Building Out a Cloud Network

Now that we have launched servers in Microsoft Azure, we will start looking at launching services within **Amazon Web Services** (**AWS**).

Before we launch virtual machine instances, we must create a network to host them. This is called a **virtual private cloud** (**VPC**) and there are a few different elements we will need to bring together in a playbook to create one, which we will then be able to use for our instances.

In this chapter, we will do the following:

* Receive an introduction to AWS
* Cover what it is we are trying to achieve and why
* Create a VPC, subnets, and routes (networking and routing)
* Create security groups (firewall)

We will look at more advanced Ansible techniques as we launch and manage more dynamic resources with complex dependencies.

The chapter covers the following topics:

* An introduction to AWS
* Amazon VPC overview
* Creating an access key and secret
* Getting Ansible ready for targeting AWS
* The AWS playbook
* Running the playbook

Technical requirements

This chapter will use AWS; you will need administrator access to create the roles needed to allow Ansible to interact with your account. As with other chapters, you can find the complete playbooks in the Chapter10 folder in the accompanying GitHub repository at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter10.

An introduction to AWS

AWS has been around since 2002; it started by offering a few services that were not linked in any way. It progressed in this form until early 2006 when it was relaunched. The relaunched AWS brought together three services:

* **Amazon Elastic Compute Cloud** (**Amazon EC2**): This is the AWS compute service
* **Amazon Simple Storage Service** (**Amazon S3**): Amazon’s scalable object storage service
* **Amazon Simple Queue Service** (**Amazon SQS**): This service provides a message queue, primarily for web applications

Since 2006, it has grown from three unique services to over 160, covering over 15 primary areas such as the following:

* Compute
* Storage
* Database
* Networking and content delivery
* Machine learning analytics security, identity, and compliance
* Internet of things

At its earnings call in October 2023, it was revealed that AWS had USD 23.06 billion in revenue in the third quarter of 2023, good for a service that initially offered to share idle compute time.

At the time of writing, AWS spans 32 geographic regions, which host a total of 102 availability zones (https://aws.amazon.com/about-aws/global-infrastructure/).

So, what makes AWS so successful? Not only its coverage but its approach to putting out its services. Andy Jassy, AWS CEO, has been quoted as saying:

“*Our mission is to enable any developer or any company to be able to build all their technology applications on top of our infrastructure technology platform*.”

As an individual, you have access to the same APIs, services, regions, tools, and pricing models as large multi-national companies and Amazon themselves, as they consume their services. This gives you the freedom to start small and scale massively. For example, Amazon EC2 instances start from around USD 4.50 per month for a t2.nano (1 vCPU and 0.5G) all the way up to over USD 19,000 per month for an x1e.32xlarge (128 vCPU, 3,904 GB RAM, and two 1920 GB SSD storage); as you can see, there are instance types for every workload imaginable.

Both instances and most services are billed under pay-as-you-go, from per-second billing for EC2 instances to pay per GB per month for the storage you are using.

Amazon VPC overview

In this chapter, we are going to be concentrating on launching an **Amazon Virtual Private Cloud** (**Amazon VPC**); this is the networking layer that will host the computing and other Amazon services that we will be launching in [*Chapter 11*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/11), *Highly Available Cloud Deployments*.

We are going to be launching our VPC into the **EU-West #1 (Ireland)** region; we will be spanning all three availability zones for our **EC2** instances and also the **Application Elastic Load Balancer**. We will, again, be using the three availability zones for our **Amazon Relational Database Service** (**RDS**) instance and also two zones for the **Amazon Elastic File System** (**Amazon EFS**) volumes.

This all means our Ansible playbook needs to create/configure the following:

* One Amazon VPC
* Three subnets for EC2 instances
* Three subnets for Amazon RDS instances
* Three subnets for Amazon EFS volumes
* Three subnets for the Application Load Balancer
* One internet gateway

We will also need to configure the following:

* One route to allow access through the internet gateway
* One security group that allows everyone to access port 80 (HTTP) and 443 (HTTPS) on the Application Load Balancer
* One security group that allows trusted source access to port 22 (SSH) on the EC2 instances
* One security group that allows access to port 80 (HTTP) from the Application Load Balancer to the EC2 instances
* One security group that allows access to port 3306 (MySQL) on the Amazon RDS instances from the EC2 instances
* One security group that allows access to port 2049 (NFS) on the Amazon EFS volumes from the EC2 instances

This will give us our primary network, allowing restrictive access to everything but the Application Load Balancer, which we want to be publicly available.

Before creating an Ansible playbook that deploys the network, we need to get an AWS API access key and secret.

Creating an access key and secret

It is more than possible to create an access key and secret key for your AWS user to give Ansible full access to your AWS account.

Because of this, we are going to look at creating a user for Ansible, which only has permission to access the parts of AWS we know that Ansible will need to interact with for the tasks we are covering in this chapter. We will be giving Ansible full access to the following services:

* Amazon VPC
* Amazon EC2
* Amazon RDS
* Amazon EFS

To do this, follow these steps:

1. Log in to the AWS console, which can be found at https://console.aws.amazon.com/.
2. Once logged in, click on **Services**, which can be found in the menu at the top of the screen. In the open menu, enter IAM into the search box and then click on the **IAM “Manage access to AWS resources”** result.
3. On the **IAM** page, click **User Groups** in the left-hand side menu; we will create a group with the permissions assigned to it, and then we will create a user and add it to our group.
4. Once on the **User Groups** page, click the **Create Group** button. This process has two steps, the first of which is setting the group’s name. In the space provided, enter the group name Ansible.
5. Now, in the **Attach permissions policies – Optional** section, select **AmazonEC2FullAccess**, **AmazonVPCFullAccess**, **AmazonRDSFullAccess**, and **AmazonElasticFileSystemFullAccess**; once all four have been selected, click on the **Create Group** button at the bottom of the page.
6. Now that we have our Ansible group, click **Users** in the left-hand side menu.
7. Once on the **Users** page, click **Create user**, and this will take you to a page where you can configure your desired username and the type of user you want. Enter the following information:
   * **User name**: Enter LearnAnsible in here
   * Leave the **Provide user access to the AWS Management Console – optional** option unchecked, as we will create a programmatic user
8. Click on the **Next** button to take you to the **Set Permissions** page. Ensure that **Add user to group** is selected and that you have the Ansible group we created earlier ticked, and then click **Next**, which will take you to the **Review and Create** page.
9. Once you have reviewed the details, you need to click the **Create user** button, which will precisely do that: create our LearnAnsible user.
10. The final step is to get an access key for our user. To get this, click on the LearnAnsible user and select the **Security credentials** tab; from there, scroll down to **Access Keys** and click the **Create access key** button.
11. In the list of **Access key best practices & alternatives** select **Other** and then the **Next** button. Enter For use with Learn Ansible for the description tag value and then click **Create access key**.
12. The **Retrieve access keys** page is the only time you get access to the Secret access key, so I recommend downloading the CSV file. Once downloaded, click on **Done**.

Important note

The CSV file you have just downloaded contains credentials allowing whoever has them to launch resources in your AWS account; please do not share them and keep them safe, as they could be misused, resulting in a huge and unexpected AWS bill should they fall into the wrong hands.

Now that we have an access key ID and secret access key for a user with the permissions, we need to launch our VPC using Ansible; we can start getting Ansible ready and reviewing the playbook.

Getting Ansible ready for targeting AWS

We first need to discuss how to pass our access key ID and secret access key to Ansible safely and securely. As I will share the final playbooks in a public repository on GitHub, I want to keep my AWS keys private from the world as that could get expensive! Typically, if it were a private repository, I would use Ansible Vault or some other secret management to encrypt the keys and include them with other potentially sensitive data, such as deployment keys.

In this case, I don’t want to include any encrypted information in the repository, as it would mean that people would need to unencrypt it, edit the values, and then re-encrypt it. Luckily, the AWS modules allow you to set two environment variables on your Ansible controller; those variables will then be read as part of the playbook execution.

To set the variables, run the following commands to make sure that you replace the content with your access key and secret after = (the information listed as follows is just placeholder values):

$ export AWS\_ACCESS\_KEY=AKIAI5KECPOTNTTVM3EDA

$ export AWS\_SECRET\_KEY=Y4B7FFiSWl0Am3VIFc07lgnc/TAtK5+RpxzIGTr

CopyExplain

Once set, you can view the contents by running the following:

$ echo $AWS\_ACCESS\_KEY

CopyExplain

Now that we can securely pass our credentials to Ansible, we can install the Python modules needed by the AWS Ansible modules to interact with the AWS API.

Important note

You must set the environment variables for each terminal session, as they will be lost each time you close your terminal.

To install the Python modules, run the following command:

$ pip3 install botocore boto3

CopyExplain

Now that we have the basics configured, we can review our playbook.

The AWS playbook

As mentioned at the start of the chapter, we are going to be using some more advanced techniques when it comes to deploying resources in AWS where possible; I have tried to allow the resources to be deployed as dynamically as possible, a lot of which comes down to how we define our variables, which is where we are going to start our playbook review.

The playbook variables

Most of the variables we define can be found in group\_vars/common.yml, and as you can see from the following, they start by looking a lot like the variables we described in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*:

debug\_output: false

app:

  name: "learnansible"

  region: "eu-west-1"

  env: "prod"

CopyExplain

As you can see, we have the same debug\_output feature flag and selection of variables used to describe our app and the AWS region in which it will be launched.

Next up, we have the resource names:

vpc\_name: "{{ app.name }}-{{ app.env }}-{{ playbook\_dict.vpc }}"

internet\_gateway\_name: "{{ app.name }}-{{ app.env }}-{{ playbook\_dict.internet\_gateway }}"

internet\_gateway\_route\_name: "{{ internet\_gateway\_name }}-{{ playbook\_dict.route }}"

CopyExplain

Nothing too out of the ordinary so far, but here we will find our first difference in approach:

vpc:

  cidr\_block: "10.0.0.0/23"

  dns\_hostnames: true

  dns\_support: true

  subnet\_size: "27"

  subnets:

    - name: "ec2"

      role: "{{ subnet\_role\_compute }}"

    - name: "rds"

      role: "{{ subnet\_role\_database }}"

    - name: "efs"

      role: "{{ subnet\_role\_storage }}"

    - name: "dmz"

      role: "{{ subnet\_role\_public }}"

CopyExplain

At first glance, that doesn’t look too dissimilar to what we did for Microsoft Azure.

However, you might have noticed that there are no IP address CIDR ranges listed for the subnets, just some details about the subnets, including a dictionary of roles:

subnet\_role\_compute: "compute"

subnet\_role\_database: "database"

subnet\_role\_storage: "storage"

subnet\_role\_public: "public"

CopyExplain

We will look at why the subnet’s CIDR ranges are missing when we get to the tasks that create the subnet.

Next, we have the variables for creating the security groups; in total, we will be configuring four security groups, so in the interest of space, I will only be showing one of the small groups here:

security\_groups:

  - name: "{{ app.name }}-rds-{{ playbook\_dict.security\_group }}"

    description: "opens port 3306 to the ec2 instances"

    id\_var\_name: "rds\_group\_id"

    rules:

      - proto: "tcp"

        from\_port: "3306"

        to\_port: "3306"

        group\_id: "{{ ec2\_group\_id | default('') }}"

        rule\_desc: "allow {{ ec2\_group\_id | default('') }} access to port 3306"

CopyExplain

See the GitHub repo for the full configuration for the four security groups; there is only one thing at this point to highlight, and that is this: where we reference {{ ec2\_group\_id | default('') }}, we are setting a default value of nothing (which is the '' part). We will discuss why we are doing this when we cover the security role.

The final set of variables is the dictionary (playbook\_dict) and a variable, which sets the value of region using app.region; again, see the GitHub if you want to see all the contents.

The VPC role

Before we get to the exciting tasks, we need to create the VPC. The task in roles/vpc/tasks/main.yml looks like the following:

- name: "Create VPC"

  amazon.aws.ec2\_vpc\_net:

    name: "{{ vpc\_name }}"

    region: "{{ region }}"

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

    dns\_hostnames: "{{ vpc.dns\_hostnames }}"

    dns\_support: "{{ vpc.dns\_support }}"

    state: "{{ state }}"

    tags:

      Name: "{{ vpc\_name }}"

      projectName: "{{ app.name }}"

      environment: "{{ app.env }}"

      deployedBy: "{{ playbook\_dict.deployedBy }}"

      description: "{{ playbook\_dict.ansible\_warning }}"

  register: vpc\_output

CopyExplain

The task is pretty much as you would expect, apart from the tags being set a little more in line than those we defined in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*. There is also a debug statement that prints the results of creating the VPC if you set debug\_output to true:

- name: "Debug - VPC result"

  ansible.builtin.debug:

    var: "vpc\_output"

  when: debug\_output

CopyExplain

From now on, it is safe to assume that all registered output will be followed by an ansible.builtin.debug task. Now that we have our VPC launched, we can start putting things inside it, beginning with the subnets, where things get more interesting.

The subnets role

As mentioned in the AWS overview, there are 32 geographic regions and, at the time of writing, 102 Availability Zones. AWS differs from Microsoft Azure in that you need a subnet per Availability Zone rather than a single subnet spanning all the availability zones.

The eu-west-1 region, which is the region we will target, is made up of three availability zones, and we have subnets for four different roles, meaning that we need 12 subnets in total, but our playbook could easily be targeting a region that only has two availability zones, or in some cases, even more.

So, our first task is to get information on the availability zones in our target region:

- name: "Get some information on the available zones"

  amazon.aws.aws\_az\_info:

    region: "{{ region }}"

  register: zones\_output

CopyExplain

Now that we know some information on the region, we can use that information and create our subnets:

- name: "Create all subnets"

  ansible.builtin.include\_tasks: create\_subnet.yml

  loop: "{{ vpc.subnets }}"

  loop\_control:

    loop\_var: subnet\_item

    index\_var: subnet\_index

  vars:

    subnet\_name: "{{ subnet\_item.name }}"

    subnet\_role: "{{ subnet\_item.role }}"

    az\_zones\_from\_main: "{{ zones\_output }}"

  register: subnet\_output

CopyExplain

This task is quite different from the ones we have been using so far in the book, so let’s take a deeper dive into what is happening.

Here, we are using a loop to automate the creation of multiple subnets. Each iteration of the loop processes one subnet from the vpc.subnets list, which, as we have already seen, contains the configuration details for each subnet.

As the loop runs, it assigns the current subnet’s details to the subnet\_item variable and its index in the list to subnet\_index. These variables are then utilized to customize the creation process for each subnet.

The task includes and executes the steps defined in create\_subnet.yml (which we will cover next) for each subnet, using the specific details of that subnet (such as its name and role).

You may have noticed that we still haven’t passed in any CIDR ranges for the subnets; this is all handled within the create\_subnet.yml task, which we loop over for each of our four subnet types; this is also where a second loop happens:

- name: "Create subnet in the availability zone"

  amazon.aws.ec2\_vpc\_subnet:

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_id: "{{ vpc\_output.vpc.id }}"

    cidr: "{{ vpc\_output.vpc.cidr\_block | ansible.utils.ipsubnet(vpc.subnet\_size, az\_loop\_index + (subnet\_index \* az\_zones\_from\_main.availability\_zones|length)) }}"

    az: "{{ az\_item.zone\_name }}"

    tags:

      Name: "{{ subnet\_name }}-{{ playbook\_dict.subnet }}-{{ az\_item.zone\_id }}"

      projectName: "{{ app.name }}"

      environment: "{{ app.env }}"

      deployedBy: "{{ playbook\_dict.deployedBy }}"

      description: "{{ playbook\_dict.ansible\_warning }}"

      role: "{{ subnet\_role }}"

  loop: "{{ az\_zones\_from\_main.availability\_zones }}"

  loop\_control:

    loop\_var: az\_item

    index\_var: az\_loop\_index

CopyExplain

Please stick with me, as this is where it gets a little confusing; for each of the four loops we are enacting from our main loop, we are taking the information on the availability zones and then looping over them, creating a subnet per availability zone for the role we are currently looping over.

So, what about the CIDR range for the subnet?

You may have noticed something where you would expect to see the CIDR range; we have this expression:

vpc\_output.vpc.cidr\_block | ansible.utils.ipsubnet(vpc.subnet\_size, az\_loop\_index + (subnet\_index \* az\_zones\_from\_main.availability\_zones|length))

CopyExplain

We have the following components in the expression:

* vpc\_output.vpc.cidr\_block: This is the CIDR block of the VPC, within which the subnets will be created. For our example, it’s 10.0.0.0/22.
* vpc.subnet\_size: This specifies the size of each subnet. We are using /27, representing a subnet with 32 IP addresses.
* az\_zones\_from\_main.availability\_zones|length: This is the total number of availability zones available. The region we are targeting has 3 availability zones.
* az\_loop\_index: This is the current index in the loop over the availability zones.
* subnet\_index: This is the index of the current subnet being processed.

This means that for our expression, we will get the following results. The first subnet, which is labeled **ec2**, in the availability zone (az1) will have the following:

* az\_loop\_index = 0
* subnet\_index = 0

So, the formula would be 0+(0\*3)=0, meaning that we would get the following:

cidr = "{{ vpc\_output.vpc.cidr\_block  | ansible.utils.ipsubnet(27, 0) }}"

CopyExplain

With vpc\_output.vpc.cidr\_block being 10.0.0.0/22, we could get the first /27, which would be 10.0.0.0/27.

For the second availability zone (az2), the loop would be the following:

* az\_loop\_index = 1
* subnet\_index = 0

1+(0\*3)=1 means we would get 10.0.0.32/27 since the next subnet block starts immediately after the previous one at the next 32 IP address interval.

The third Availability Zone (az3) would be 2+(0\*3)=2, and the CIDR block would be 10.0.0.64/27.

The next subnet role, which is the RDS role, would give the following for az1:

* az\_loop\_index = 0
* subnet\_index = 1

The formula would be 0+(1\*3)=3, giving us a CIDR block 10.0.0.96/27.

This pattern would follow the sequence, where the next subnet for RDS az2 would be at 10.0.0.128/27, and for az3, it would be at 10.0.0.160/27, and so on.

This expression ensures that each subnet created within the VPC is assigned a unique and non-overlapping CIDR block, segmented adequately according to the defined subnet size, and distributed across different availability zones.

Taking this approach not only simplifies the management of subnet creation but also ensures efficiency when it comes to writing the role, as it means that we don’t have to hardcode tasks to consider changes between regions or the number of subnets we are defining in our variables.

The remaining tasks in the role build a list of the subnet IDs for each of the roles we have defined. An example of one of these tasks is as follows:

- name: "Gather information about the compute subnets"

  amazon.aws.ec2\_vpc\_subnet\_info:

    region: "{{ region }}"

    filters:

      "tag:role": "{{ subnet\_role\_compute }}"

      "tag:environment": "{{ app.env }}"

      "tag:projectName": "{{ app.name }}"

  register: subnets\_compute\_output

CopyExplain

This gets information on the three subnets assigned the subnet\_role\_compute role. A few more of these data-gathering tasks can be found in the repo; these cover the subnet\_role\_database, subnet\_role\_storage, and subnet\_role\_public roles.

Finally, the final task in the role prints the subnet IDs that we have gathered using the previous set of tasks; this looks slightly different to the debug statements we have been using in the playbook so far, as we are using the msg function rather than the var one when calling the ansible.builtin.debug module.

The gateway role

The gateway role is relatively simple compared to the previous one. In comparison, it deploys an internet gateway. Then, it creates a route to send all traffic destined for the internet (represented by using 0.0.0.0/0, the CIDR notation for all network traffic) to our newly launched internet gateway.

The task that creates the internet gateway looks like the following:

- name: "Create an Internet Gateway"

  amazon.aws.ec2\_vpc\_igw:

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_id: "{{ vpc\_output.vpc.id }}"

    tags:

      "Name": "{{ internet\_gateway\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

      "deployedBy": "{{ playbook\_dict.deployedBy }}"

      "description": "{{ playbook\_dict.ansible\_warning }}"

      "role": "igw"

  register: internet\_gateway\_output

CopyExplain

As per the rest of the tasks, a debug task follows this, and then the task that creates the route table, which is then associated with our newly created internet gateway and also the computing and public subnets that we defined and gathered the information for in the subnet’s role:

- name: "Create a route table so the internet gateway can be used by the public subnets"

  amazon.aws.ec2\_vpc\_route\_table:

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_id: "{{ vpc\_output.vpc.id }}"

    subnets: "{{ subnet\_compute\_ids + subnet\_public\_ids }}"

    routes:

      - dest: "0.0.0.0/0"

        gateway\_id: "{{ internet\_gateway\_output.gateway\_id }}"

    resource\_tags:

      "Name": "{{ internet\_gateway\_route\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

      "deployedBy": "{{ playbook\_dict.deployedBy }}"

      "description": "{{ playbook\_dict.ansible\_warning }}"

      "role": "route"

  register: internet\_gateway\_route\_output

CopyExplain

We then do a debug task that completes this role, and we then move on to the final role of the playbook: the security group’s role.

The security group’s role

While this role, in my opinion, is not as complicated as the subnet’s role, we have built a little more logic into the task than some of the more straightforward tasks in the book that we have run so far.

If you recall, earlier in the chapter, when we covered the variables being used by the playbook, we gave the following example of the security groups being deployed:

  - proto: "tcp"

    from\_port: "3306"

    to\_port: "3306"

    group\_id: "{{ ec2\_group\_id | default('') }}"

    rule\_desc: "allow {{ ec2\_group\_id | default('') }} access to port 3306"

CopyExplain

The preceding rule, as per rule\_desc, opens up port 3306 for any devices that have the EC2 security group attached to them, which, as we will see in [*Chapter 11*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/11), *Highly Available Cloud Deployments*, will be the EC2 instances that will be running our workload.

You may think to yourself, *“Now that makes sense.”* However, this is a little bit of a flaw in the logic we must work around. ec2\_group\_id is referencing a group ID, which, at the time we first run our playbook, doesn’t exist. So, how can we create the groups and populate them with rules that reference groups that don’t yet exist?

As we have already seen, looping over the resources defined in our variables is more efficient. It reduces the hard-coded logic at the role level, making the role more re-useable between projects and playbooks.

Before we look at the logic of creating the groups, we need to gather one bit of information: the public IP address of the resource running Ansible. To do this, we call the following task:

- name: "Find out your current public IP address using https://ipify.org/"

  community.general.ipify\_facts:

  register: public\_ip\_output

CopyExplain

Then we set a fact called your\_public\_ip, which we can reference in our rules where needed:

- name: "Set your public ip as a fact"

  ansible.builtin.set\_fact:

    your\_public\_ip: "{{ public\_ip\_output.ansible\_facts.ipify\_public\_ip }}/32"

CopyExplain

Now that we have that snippet of information, we can return to the question of how we can reference the IDs of resources that have yet to be launched.

To create the security groups, we will be using the amazon.aws.ec2\_security\_group module. The module has a flag called purge\_rules, set to true by default; in this default state, when our playbook finds and needs to update an existing security group, it will drop all the rules in the group and then add just the ones defined in the playbook to maintain a consistent state.

While it is a valid use case, in our example, disabling this functionality by setting purge\_rules to false will allow us to create some unpopulated security groups:

- name: "Create the base security groups"

  amazon.aws.ec2\_security\_group:

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_id: "{{ vpc\_output.vpc.id }}"

    name: "{{ item.name }}"

    description: "{{ item.description }}"

    purge\_rules: false

    tags:

      "Name": "{{ item.name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

      "deployedBy": "{{ playbook\_dict.deployedBy }}"

      "role": "securitygroup"

  loop: "{{ security\_groups }}"

  register: base\_security\_groups\_output

CopyExplain

This will loop through and create the base, unpopulated security groups if they don’t exist, and if they do already exist, no changes will be made to them.

So, now that we have our groups created, or if they already exist, we have the information we need to dynamically define some facts based on the output of the previous tasks:

- name: "Set the fact for the security group ids"

  ansible.builtin.set\_fact:

    "{{ item.id\_var\_name }}": "{{ base\_security\_groups\_output.results | selectattr('item.name', 'equalto', item.name) | map(attribute='group\_id') | first }}"

  loop: "{{ security\_groups }}"

  when: base\_security\_groups\_output.results | selectattr('item.name', 'equalto', item.name) | map(attribute='group\_id') | list | length > 0

CopyExplain

This task uses the ansible.builtin.set\_fact module, allowing the creation or update of new variables during runtime. This task aims to extract the unique ID of each security group created in the first task and assign it to a specific variable name.

There are two expressions we use to do this. The first is the following:

"{{ item.id\_var\_name }}": "{{ base\_security\_groups\_output.results | selectattr('item.name', 'equalto', item.name) | map(attribute='group\_id') | first }}"

CopyExplain

This is used to create the dynamic set of variables based on the loop created by the second expression. A breakdown of this first expression follows:

* base\_security\_groups\_output.results: This refers to the list of results from the previous task that created the security groups. Each result in this list contains data about one of the security groups.
* selectattr('item.name', 'equalto', item.name): The selectattr filter is used to search through the list of results. It looks for results where the name attribute of the item (each security group) is equal to the current item.name in the loop. In other words, it filters the results to find the specific security group we’re currently interested in.
* map(attribute='group\_id'): The map filter is then used to transform the filtered list of results. It extracts only the group\_id attribute from each result, which is the ID of the security group.
* first: Since the previous step can still return a list (albeit with a single element), the first filter takes only the first element from this list, which should be the unique ID of the security group.

The result of this expression is the ID of the security group that matches the current item in the loop, and it’s assigned to a variable named according to item.id\_var\_name.

The second expression, which is in the when condition, runs as part of the loop:

when: base\_security\_groups\_output.results | selectattr('item.name', 'equalto', item.name) | map(attribute='group\_id') | list | length > 0

CopyExplain

This expression determines whether the task should be executed for a particular item in the loop. It follows a similar logic to the first expression:

* It starts with the same filtering process to find the security group that matches the current item.name.
* After extracting the group\_id, it ensures the output is treated as a list using the list filter.
* length > 0: This part checks whether the length of the list (the number of items in it) is greater than 0. This means at least one security group with the specified name must exist. If the list is empty, no matching security group is found, and the task will be skipped for the current item.

In theory, we should have now populated the variables that contain the security group IDs, meaning that we can now add the rules:

- name: "Provision security group rules"

  amazon.aws.ec2\_security\_group:

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_id: "{{ vpc\_output.vpc.id }}"

    name: "{{ item.name }}"

    description: "{{ item.description }}"

    purge\_rules: false

    rules: "{{ item.rules }}"

  loop: "{{ security\_groups }}"

  register: security\_groups\_with\_rules\_output

CopyExplain

This will loop over the already created groups and populate the rules for each one, using the group IDs from the variables we dynamically defined in the previous task.

Running the playbook

As mentioned earlier, we worked our way through the playbook code; before you run the playbook, you must set the AWS\_ACCESS\_KEY and AWS\_SECRET\_KEY environment variables on your terminal session by running the following, making sure to update any values to those that you made a note of when you created the Ansible user in the AWS console:

$ export AWS\_ACCESS\_KEY=AKIAI5KECPOTNTTVM3EDA

$ export AWS\_SECRET\_KEY=Y4B7FFiSWl0Am3VIFc07lgnc/TAtK5+RpxzIGTr

CopyExplain

With the environment variables set, you can run the playbook running the now very familiar following code:

$ ansible-playbook -i hosts site.yml

CopyExplain

Once completed, you should see something like the following terminal output:

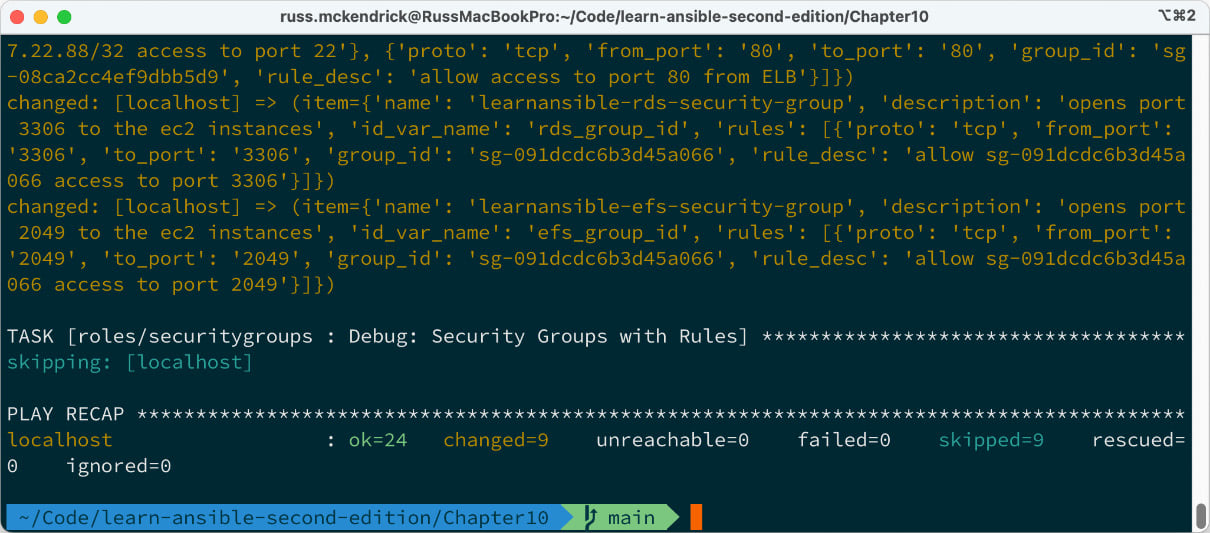


Figure 10.1 – Running the playbook in a terminal

Going to the VPC and viewing the resource map in http://console.aws.amazon.com/ should display something like the following resource map:

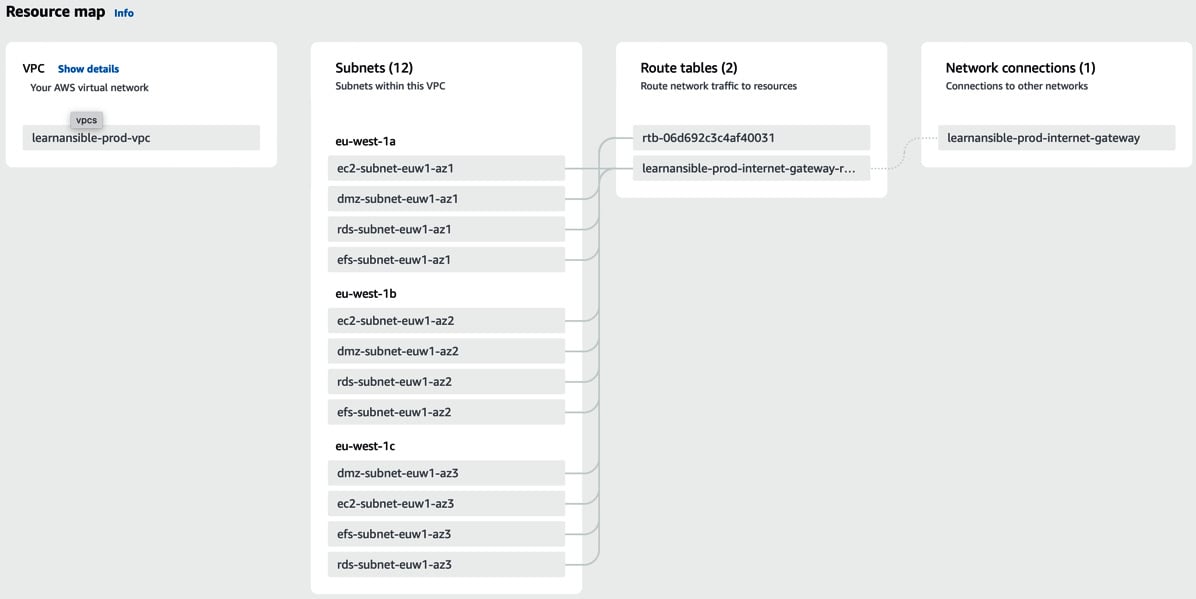


Figure 10.2 – Viewing the resource map

By going to **Security Groups**, you should also see the groups that we created listed:

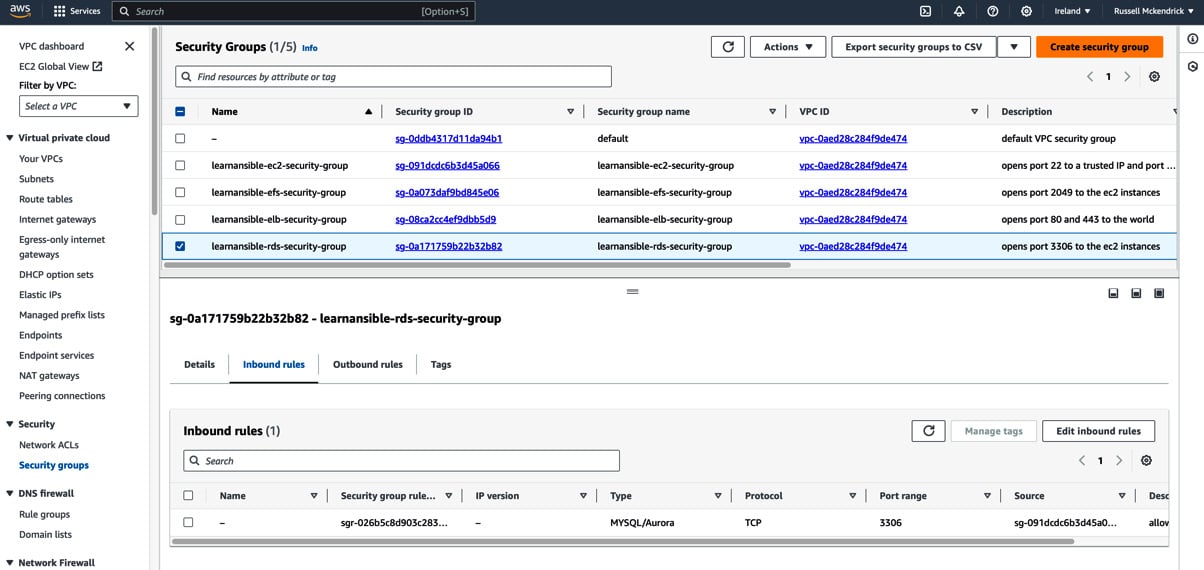


Figure 10.3 – Reviewing the security groups

I have included a second playbook in the repo, which destroys all of the resources created by running the site.yml playbook called destroy.yml. You can run it using the following command:

$ ansible-playbook -i hosts destroy.yml

CopyExplain

I am not going to cover the contents of the playbook here, but if you review the code, you will notice that, in essence, it runs the same tasks in the role we have covered in this chapter in reverse order, setting the state to absent rather than present.

Summary

In this chapter, we have taken our next step in using Ansible to launch resources in a public cloud. We have laid the groundwork for automating quite a complex environment by creating a VPC, setting up the subnets we need for our application, provisioning an internet gateway, and setting our instances to route their outgoing traffic through it.

We have configured four security groups, with three containing dynamic content, to secure the services launching into our VPC.

In the next chapter, we will build on the foundations laid in this chapter and launch a more complex set of services alongside the VPC.