our US\_Prod VPC



And below are the tags assigned to this Subnet



**See Also..**

For more information regarding the ec2\_vpc\_subnet module and the other parameters available within this module please check the below URL.

<https://docs.ansible.com/ansible/latest/modules/ec2_vpc_subnet_module.html#ec2-vpc-subnet-module>

**Deploying IGWs using Ansible**

In this recipe we will outline how to deploy Internet Gateways to our AWS VPCs using Ansible. Internet Gateways are our exit point from our VPC to the Internet in order to reach public external destination. Since we are building a public facing Service we need to have internet reachability from our VPC and this is accomplished by the IGW construct in AWS cloud.

**Getting Ready**

The Ansible Control machine must have internet reachability and the VPCs must be already provisioned as per the previous recipe.

**How to do it..**

* Update the eu\_prod\_vpc.yml file with the IGW data as shown below and same for us\_prod\_vpc.yml.

$ cat host\_vars/eu\_prod\_vpc.yml  
  
< -- Output Omitted for brevity -->  
  
igw\_name: eu\_prod\_igw  
  
  
$ cat host\_vars/eu\_prod\_vpc.yml  
  
< -- Output Omitted for brevity -->  
  
igw\_name: us\_prod\_igw

* Update the playbook pb\_aws\_net\_build.yml and populate it with the new task to build the IGW nodes

$ cat pb\_aws\_net\_build.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Create IGW

ec2\_vpc\_igw:

region: "{{ aws\_region }}"

vpc\_id: "{{ vpc\_id }}"

state: present

tags: "{{ vpc\_tags | combine({'Name': igw\_name}) }}"

register: vpc\_igw\_create

tags:

- igw  
  
 - name: Extract VPC IGW ID

set\_fact:

igw\_id: "{{ vpc\_igw\_create.gateway\_id }}"

**How it works..**

The IGW network construct is our exit point from our VPC to reach public destinations across the internet. The IGW is attached to the VPC and it provides the internet connectivity to any resource located within the VPC (like EC2 or RDS instances). In order to create the IGW we need to specify the VPC that this IGW will be attached to, thus we need the vpc-id for the VPC.

As we discussed in the previous recipe, we get the vpc-id when we create the VPC and we save this variable using a separate task. We use the value of this variable during IGW creation. We use the ec2\_vpc\_igw module to create the IGW and we specify the region that we want this IGW deployed into and we specify the vpc-id that the IGW will be attached to. Finally, we specify the tags that we will allocate to the IGW node. IGW tags are optional however they are critical when using automated deployment since they allow us to reference the objects that we have created. We will outline the use of Tags when we discuss deployment validation and fact collection in the next recipes.

When we deploy a new IGW the ec2\_vpc\_igw module return the IGW parameters that was provisioned inside AWS, one particular parameter is important which is the igw-id. This parameter uniquily identify the IGW node that was provisioned and we must use it when we reference the IGW in any opertation related to this IGW node.

Below is a snippet of the IGW parameters returned by the ec2\_vpc\_igw which we captured in the vpc\_igw\_create variable for the IGW node in us\_prod\_vpc

ok: [us\_prod\_vpc] => {

"vpc\_igw\_create": {

"changed": true,

"failed": false,

"gateway\_id": "igw-05d3e4c664486790b",

"tags": {

"Name": "us\_prod\_igw",

"region": "US",

"role": "prod"

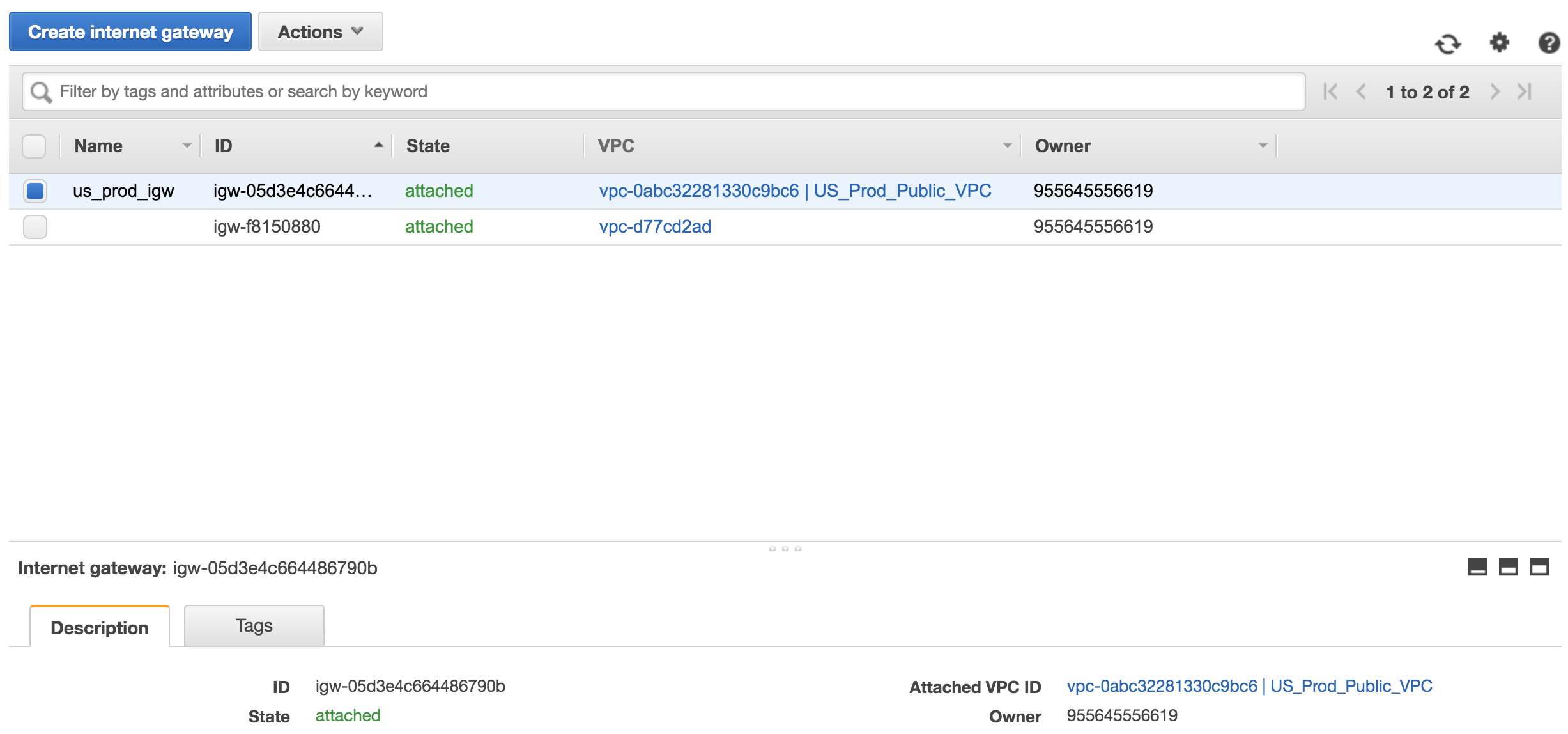
},

"vpc\_id": "vpc-0abc32281330c9bc6"

}

}

In the last task we capture the gateway-id returned by the ec2\_vpc\_igw and store it in a new variable called igw\_id which we will use it in subsequent tasks when referencing the IGW node.

The Below diagram outline the IGW node that was provisioned and attached to the VPC  
  


**See Also..**

For more information regarding the ec2\_igw\_vpc module and the other parameters available within this module please check the below URL.

<https://docs.ansible.com/ansible/latest/modules/ec2_vpc_igw_module.html#ec2-vpc-igw-module>

**Controlling Routing within VPC using Ansible**

In this recipe, we will outline how to adjust the routing inside the AWS VPC in order to control the traffic forwarding within the subnets inside the VPC. Controlling the Routing within the VPC allow us to customize the VPC design and how the traffic is forwarded within the VPC as well as how to exit the VPC to external destinations.

**Getting Ready**

The Ansible Control machine must have internet reachability and the VPCs must be already provisioned as per the previous recipe.

**How to do it..**

* Update the eu\_prod\_vpc.yml file with the Routing table data as shown below and same for us\_prod\_vpc.yml.

$ cat host\_vars/eu\_prod\_vpc.yml  
  
< -- Output Omitted for brevity -->  
  
route\_table:

tags:

Name: eu\_public\_rt

igw:

- dest: 0.0.0.0/0

gateway\_id: "{{ igw\_id }}"

public:

- eu-prod-public-a

- eu-prod-public-b

* Update the playbook pb\_aws\_net\_build.yml and populate it with the below tasks to get the route table attached to the VPC that we have created

$ cat pb\_aws\_net\_build.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Get Default VPC Route Table

ec2\_vpc\_route\_table\_facts:

region: "{{ aws\_region }}"

filters:

vpc-id: "{{ vpc\_id }}"

register: vpc\_route\_table\_facts

tags: rt

- name: Extract Route Table IDs

set\_fact:

rt\_id: "{{vpc\_route\_table\_facts.route\_tables[0].id }}"

tags: rt

* Update the playbook pb\_aws\_net\_build.yml and populate it with the below tasks to get update the route table with the required routes.

$ cat pb\_aws\_net\_build.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Update Default VPC Route Table

ec2\_vpc\_route\_table :

region: "{{ aws\_region }}"

vpc\_id: "{{ vpc\_id }}"

route\_table\_id: "{{ rt\_id }}"

routes: "{{ route\_table.igw }}"

subnets: "{{ route\_table.public }}"

lookup: id

state: present

tags: "{{ vpc\_tags | combine(route\_table.tags) }}"

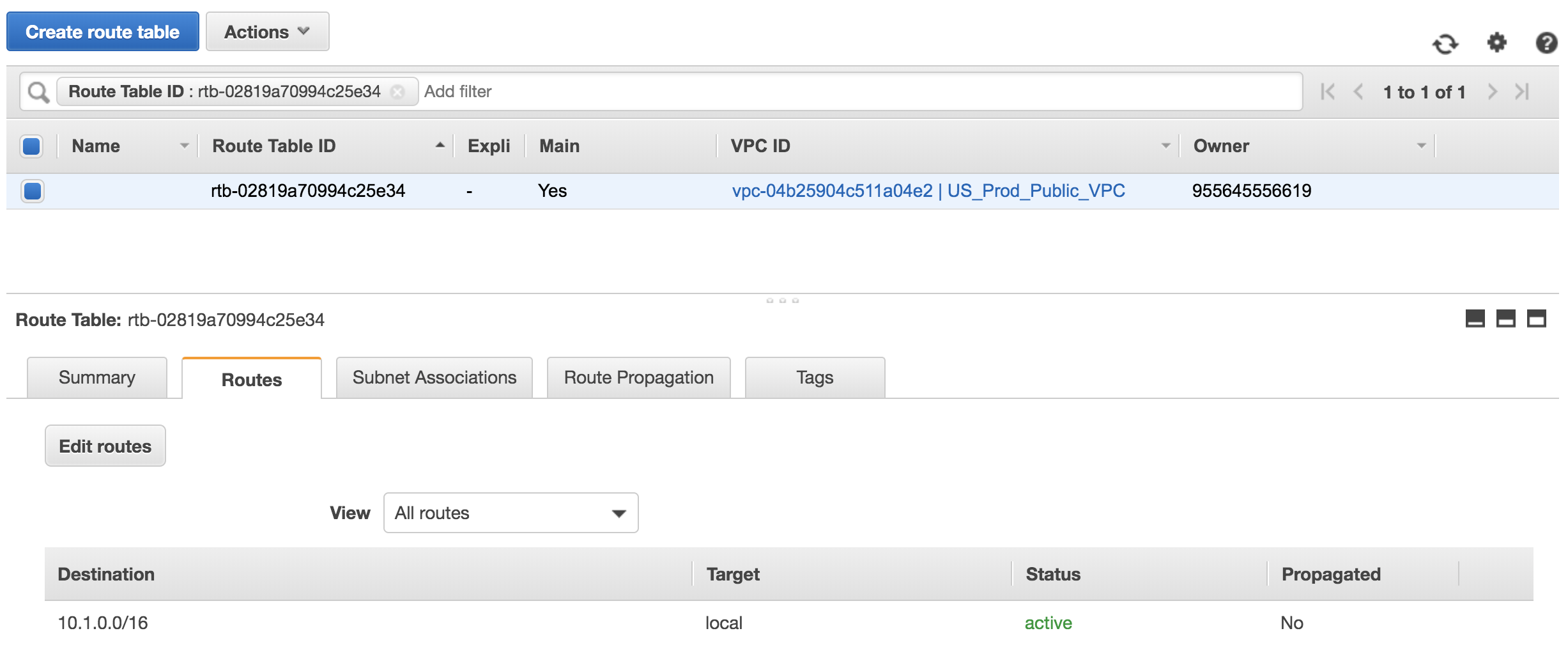
tags:

- rt

**How it works..**

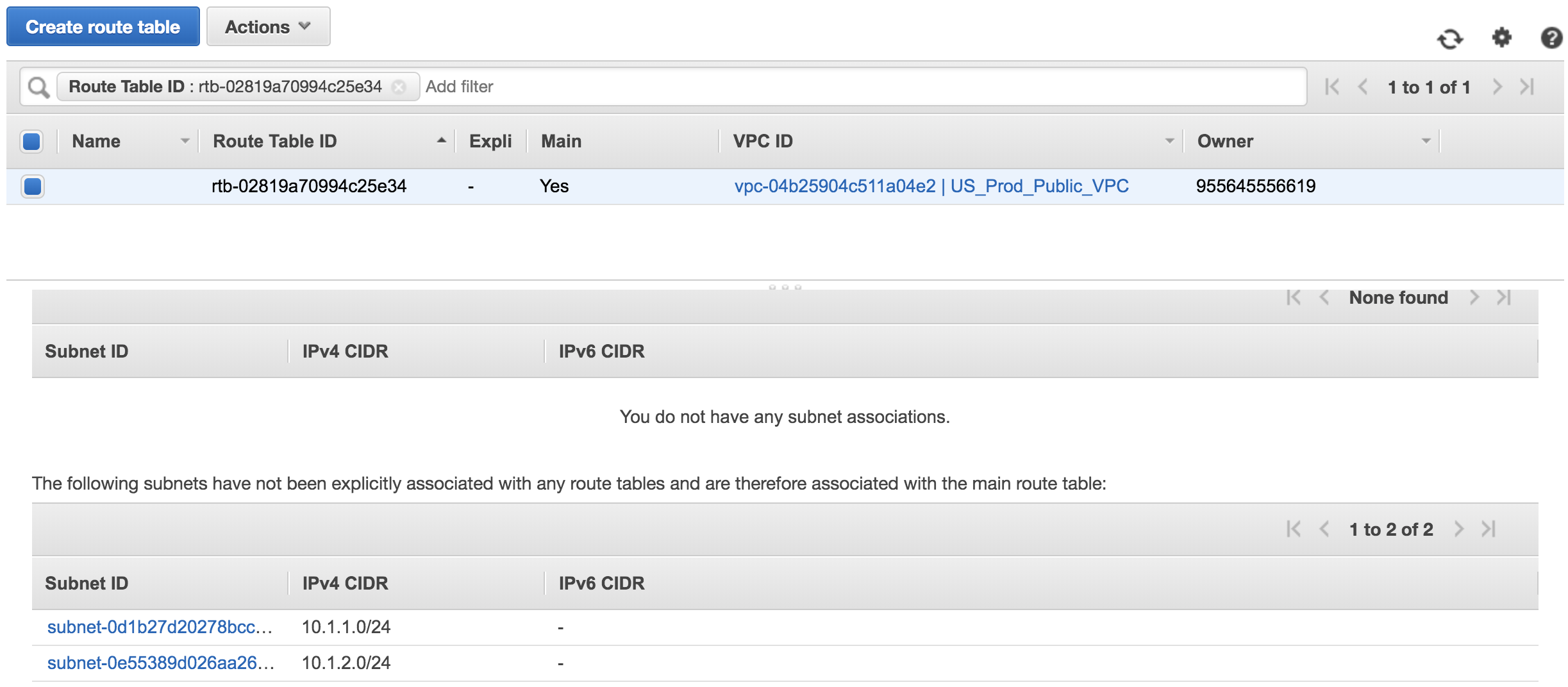
Up until that point this point, we have managed to setup the VPC, the Subnets and the IGW. However, although the IGW node is connected to the Internet and it is attached to the VPC, no traffic within the VPC will use the IGW node since the routing table associated with the VPC is still not updated and there is no route to point IGW.

Below is a snippet of the Default routing table for the us\_prod\_vpc before changing the Route table



**Note**  
AWS VPCs have a default route-table which is assigned to the VPC and to all the subnets which doesn’t have a specific route-table associated with them. So by default all the Subnets within the VPC are associated with the VPC default route-table.

Below is a diagram outline that the Subnets created within the us\_prod\_vpc is associated with the default route-table



In the VPC metadata that we have declared for each of our VPCs, we include a new data structure called route\_table which include all the information we need to adjust the routing table for our VPC and associate all the Subnets within this route table.

The First task that we execute in this recipe is to get the ID for the default route-table that is associated with the VPC that we have created. We use the ec2\_vpc\_route\_table\_facts module to get the facts for the route-table and we supply the vpc-id to uniquely identify the VPC. We store the ID for the default route-table in the new variable rt\_id.

The below snippet outline the route-table facts that we retrieve from the ec2\_vpnc\_facts module

ok: [us\_prod\_vpc] => {

"vpc\_route\_table\_facts": {

"route\_tables": [

{

< -- Output Omitted for brevity --> ],

"id": "rtb-0b6669ba5fd9eb9c8",

"routes": [

{

"destination\_cidr\_block": "10.1.0.0/16",

"gateway\_id": "local",

< -- Output Omitted for brevity -->  
  
 }

],

"tags": {},

"vpc\_id": "vpc-005b1dcb981791d86"

}

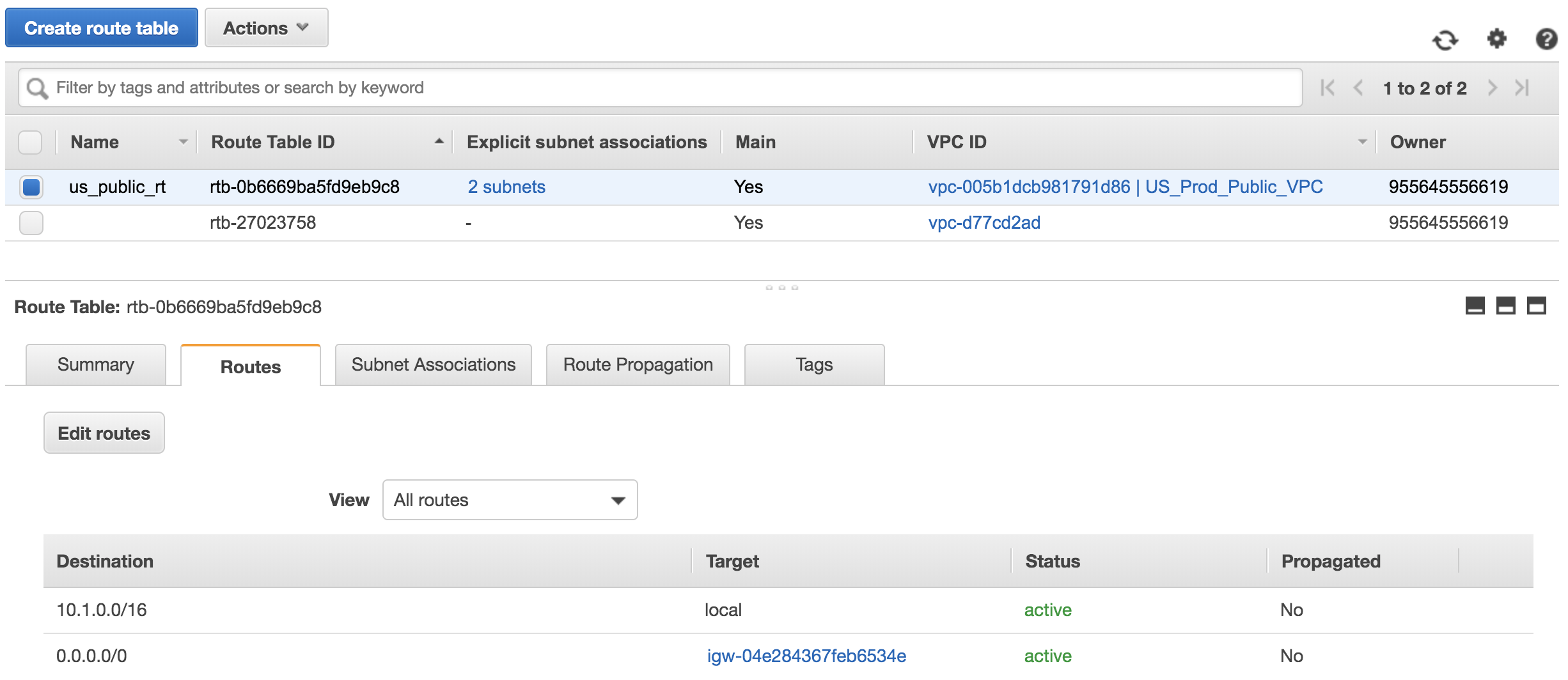
]

}

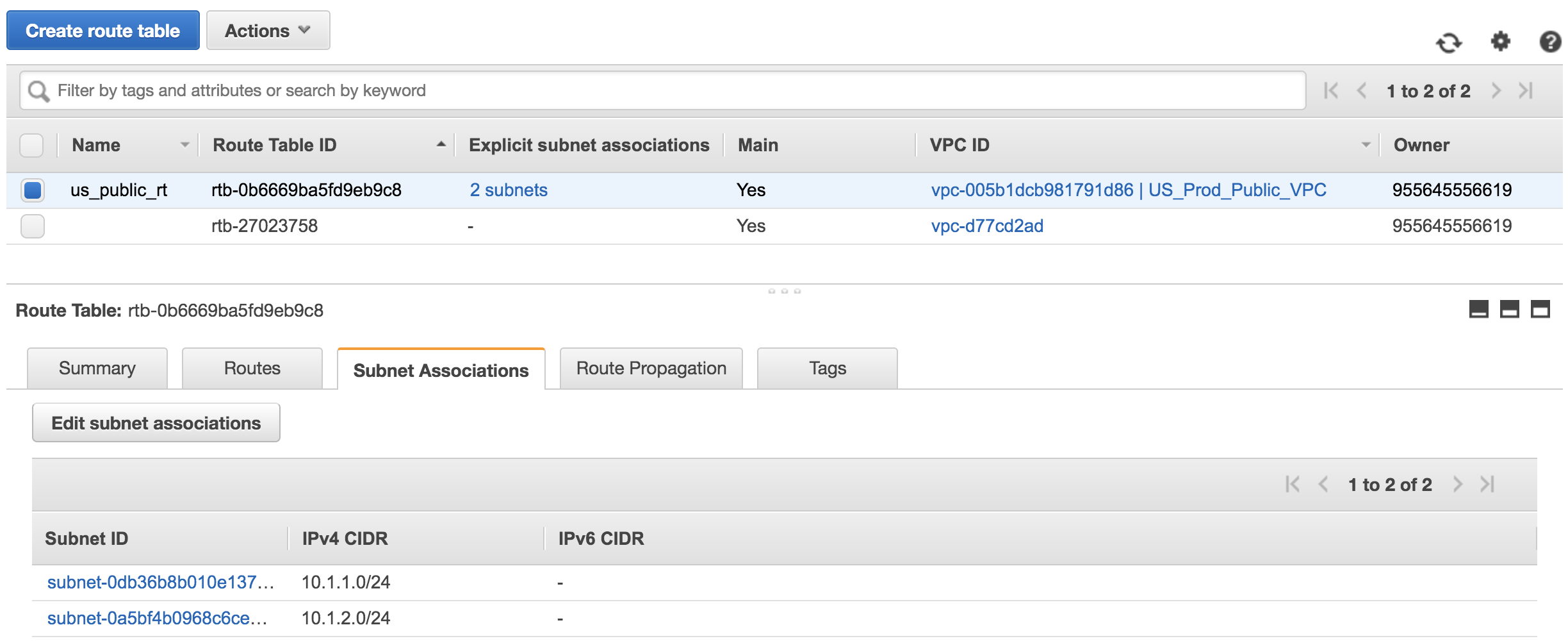
}

Once we get the ID of the route-table associated with the VPC, we use the ec2\_vpc\_route\_table module to adjust the routing table for the default route-table associated with the VPC. We must supply the vpc-id and route-table id to uniquely identify these components. We specify routes that we want to inject in the routing table and the subnets that we want to associate with this route table. We inject the default route and point this default route towards the IGW that we have created in the previous recipe, using the igw-id.

The Below diagram outline the routing table for our VPC after adjusting the routing.



The below diagram outline how the two subnets that we have in the VPC are now associated with this default route table



**See Also..**

For more information regarding the multiple modules to interact with the routing-table of the AWS VPC please check the below links for the associated modules.

<https://docs.ansible.com/ansible/latest/modules/ec2_vpc_route_table_module.html#ec2-vpc-route-table-module>

<https://docs.ansible.com/ansible/latest/modules/ec2_vpc_route_table_facts_module.html#ec2-vpc-route-table-facts-module>

**Deployment Validation using Ansible**

In this recipe we will outline how we can validate the operational state of the network by defining the intended state of the network and letting NAPALM validate that the actual/operational state of the network matches our intended state. This is useful in network auditing and compliance reports for our network infrastructure.

**Getting Ready**

To follow along with this recipe, an Ansible inventory is assumed to be already in place and Network reachability between the Ansible controller and the Network is established. Finally, The Network is configured as per the previously outline recipe.

**How to do it..**

* Create a new folder called napalm\_validate and create a YAML file for each device we want to validate its state as shown below.

$ cat napalm\_validate/mxpe01.yml  
  
---  
- get\_interfaces\_ip:

ge-0/0/0.0:

ipv4:

10.1.1.3:

prefix\_length: 31

- get\_bgp\_neighbors:

global:

router\_id: 10.100.1.1

* Create a new playbook pb\_napalm\_validation.yml with the below contents

$ cat pb\_napalm\_validation.yml  
  
---

- name: " Validating Network State via NAPALM"

hosts: junos:&pe

tasks:  
 - name: "P1T1: Validation with NAPALM"

napalm\_validate:

hostname: "{{ ansible\_host }}"

username: "{{ ansible\_user }}"

password: "{{ ansible\_ssh\_pass }}"

dev\_os: "{{ ansible\_network\_os }}"

validation\_file: "napalm\_validate/{{ inventory\_hostname}}.yml"

ignore\_errors: true

register: net\_validate

* Update the playbook to create folder to store compliance reports for each device as shown below

$ cat pb\_napalm\_validation.yml  
  
< -- Output Omitted for brevity -->  
  
 - name: Create Compliance Report Folder

file: path=compliance\_folder state=directory

- name: Create Compliance Report File per Node

file: path=compliance\_folder/{{inventory\_hostname}}.txt state=absent

- name: Create Compliance Report

copy:

content: "{{ net\_validate.compliance\_report | to\_nice\_yaml }}"

dest: "compliance\_folder/{{ inventory\_hostname }}.txt"

**How it works..**

NAPALM provides another module for network validation which is the napalm\_validate module. This module is mainly used to perform auditing and compliance reports for the network infrastructure and the main goal is to provide an intended state for the network defined and declared in a YAML document formatted in a specific format following the same structure that the different NAPALM getters uses. In this YAML file we specify the NAPALM facts that we want to retrieve from the network and the expected output that we expect the network state to be like.

We supply these validation files to napalm\_validate module and NAPALM will connect to the devices, retrieve the facts specified in these validation files and compare the output retrieved from the network against the network stated declared in these validation files. Finally, it generates a compliance\_report object which has the result of the comparison and whether the network comply with these validation files or not. We also set the **ignore\_errors** parameter in order to continue with the other tasks in this playbook in case the device doesn’t comply, so we can capture this compliance problem in the compliance report that we will generate.

Finally, we save these output in a separate folder called compliance\_folder for each node and we copy the contents of the compliance\_report parameter and we formate it using to\_nice\_yaml filter.

A snippet for a correct compliance report generated for mxpe01 device is shown below

complies: true

get\_bgp\_neighbors:

complies: true

extra: []

missing: []

present:

global:

complies: true

nested: true

get\_interfaces\_ip:

complies: true

extra: []

missing: []

present:

ge-0/0/0.0:

complies: true

nested: true

skipped: []

**See Also..**

For Further information regarding NAPALM validation and the other options available for napalm\_validation please check the below URLs

<https://napalm.readthedocs.io/en/latest/integrations/ansible/modules/napalm_validate/index.html>  
<https://napalm.readthedocs.io/en/latest/validate/index.html>

**Decommission Resources on AWS Using Ansible**

In this recipe, we outline how to decommission a complete Network within AWS with all the associated network resources. This outline how easily we can build and tear down resources on the cloud with a simple playbook execution using ansible.

**Getting Ready**

The Ansible Control machine must have internet reachability , and all the networking components that we have outlined in the previous recipes should be in place.

**How to do it..**

* Create a new playbook pb\_delete\_vpc.yml and include the first task to collect the Facts of the established resources within the VPC.

$ cat pb\_delete\_vpc.yml

- name: Delete all VPC resources

hosts: all

gather\_facts: no

environment:

AWS\_ACCESS\_KEY: "{{ aws\_access\_key }}"

AWS\_SECRET\_KEY: "{{ aws\_secret\_key\_id }}"

AWS\_REGION: "{{ aws\_region }}"

tasks:

- name: Collect VPC Facts

include\_tasks: tasks/vpc\_facts.yml

* Update the playbook with the below tasks to remove all the Subnets and IGW nodes within the VPC

- name: Start Delete VPC Resources

block:

- name: Delete Subnets

ec2\_vpc\_subnet:

cidr: "{{ item.value.cidr }}"

vpc\_id: "{{ vpc\_id }}"

state: absent

with\_dict: "{{ vpc\_subnets }}"

- name: Delete IGW

ec2\_vpc\_igw:

vpc\_id: "{{ vpc\_id }}"

state: absent

* Update the playbook with the final task to remove all the VPCs.

- name: Delete VPC

ec2\_vpc\_net:

cidr\_block: "{{ vpc\_cidr }}"

name: "{{ vpc\_name }}"

state: absent

when: vpc\_id is defined

**How it works..**

We start our new playbook with the collection of facts from the different resources within our VPC that are provisioned. We include all the tasks for fact collection in a single task which is vpc\_fact.yml file. This task retrieves the ID for the VPCs, Subnets and IGW nodes within our AWS account.

Once we have all this information we start to delete the resources, however the order by which to start to delete the resources is important. We need to remove any dependent resource first, so we must remove the subnets before we can remove the VPC. So, for example if there was EC2 instances attached to the subnet, we must remove these EC2 instances before we can remove the subnets and so on. So in our case, we remove the subnets, then the IGW node and finally we remove the VPC.

In all the tasks we are using the same exact modules that we have outlined in the previous recipes, the only change is that we are setting the state to be absent and we supply the required vpc-id to uniquely identify the VPC that we need to remove the required resources from it.

Finally, we are using a conditional when we start to remove the resources within the VPC since in case if the resources are already deleted and we run the playbook again, the deletion step is skipped since no vpc-id is retrieved by the facts task.