Highly Available Cloud Deployments

Continuing with our AWS deployment, we will start to deploy services into the network we created in the previous chapter, and by the end of the chapter, we will be left with a highly available WordPress installation.

Building on top of the roles we created in the previous chapter, we will be doing the following:

* Launching and configuring an Application Load Balancer
* Launching and configuring Amazon **Relational Database Service** (**RDS**) (database)
* Launching and configuring Amazon **Elastic File System** (**EFS**) (shared storage)
* Launching an **Elastic Compute Cloud** (**EC2**) instance and creating an **Amazon Machine Image** (**AMI**) from it (deploying the WordPress code)
* Launching and configuring a launch template to use the newly created AMI and autoscaling group (high availability)

The chapter covers the following topics:

* Planning the deployment
* The Playbook
* Running the Playbook
* Terminating all the resources

Technical requirements

As in the previous chapter, we will be using AWS; you will need the access key and secret key we created in the previous chapter to launch the resources needed for our highly available WordPress installation. Please note that we will be launching resources that incur charges. Again, you can find the complete playbook in the Chapter11 folder of the accompanying GitHub repository at https://github.com/PacktPublishing/Learn-Ansible-Second-Edition/tree/main/Chapter11/.

Planning the deployment

Before diving into the playbooks, we should get an idea of what we are trying to achieve. As mentioned, we will build on our AWS **Virtual Private Cloud** (**VPC**) role by adding instances and storage; our final deployment will look like the following diagram:

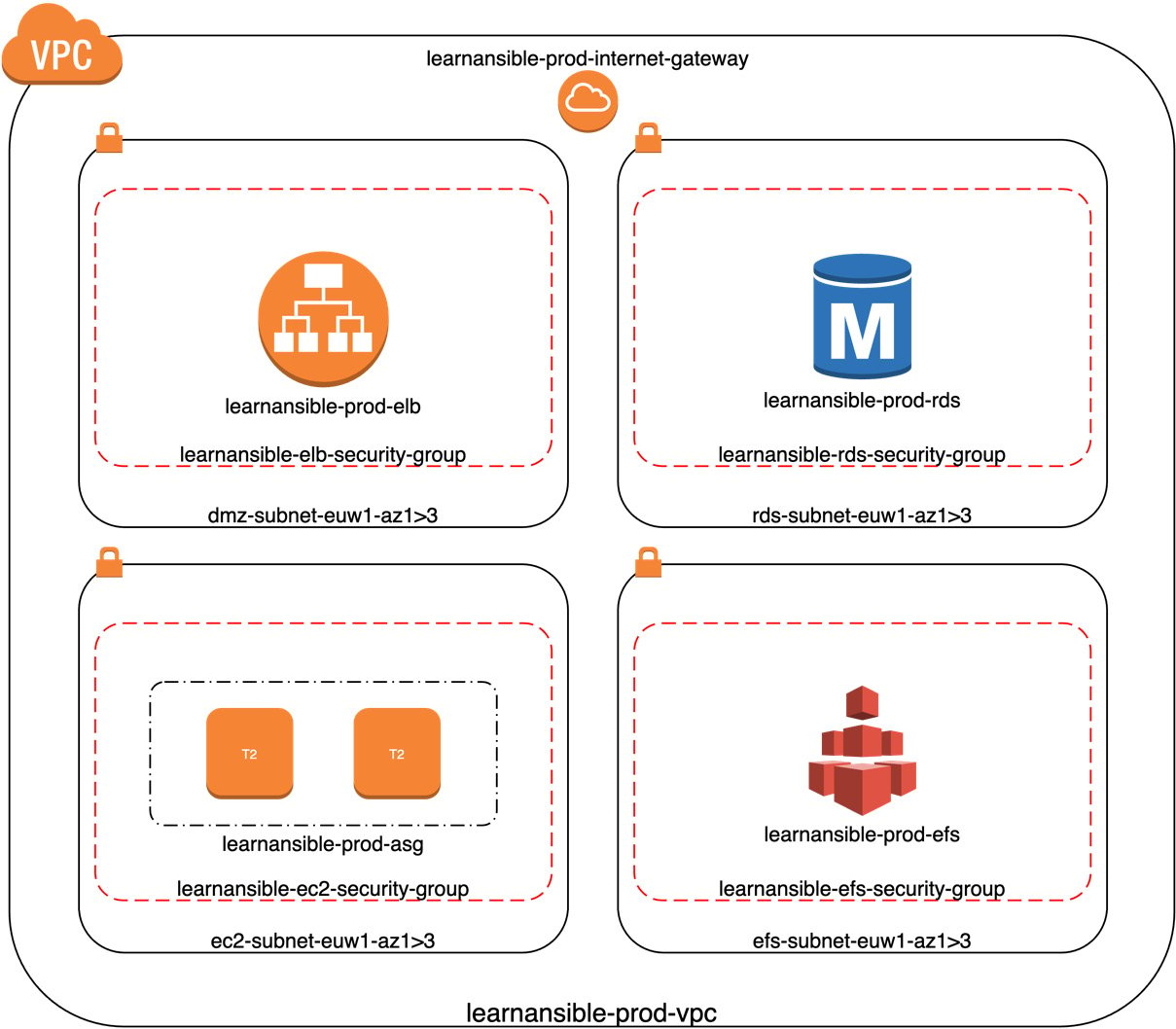


Figure 11.1 – An overview of what we shall be launching

In the diagram, we have the following:

* 2 x EC2 instances (t2.micro), deployed across different availability zones
* 1 x RDS instances (t2.micro)
* 1 x EFS storage across three availability zones

Before we talk about the deployment itself, based on the diagram and specifications here, how much is this deployment going to cost us to run?

Costing the deployment

The cost of running this deployment in the EU-West-1 region is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Instance Type** | **# Number** | **Instance cost** | **Total Monthly Cost** |
| EC2 instances (t2.micro) | x2 | $9.20 | $18.40 |
| RDS instance (t2.micro) | x1 | $13.14 | $13.14 |
| Application Load Balancer | x1 | $24.24 | $24.24 |
| EFS | 5GB | $0.88 | $4.40 |
| **Total** |  |  | $61.83 |

Table 11.1 – Cost of running the deployment

There will be a few other minor costs, such as bandwidth and storing the AMI that contains our software stack. We could also consider increasing these costs by adding additional redundancy, such as updating our RDS instance to a multi-AZ RDS primary and stand-by instance deployment and increasing the number of EC2 instances.

However, this introduces additional complexity to our deployment, as we are about to spend the rest of the chapter covering the playbook, which will be deploying the resources. I want to keep this playbook as simple as possible for now.

WordPress considerations and high availability

So far, we have been launching WordPress on a single server, which is fine. Still, as we are trying to remove as many of the single points of failure within our deployment as possible, we must put a little thought into how we initially configure and launch our deployment.

First, let’s discuss the order we need to launch our deployment. The primary order in which we will need to tackle the elements is as follows:

* **VPC, subnets, internet gateway, routing, and security groups**: These are all needed to launch our deployment.
* **The Application Elastic Load Balancer**: We will be using the public hostname of the Elastic Load Balancer for our installation, so this needs to be launched before we start our installation.
* **The RDS database instance**: Our database instance must be available before we launch our installation, as we need to create the WordPress database and bootstrap the installation.
* **The EFS storage**: We need some storage to share between the EC2 instances we will be launching next.

So far, so good; however, this is where we have to start taking WordPress into account.

As some of you may know from experience, the current version of WordPress is not designed to be spread across multiple servers. We can apply plenty of hacks and workarounds to make WordPress play nicely in this sort of deployment; however, this chapter is about something other than the finer points of deploying WordPress. Instead, it is about using Ansible to deploy a multi-tiered web application.

Because of this, we will be going for the most basic of the multi-instance WordPress options by deploying our code and content on the EFS volume. This means that all we must do is install our LEMP stack. It should be noted that this option could be more performant at a large scale, but it will serve our needs.

Now, back to the list of tasks. When it comes to launching our instances, we need to do the following:

1. Launch a temporary EC2 instance running Ubuntu to reuse parts of existing playbooks.
2. Update the operating system and install the software stack, supporting tools, and configuration needed for us to install and run our WordPress installation.
3. Mount the EFS volume, set the correct permissions, and configure it to mount when the instance boots.
4. Bootstrap WordPress itself.
5. Create an AMI from our temporary instance and then terminate the temporary instance as it will not be needed now.
6. Create a launch template that uses the AMI we just created.
7. Create an autoscaling group and attach the launch configuration; it should also register our WordPress instances with the Elastic Load Balancer.

Further playbook runs, which will update the operating system and non-WordPress configuration, should repeat the process with the existing instances up and running, and then, once the AMI is built, it should be deployed alongside the current instances, which will then be terminated once the new instances are registered with the Elastic Load Balancer and receiving traffic.

This will allow us to update our operating system packages and configurations without downtime if everything goes as planned!

Now that we have an idea of what we are trying to achieve, let’s make a start on our playbook.

The Playbook

We will use the Playbook we looked at in [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, as a starting point, as all the roles are relevant to our deployment, and it already has the structure we need for our playbook.

We will also be using the roles to deploy and configure WordPress and the supporting software stack we used in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*, with a few tweaks, which are needed as we are targeting AWS and not Microsoft Azure; I will let you know when we get to them.

Unlike previous chapters, we will first look at the site.yml file to get an idea of the order in which we will run the roles.

There are three stages in the file, starting with the stage that deploys and configures our underlying AWS resources:

- name: "Deploy and configure the AWS Environment"

  hosts: localhost

  connection: local

  gather\_facts: true

  vars:

    state: "present"

  vars\_files:

    - group\_vars/common.yml

  roles:

    - vpc

    - subnets

    - gateway

    - securitygroups

    - elb

    - efs

    - rds

    - ec2tmp

    - endpoints

CopyExplain

As you can see, this is the same as the site.yml file from [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, with additional roles added to the list from the securitygroups role downwards.

By the time our Playbook run gets to the second stage:

- name: "Install and configure Wordpress"

  hosts: vmgroup

  gather\_facts: true

  become: true

  become\_method: "ansible.builtin.sudo"

  vars\_files:

    - group\_vars/common.yml

    - group\_vars/generated\_aws\_endpoints.yml

  roles:

    - stack\_install

    - stack\_config

    - wordpress

CopyExplain

A file called group\_vars/generated\_aws\_endpoints.yml will have been generated, and there should be a temporary virtual machine instance up and running, meaning SSH should be accessible to the host running the Playbook.

Once this stage has been completed, our temporary virtual machine instance should have our software stack installed. WordPress will be freshly installed if this is the first time the playbook has been run, or if the playbook has detected an existing WordPress installation and left it alone unless there have been any changes to the plugin configuration from within the playbook.

The final stage is then run:

- name: "Create AMI and update the Auto Scaling Group"

  hosts: localhost

  connection: local

  gather\_facts: true

  vars:

    state: "present"

  vars\_files:

    - group\_vars/common.yml

  roles:

    - ec2ami

    - autoscaling

CopyExplain

This stage creates an AMI from the temporary virtual machine instance, terminates the temporary instance as we no longer need it, creates a new version of our launch template, and then creates/updates the Auto Scaling Group to deploy the new version on the EC2 instances.

Sounds simple? Well, let’s find out.

The variables

Out of the box, there is a single variables file called group\_vars/common.yml that contains all the static variables needed to deploy our environment.

Some additional files will be created in the group\_vars folder throughout the Playbook run; they will contain some dynamically generated resources, such as passwords, resource names/endpoints, and other information.

We will discuss these files in more detail when we look at the tasks that create and interact with them; for now, we will look at the static variables defined within group\_vars/common.yml, starting with the base application configuration.

Application and resource configuration

We start the configuration with the option to enable/disable debug when running the Playbook. By default, it is set to false; however, when running the Playbook, I recommend switching it to true and reviewing the output:

debug\_output: false

CopyExplain

Next, we have the application name, region, and environment reference:

app:

  name: "learnansible"

  region: "eu-west-1"

  env: "prod"

CopyExplain

The next block of variables defines details for the WordPress database; as we will be using the Amazon RDS service, we are just using the variables that are defined later in the file, so we only have to update the information in one place:

wp\_database:

  name: "{{ rds.db\_name }}"

  username: "{{ rds.db\_username }}"

  password: "{{ rds.db\_password }}"

CopyExplain

The next block is the various variables used to configure WordPress itself:

wordpress:

  domain: "http://{{ aws\_endpoints.elb }}/"

  title: "WordPress installed by Ansible on {{ os\_family }}"

  username: "ansible"

  password: "{{ rds.db\_password }}"

  email: "test@test.com"

  plugins:

    - "jetpack"

    - "wp-super-cache"

    - "wordpress-seo"

    - "wordfence"

    - "nginx-helper"

CopyExplain

There are no significant changes to when we last defined these in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9)*, Moving to the Cloud*, apart from using the aws\_endpoints.lb variable, which won’t be known until the Elastic Load Balancer has been launched. Also, for ease of use, we are reusing the password, which will be dynamically generated later in the file, as the WordPress admin password.

Stack configuration

The next section overrides the defaults in the roles/stack\_install role:

stack\_packages:

  - "nginx"

  - "mariadb-client"

  - "php-cli"

  - "php-curl"

  - "php-fpm"

  - "php-gd"

  - "php-intl"

  - "php-mbstring"

  - "php-mysql"

  - "php-soap"

  - "php-xml"

  - "php-xmlrpc"

  - "php-zip"

  - "nfs-common" # Added for AWS

  - "nfs4-acl-tools" # Added for AWS

  - "autofs"  # Added for AWS

  - "rpcbind"  # Added for AWS

CopyExplain

We have removed mariadb-server from the list of packages as we no longer need to install or configure a local database server, and we have added four packages at the end (all labeled # Added for AWS). These packages install the software required to mount the EFS filesystem using the NFS protocol, which leads us nicely into the next block:

nfs:

  mount\_point: "/var/www/"

  mount\_options: "nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=600,retrans=2"

  state: "mounted"

  fstype: "nfs4"

CopyExplain

As you can see, this defines some basic information on where the EFS filesystem should be mounted, with what options and the type of filesystem it is.

Resource names

This next section builds up the names of the resources we are going to be deploying; there is nothing too special happening here – it is just defined like this, so we don’t have to update repeated information in several places manually:

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 }}"

elb\_target\_group\_name: "{{ app.name }}-{{ app.env }}-{{ playbook\_dict.elb\_target\_group }}"

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

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

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

ec2\_tmp\_name: "{{ app.name }}-tmp-{{ playbook\_dict.ec2 }}"

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

ec2\_name: "{{ app.name }}-{{ app.env }}-{{ playbook\_dict.ec2 }}"

launch\_template\_name: "{{ app.name }}-{{ app.env }}-{{ playbook\_dict.lt }}"

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

CopyExplain

We will not be covering the full playbook\_dict block here as there is not much to see, although as a reminder, this is what the start of it looks like:

playbook\_dict:

  deployedBy: "Ansible"

  ansible\_warning: "Resource managed by Ansible"

  vpc: "vpc"

CopyExplain

It just continues defining service names. The following section is where we start to define the variables used for the AWS resource deployment.

EC2 configuration

The ec2 variable is split into a few different layers. Layers for the auto-scaling group, the AMI, and the SSH keypair follow some general settings:

ec2:

  instance\_type: "t2.micro"

  public\_ip: true

  ssh\_port: "22"

CopyExplain

The variables are used across instances apart from the public\_ip reference, which is only used when launching the temporary virtual machine instance to bootstrap WordPress.

The next layer defines some details about the auto-scaling group and launch template when used; they help define how many instances are launched, how updated instances are rolled out, and also, how the load balancer will check to see if they are healthy:

  asg:

    min\_size: 1

    max\_size: 3

    desired\_capacity: 2

    health\_check\_type: "EC2"

    replace\_batch\_size: 1

    health\_check\_period: 300

    replace\_all\_instances: true

    wait\_for\_instances: true

    wait\_timeout: 900

    disable\_api\_termination: true

CopyExplain

Next, we define the details about the base AMI we will use; as you can see, we are using Ubuntu 22.04, which is supplied by Canonical, the publisher and maintainer of Ubuntu:

  ami:

    owners: "099720109477"

    filters:

      name: "ubuntu/images/hvm-ssd/ubuntu-jammy-22.04-amd64-server-\*"

      virtualization\_type: "hvm"

CopyExplain

Finally, we have some details on the keypair to upload to AWS and use when launching our Virtual Machine instances:

  keypair:

    name: "ssh\_keypair"

    key\_material: "{{ lookup('file', '~/.ssh/id\_rsa.pub') }}"

CopyExplain

Next up are the variables used when launching the RDS service.

RDS configuration

These are all standard, apart from the rds.db\_password variable:

rds:

  db\_username: "{{ app.name }}"

  db\_password: "{{ lookup('password', 'group\_vars/generated\_rds\_passwordfile chars=ascii\_letters,digits length=30') }}"

  db\_name: "{{ app.name }}"

  instance\_type: "db.t2.micro"

  engine: "mysql"

  engine\_version: "8.0"

  allocated\_storage: "5"

CopyExplain

As you can see, we are using a lookup module to add a random password to the group\_vars/generated\_rds\_passwordfile file; we are instructing the module to generate a 30-character random password comprising letters and numbers only.

EFS configuration

Here, we define the variables used to tell Ansible to wait and how long when creating the EFS resource:

efs:

  wait: "yes"

  wait\_time: "1200"

CopyExplain

VPC and subnet configuration

This block remains unchanged from [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*.

Security group configuration

Most of this block is unchanged from [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, as we now define the SSH port as ec2.ssh\_port. I have updated the EC2 group to use this reference rather than hardcoding port 22 into the block. The only other addition is the following:

elb\_seach\_string: "elb"

ec2\_seach\_string: "ec2"

rds\_seach\_string: "rds"

efs\_seach\_string: "efs"

CopyExplain

These will be used throughout the playbook when we query the AWS API for information on our security groups.

The final block

As per [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, this contains the following:

region: "{{ app.region }}"

CopyExplain

That concludes our whistle-stop tour of the group\_vars/common.yml file; as you can see, structure- and content-wise, we are following the same patterns as the last few chapters, where we group variables into logical blocks and trying to reuse references as much as possible throughout so that we don’t have to repeat information repeatedly.

The Playbook roles

Now that we have covered the variables, we can work through the roles in the order they appear in the site.yml file.

The VPC, subnets, gateway, and security groups roles

There are no changes to these roles from [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*; they are just dropped in place and work as expected. The remaining roles in this section of the Playbook will reference the output of these roles when referring to subnets, security groups, and the VPC.

The Application Elastic Load Balancer (ELB) role

In this role, we will deploy two resources, the first of which is a target group. This will be used when we launch our auto-scaling virtual machine instances – we attach our instances to the target group. Then, the target group is attached to the Application Elastic Load Balancer, which we will also launch in this role.

The task itself is pretty static, as you can see from the code for the following task:

- name: "Provision the target group"

  community.aws.elb\_target\_group:

    name: "{{ elb\_target\_group\_name }}"

    region: "{{ region }}"

    state: "{{ state }}"

    protocol: "http"

    port: "80"

    deregistration\_delay\_timeout: "15"

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

    modify\_targets: "false"

    tags:

      "Name": "{{ elb\_target\_group\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "target-group"

  register: elb\_target\_group\_output

CopyExplain

We are just referencing variables, with the only dynamic content being the ID of the VPC, which is referenced from the vpc\_output variable we registered when launching the VPC in the VPC role.

As we are registering some output in this role, we will continue by adding a debug task straight after; in this case, the task looks like the following:

- name: "Debug: ELB Target Group Output"

  ansible.builtin.debug:

    var: "elb\_target\_group\_output"

  when: debug\_output

CopyExplain

As we have already covered in [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, we will not be repeating these tasks in our overview of the Playbook unless we are doing something different – so, from now on, if we are registering an output, please assume that a debug task immediately follows.

There is one more bit of information we need before we create the ELB, and that’s the ID of the security group.

To get this, we can loop through the security\_groups\_with\_rules\_output variable and use set\_fact to set the group\_id when the group\_name contains the contents of the elb\_seach\_string variable:

- name: Extract ELB Group ID

  ansible.builtin.set\_fact:

    elb\_group\_id: "{{ item.group\_id }}"

  loop: "{{ security\_groups\_with\_rules\_output.results }}"

  when: item.group\_name is search(elb\_seach\_string)

CopyExplain

Whenever we need the ID of a security group, we will use this same pattern but update the name of the fact that is being set and the corresponding search steering variable.

The following task provisions the Application Elastic Load Balancer, which will be used to distribute HTTP requests across our auto-scaling managed virtual machine instances to serve our WordPress site:

- name: "Provision an application elastic load balancer"

  amazon.aws.elb\_application\_lb:

    region: "{{ region }}"

    name: "{{ elb\_name }}"

    state: "{{ state }}"

    security\_groups: "{{ elb\_group\_id }}"

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

    listeners:

      - Protocol: "HTTP"

        Port: "80"

        DefaultActions:

          - Type: "forward"

            TargetGroupArn: "{{ elb\_target\_group\_output.target\_group\_arn }}"

    tags:

      "Name": "{{ elb\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "load-balancer"

  register: loadbalancer\_output

CopyExplain

As you can see, we are attaching the Application Elastic Load Balancer to the subnets defined listed in the subnet\_public\_ids, and we are attaching the security group with the ID defined in the elb\_group\_id fact that registered in the previous task.

We are then configuring a listener on port 80 to accept HTTP traffic and forward it to the Target Group we launched at the start of the role – which concludes the Application Elastic Load balancer role.

The Elastic File System (EFS) role

The role starts with the task which sets the efs\_group\_id using the efs\_seach\_string variable. Once we know the ID of the security group we are applying to the EFS service, we can move on to the next task.

This task generates a file using a template and places it in the group\_vars folder:

- name: "Generate the efs targets vars file"

  ansible.builtin.template:

    src: "targets.j2"

    dest: "group\_vars/generated\_efs\_targets.yml"

    mode: "0644"

CopyExplain

The template file used to populate the file at group\_vars/generated\_efs\_targets.yml looks like the following:

efs\_targets:

{% for item in subnet\_storage\_ids %}

      - subnet\_id: "{{ item }}"

        security\_groups: [ "{{ efs\_group\_id }}" ]

{% endfor %}

CopyExplain

Here, we are using a Jinja2 for loop to loop through the contents of subnet\_storage\_ids, which will create a file that looks something like the following:

efs\_targets:

      - subnet\_id: "subnet01\_id"

        security\_groups: [ "efs\_group\_id" ]

      - subnet\_id: "subnet02\_id"

        security\_groups: [ "efs\_group\_id" ]

      - subnet\_id: "subnet03\_id"

        security\_groups: [ "efs\_group\_id" ]

CopyExplain

This means that when we create the EFS file system, it will be available across all the availability zones in our chosen region.

Well, it will be once we load in the contents of the file we have just loaded, which we do in the next task, as you can see here:

- name: "Include the efs targets vars file"

  ansible.builtin.include\_vars: "group\_vars/generated\_efs\_targets.yml"

CopyExplain

We now have everything in place to create the EFS file system, which is done using this task:

- name: "Create the EFS File System"

  community.aws.efs:

    name: "{{ efs\_name }}"

    region: "{{ region }}"

    state: "{{ state }}"

    tags:

      "Name": "{{ efs\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "efs"

    targets: "{{ efs\_targets }}"

    wait: "{{ efs.wait }}"

    wait\_timeout: "{{ efs.wait\_time }}"

  register: efs\_output

CopyExplain

It can take a few minutes to create the file system, and we must wait until this task has succeeded before we continue, which is why we are using the wait flag. If we don’t wait, we increase the risk that the file system will not be ready by the time our virtual machine is launched and unable to mount it, which will cause the Playbook execution to fail.

Speaking of tasks that take a while, the next role deals with launching the Amazon RDS instance, which we will use as the database for our WordPress site. This task can take up to 10 minutes to complete.

The Amazon RDS role

There are two main parts to the role; the first does a similar task to the one we had to do in the previous role when we created the targets for the EFS to be attached to.

The RDS service differs in that rather than passing in the subnets manually when we deploy the service, we can create a group natively on the AWS side and then reference it when we launch the RDS instance.

The task to create the RDS subnet group looks like the following:

- name: "Add RDS subnet group"

  amazon.aws.rds\_subnet\_group:

    name: "{{ rds\_name }}"

    region: "{{ region }}"

    state: "{{ state }}"

    description: "{{ dict.ansible\_warning }}"

    subnets: "{{ subnet\_database\_ids }}"

    tags:

      "Name": "{{ rds\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "rds"

  register: rds\_subnet\_group\_output

CopyExplain

Once we have created the subnet group, we need to find the security group ID using the rds\_seach\_string variable and set a fact called rds\_group\_id.

Now we have all the information we need to launch the RDS instance, the task for which looks like the following:

- name: "Create the RDS instance"

  amazon.aws.rds\_instance:

    id: "{{ rds\_name }}"

    region: "{{ region }}"

    state: "{{ state }}"

    db\_instance\_class: "{{ rds.instance\_type }}"

    engine: "{{ rds.engine }}"

    engine\_version: "{{ rds.engine\_version }}"

    allocated\_storage: "{{ rds.allocated\_storage }}"

    username: "{{ rds.db\_username }}"

    password: "{{ rds.db\_password }}"

    db\_name: "{{ rds.db\_name }}"

    db\_subnet\_group\_name: "{{ rds\_subnet\_group\_output.subnet\_group.name }}"

    vpc\_security\_group\_ids: ["{{ rds\_group\_id }}"]

    tags:

      "Name": "{{ rds\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "rds"

  register: rds\_instance\_output

CopyExplain

As mentioned at the end of the previous task, this can take quite a while to deploy, typically just over 10 minutes, so when we run the Playbook, this task will appear to have stalled.

So please do not worry – it is busy working away in the background.

Once this role has finished running, we will have all the core AWS resources we need to launch an EC2 instance, perform the software configuration, and install WordPress.

The temporary EC2 instance role

Before we work through the tasks that launch the temporary instance, let’s go into a little more detail on why we need a temporary EC2 instance in the first place.

As we mentioned in the introduction, this instance will be running Ubuntu, and we will be targeting it with slightly modified copies of the stack\_install, stack\_config, and wordpress roles that we first ran locally in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, and against a single cloud instance in [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud*.

One of the modifications we will be making to the roles is installing the software needed to mount our EFS, which we will then use to store the WordPress code and supporting files for our WordPress installation, meaning that we have everything we need file-wise for WordPress on a shared file system we can then mount on multiple virtual machine instances.

The second change is that rather than installing a database server on our local instance, we will be using the Amazon RDS database service for WordPress, meaning that we can have multiple instances of WordPress, all being able to connect to a single remote database.

Great, you may be thinking to yourself, but that doesn’t explain why this is a temporary instance.

Well, once everything has been installed, mounted, configured, and WordPress bootstrapped, we will be making our own **Amazon Machine Image** (**AMI**) and terminating the temporary EC2 instance. Once it’s been terminated, we will take the AMI and configure our Auto Scaling Group to use the newly created image, which will either trigger the deployment of new hosts if it is our first time running the Playbook or it will launch more instances and terminate the old ones if we have already had virtual machine instances running our WordPress installation.

When these virtual machine instances boot up using our custom AMI, they will already have NGINX and PHP installed and configured, ready to serve WordPress, and the EFS containing our WordPress files will be mounted, meaning that our servers will be good to go as soon as they are deployed.

All of this means our WordPress installation should be sound to scale up if we have an influx of traffic hitting the site for whatever reason, and all instances of our virtual machines will be running a known good configuration; in fact, it will be the same configuration as the other hosts serving our WordPress site.

Just as important, as we are not relying on anything on the local virtual machine instances filesystem, we are just as good at automatically scaling down by terminating hosts automatically when the influx of traffic has subsided without the risk of data loss or availability.

If this approach is planned right – in theory, we don’t even need SSH access to the hosts launched by the Auto Scaling Group as we should never need to manage them manually, and we can treat them as short-lived instances where we don’t have to care if they are running or terminated – just that we have the desired of instances delivering our application.

So, now that we know why we are taking this approach, let’s return to the Playbook and look at the tasks needed to get this temporary EC2 instance up and running to the point where we can SSH to it and install our software and WordPress.

The first task is to get a list of all the Ubuntu AMIs using the variables we covered earlier in the chapter:

- name: "Gather information about AMIs with the specified filters"

  amazon.aws.ec2\_ami\_info:

    region: "{{ region }}"

    owners: "{{ ec2.ami.owners }}"

    filters:

      name: "{{ ec2.ami.filters.name }}"

      virtualization-type: "{{ ec2.ami.filters.virtualization\_type }}"

  register: ubuntu\_ami\_info

CopyExplain

The list of AMIs returned will contain all of the various AMI versions for our chosen Ubuntu version; we only need to know the ID of the latest version published by Canonical (the publisher and maintainer of Ubuntu) so we know we are using the most up-to-date image that contains the latest patches and any bug fixes.

Luckily, each AMI returned in the list has a key called creation\_date, the value of which, as you may have guessed, is the date and time the AMI was published. This means we can run the following task to get the ID of the latest version of the AMI:

- name: "Filter the list of AMIs to find the latest one"

  ansible.builtin.set\_fact:

    ami: "{{ ubuntu\_ami\_info.images | sort(attribute='creation\_date') | last }}"

CopyExplain

As you can see, this takes the content of the list, which is defined as ubuntu\_ami\_info.images, sorts the list by creation\_date, and then takes the ID of the last AMI in the list as, by default, they are sorted in ascending order.

Now that we know the ID of the most up-to-date Ubuntu AMI, we can progress with more preparation work before launching our EC2 instance.

We now need to create an SSH key pair on the AWS side. This will contain the public portion of the SSH key we will use to access the EC2 instance when it is launched – the task to configure this looks like the following and uses the variables we covered earlier in the chapter to get the contents of the public portion of our SSH key:

- name: "Create a SSH Key Pair"

  amazon.aws.ec2\_key:

    region: "{{ region }}"

    state: "{{ state }}"

    name: "{{ ec2.keypair.name }}"

    key\_material: "{{ ec2.keypair.key\_material }}"

    tags:

      "Name": "{{ ec2.keypair.name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "ssh\_keypair"

  register: keypair\_output

CopyExplain

Finally, before we launch our EC2 instance, we need the ID of the security group, which allows the public IP address of our host running Ansible SSH access to the EC2 instance. To do this, we set a fact called ec2\_group\_id using the ec2\_seach\_string variable to find the correct group ID.

Now, we have everything in place to launch the EC2 instance using the following task:

- name: "Create the temporary ec2 instance"

  amazon.aws.ec2\_instance:

    name: "{{ ec2\_tmp\_name }}"

    region: "{{ region }}"

    state: "{{ state }}"

    vpc\_subnet\_id: "{{ subnet\_compute\_ids[0] }}"

    instance\_type: "{{ ec2.instance\_type }}"

    security\_group: "{{ ec2\_group\_id }}"

    key\_name: "{{ ec2.keypair.name }}"

    network:

      assign\_public\_ip: "{{ ec2.public\_ip }}"

    image\_id: "{{ ami.image\_id }}"

    tags:

      Name: "{{ ec2\_tmp\_name }}"

      Description: "{{ dict.ansible\_warning }}"

      Project: "{{ app.name }}"

      Environment: "{{ app.env }}"

      Deployed\_by: "Ansible"

      Role: "tmp"

  register: ec2\_tmp\_instance\_output

CopyExplain

The only thing pointed out in the preceding task is that when we add the value for the vpc\_subnet\_id we can only pass in a single ID. As we don’t need this virtual machine instance to be highly available, that is not a problem, so we are using the first ID in the list of subnet IDs by using the {{ subnet\_compute\_ids[0] }}.

When launching an EC2 instance in AWS, it goes through a few stages and, by default, the amazon.aws.ec2\_instance module creates the instance and doesn’t wait for the status to change from *creating* to *running*.

Our next task polls the AWS API waiting for the status of our EC2 instance to be *running*:

- name: "Get information about the temporary EC2 instance to see if it is running"

  amazon.aws.ec2\_instance\_info:

    region: "{{ region }}"

    filters:

      instance-id: "{{ ec2\_tmp\_instance\_output.instances[0].instance\_id }}"

  register: ec2\_tmp\_instance\_state

  delay: 5

  retries: 50

  until: ec2\_tmp\_instance\_state.instances[0].state.name == "running"

CopyExplain

As you can see, the previous task takes the ID of our newly created EC2 instance and polls the AWS API every 5 seconds, a maximum of 50 times, until the value of ec2\_tmp\_instance\_state.instances[0].state.name is equal to running.

You might think to yourself that it seems a bit overkill to do that, and 99% of the time, you would be correct – it usually takes no more than a few checks for the status to change. Still, there is the odd occasion that AWS might be on a “go-slow,” and during testing, I have seen it take up to 15 checks, or just over a minute, for the status to change, so we need to take this delay into account in our Playbook as it could break the Playbook execution if we don’t.

The next task takes the details, the DNS name and IP address, of our now-running EC2 instance and adds them to the host group called vmgroup:

- name: "Add the temporary EC2 instance to the vmgroup"

  ansible.builtin.add\_host:

    name: "{{ ec2\_tmp\_instance\_output.instances[0].public\_dns\_name }}"

    ansible\_ssh\_host: "{{ ec2\_tmp\_instance\_output.instances[0].public\_ip\_address }}"

    groups: "vmgroup"

CopyExplain

Before we hand off to the next role, we should perform one more check.

Sometimes, the Ansible Playbook works through the tasks so quickly that it is possible that even though our EC2 instance has a status of *running*, it does not mean that the host has finished booting, and SSH is started and is accessible:

- name: "Wait for the temporary EC2 instance to be ready to accept SSH connections"

  ansible.builtin.wait\_for:

    host: "{{ ec2\_tmp\_instance\_output.instances[0].public\_ip\_address }}"

    port: "{{ ec2.ssh\_port }}"

    delay: 10

    timeout: 300

CopyExplain

Now that we have confirmation that our EC2 host is accessible to our machine running Ansible using SSH, we can proceed to the final role in this section of the site.yml file.

The endpoints role

This role has a single task, which creates a file at generated\_aws\_endpoints.yml containing the name of the AWS endpoints for the EFS, RDS, and ELB resources we have created:

- name: "Generate the aws endpoints file"

  ansible.builtin.template:

    src: "endponts.j2"

    dest: "group\_vars/generated\_aws\_endpoints.yml"

    mode: "0644"

CopyExplain

The endponts.j2 template file looks like the following:

aws\_endpoints:

  efs: "{{ efs\_output.efs.filesystem\_address.split(':')[0] }}"

  rds: "{{ rds\_instance\_output.endpoint.address }}"

  elb: "{{ loadbalancer\_output.dns\_name }}"

CopyExplain

Both the RDS and ELB endpoints are straightforward enough; for the EFS, you might notice something at the end – what is that for?

None of the output that is registered under the efs\_output.efs variable contains just the address of the EFS endpoint. The one we are using, filesystem\_address, has information on the file system mount, which is represented by appending :/ to the end of the DNS address we need.

To get around this, we are using the split function, passing : as the delimiter and then taking the first section, which is defined as 0, meaning that we end up with everything before the :, which is the DNS name we are after.

Now that we have a populated group\_vars/generated\_aws\_endpoints.yml file, we can load it into the second section of the site.yml file as a variable file, saving us from having to interact with the AWS from our EC2 instance.

So, now that we have our EC2 instance up and running, let’s get our software stack installed, configured, and WordPress bootstrapped.

The stack install role

The tasks in this role remain unchanged from the previous times we have executed the Playbook because all the changes we have made are in the stack\_packages variable we are passing in.

As a reminder, this role does the following:

* Updates the APT cache and ensures that the installed packages are running the latest available versions – which shouldn’t be too many as we are using the newest AMI
* Imports the APT keys for the additional repositories we will be enabling
* Installs the packages containing details of the additional repositories and enables them
* Installs the packages listed in the system\_packages, extra\_packages, and stack\_packages variables – system\_packages and extra\_packages contain the default values we have been using throughout, and because we are passing the updated stack\_packages variable via the group\_vars/common.yml file, this overrides the default values from previous chapters which are still defined in the roles/stack\_install/defaults/main.yml file

This leaves us with all the base software we need to install on the EC2 instance.

The stack configuration role

Unlike the previous role, there are some amendments to this role, starting with additional tasks out of the gate.

Three tasks are added to the top of roles/stack\_config/tasks/main.yml, the first of which is a continuation of the checks we did towards the end of the roles in the last section of the site.yml file:

- name: "Check that the EFS volume is ready"

  ansible.builtin.wait\_for:

    host: "{{ aws\_endpoints.efs }}"

    port: "2049"

    delay: 10

    timeout: 300

CopyExplain

As you can see, this checks that port 2049 is accessible at the endpoint defined in aws\_endpoints.efs; the reason why this is there is that while the EFS service is ready, it may take a little while for the DNS records for the endpoint to be updated and accessible within the VPC. As we will soon attempt to mount the EFS filesystem, we must ensure it is accessible before proceeding.

The next task is to ensure that the RPC Bind service is up and running; we will need to mount the EFS file system:

- name: "ensure rpcbind service is running"

  ansible.builtin.service:

    name: "rpcbind"

    state: "started"

    enabled: true

CopyExplain

The final additional task mounts the EFS and ensures that it is added to the file system configuration to ensure that from now on, the EFS is mounted when the EC2 instance boots:

- name: "mount the EFS volume"

  ansible.posix.mount:

    src: "{{ aws\_endpoints.efs }}:/"

    path: "{{ nfs.mount\_point }}"

    opts: "{{ nfs.mount\_options }}"

    state: "{{ nfs.state }}"

    fstype: "{{ nfs.fstype }}"

CopyExplain

As you will have already seen from when we covered the variables at the start of the chapter, we are mounting the EFS at /var/www/; we are making sure to do this before the following two tasks to ensure that our WordPress users home directory is created on the share.

These two tasks remain unchanged from the last time we installed WordPress, as does the value of wordpress\_system.home, which is /var/www/wordpress.

So, now that we have created our WordPress user and group, we can proceed with the rest of the tasks:

* Update /etc/nginx/nginx.conf with some sensible defaults
* Create the configuration for our default host at /etc/nginx/conf.d/default.conf
* Create the /etc/nginx/global directory and copy the restrictions.conf and wordpress\_shared.conf files there

The next task is more of a quality-of-life improvement to do with the way our Playbook deals with PHP, as this Playbook is designed to keep our WordPress installation up to date by taking the base Ubuntu image and bootstrapping from scratch each time rather than managing the configuration in place. It is possible that the version of PHP could change at some point during the life of our WordPress installation.

So far, whenever the stack\_config role has been executed, it has been using the following variables:

php\_fpm\_path: "/etc/php/8.1/fpm/pool.d/www.conf"

php\_ini\_path: "/etc/php/8.1/fpm/php.ini"

php\_service\_name: "php8.1-fpm"

CopyExplain

As you can see, 8.1 is a hardcoded value. While we can overwrite these variables at the variable level elsewhere in our configuration, it would be better to work out which version of PHP is installed at runtime and reference that.

To do this, we can update these values as follows:

php\_fpm\_path: "/etc/php/{{ php\_version }}/fpm/pool.d/www.conf"

php\_ini\_path: "/etc/php/{{ php\_version }}/fpm/php.ini"

php\_service\_name: "php{{ php\_version }}-fpm"

CopyExplain

This means we now must find a way to populate the php\_version variable with the relevant version of PHP.

To do this, we can run the php -v command, which returns a lot of information on the version of PHP installed. We then use the head and a few cut commands on the Linux command line using the ansible.builtin.shell and not a built-in Ansible function:

- name: "Get the PHP version"

  ansible.builtin.shell:

    cmd: "php -v | head -n 1 | cut -d ' ' -f 2 | cut -c 1-3"

  register: php\_version\_output

CopyExplain

Here is a detailed breakdown of the command we are getting Ansible to run:

* php -v: This command, when run, outputs the version information of the PHP installed on the host the command is being executed on; this output is typically a multi-line text that includes the PHP version along with additional information on how the version of the PHP was compiled.
* |: This symbol is known as a pipe. It takes the command output on its left (in this case, php -v) and uses it as the input for the command on its right. It’s a way of passing data between programs.
* head -n 1: This command processes the input received from the previous command; the head command outputs the first part of the files or data it receives. -n 1 is an option that tells head to output only the first line. So, in our case, head -n 1 takes the multiple lines of output from php -v and returns just the very first line.
* |: Another pipe, which again passes the command output on its left, head -n 1, to the command on its right.
* cut -d ' ' -f 2: This command is used for cutting out sections of each input line. -d ' ' is an option where -d stands for the delimiter, and ' ' (a space) is the delimiter being used. This tells cut to divide each line into sections based on spaces. -f 2 means *field 2*. This option tells the cut command to select the second field of the line in the standard format of the PHP version output; this field should be the version number.
* |: Again, we have another pipe, passing the output, now just the version number, to the following command.
* cut -c 1-3: This further processes the version number. -c 1-3 tells cut to return only the characters in positions 1 through 3 of the string it receives. For a typical PHP version such as 8.2.1, this would result in 8.2, which is precisely what we need to proceed with the rest of our tasks.

We can then take the output and register it as php\_version\_output, and set the php\_version variable as a fact:

- name: "Set the PHP version"

  ansible.builtin.set\_fact:

    php\_version: "{{ php\_version\_output.stdout }}"

CopyExplain

Now that we have the PHP version, we can proceed with the remainder of the PHP tasks, which copy the www.conf file to /etc/php/{{ php\_version }}/fpm/pool.d/www.conf and also update the PHP.ini file at /etc/php/{{ php\_version }}/fpm/php.ini.

With those files in place, we start the PHP-FPM and NGINX services, ensuring that they are set to start on boot.

The final task in the role is to create the ~/.my.cnf file and populate it with the information of our Amazon RDS instance. All of the other MariaDB tasks, which are there to start and configure our local MariaDB server, are commented out as we no longer install a local database server, so we don’t need to run the tasks to configure it.

The WordPress role

There are just two tasks commented out in this role. The tasks that create the database and the database user are not needed because when the Amazon RDS instance started, the database and user were made for us, meaning these two tasks are redundant.

All other tasks remain; for more details, see [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*.

The EC2 AMI role

Now that our software stack is installed and configured and WordPress is sorted, it is time to create the AMI from our temporary instance.

The first thing we need to do is get the details on our temporary EC2 instance; as our host group contains the DNS name of the instance, we can use this:

- name: "Find out some facts about the instance we have been using"

  amazon.aws.ec2\_instance\_info:

    region: "{{ region }}"

    filters:

      dns-name: "{{ groups['vmgroup'] }}"

  register: our\_instance

CopyExplain

Now that we have the information on the instance we would like to create the AMI from registered as our\_instance, we can proceed with the AMI creation:

- name: "Create the AMI"

  amazon.aws.ec2\_ami:

    region: "{{ region }}"

    state: "{{ state }}"

    instance\_id: "{{ our\_instance.instances[0].instance\_id }}"

    wait: "yes"

    name: "{{ ami\_name }}-{{ ansible\_date\_time.date }}\_{{ ansible\_date\_time.hour }}{{ ansible\_date\_time.minute }}"

    tags:

      "Name": "{{ ami\_name }}-{{ ansible\_date\_time.date }}\_{{ ansible\_date\_time.hour }}{{ ansible\_date\_time.minute }}"

      "buildDate": "{{ ansible\_date\_time.date }} {{ ansible\_date\_time.time }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "{{ playbook\_dict.ami }}"

  register: ami\_output

CopyExplain

There are just a few things to point out here. As you can see, we are using ansible\_date\_time to generate the date and get the current time as an hour and minute. We are using this both to give a unique name for the AMI and add a tag called buildDate.

The reason why we are using both the date and time is that it could be possible that we will need to create multiple AMIs on a single day, so it is important that we can easily identify them by name.

Once the AMI is created, we do not need the temporary instance, so we can terminate it:

- name: "Remove any temporary  instances which are running"

  amazon.aws.ec2\_instance:

    region: "{{ region }}"

    state: "absent"

    name: "{{ ec2\_tmp\_name }}"

    filters:

      instance-state-name: "running"

      "tag:Name": "{{ ec2\_tmp\_name }}"

      "tag:Role": "tmp"

      "tag:Project": "{{ app.name }}"

CopyExplain

Once the EC2 instance has been terminated, there is one more task in the role:

- name: "Wait for 2 minutes before continuing"

  ansible.builtin.pause:

    minutes: 2

CopyExplain

This does exactly what it says: it pauses the Playbook execution for 2 minutes.

I have included this because there was the odd occasion where the AMI was created and shown as available. Still, for some reason, it takes a short while for it to appear in the results when we query the Amazon API to find our AMIs, so rather than introduce a potential error when the next role starts, I have found it best to wait a minute or two.

The auto-scaling role

We have arrived at the final role of the Playbook; in this role, we will create all the resources needed to deploy EC2 instances using our newly created AMI and register them with the ELB to access our WordPress site.

The first thing we need to do is grab a list of all our AMIs from the API:

- name: "Search for all of our AMIs"

  amazon.aws.ec2\_ami\_info:

    region: "{{ region }}"

    filters:

      name: "{{ ami\_name }}-\*"

  register: ami\_find

CopyExplain

Now that we have a list of AMIs, we need to filter out the most recent one. To do this, we use the same logic that we used when launching the temporary EC2 instance:

- name: "Find the last one we built"

  ansible.builtin.set\_fact:

    ami\_sort\_filter: "{{ ami\_find.images | sort(attribute='creation\_date') | last }}"

CopyExplain

Now that we have filtered our list of AMIs down to the latest one, we need to set two facts, one for the name of the AMI and the other containing the ID of the AMI:

- name: "Grab AMI ID and name of the most recent result"

  ansible.builtin.set\_fact:

    our\_ami\_id: "{{ ami\_sort\_filter.image\_id }}"

    our\_ami\_name: "{{ ami\_sort\_filter.name }}"

CopyExplain

The final bit of information we need before we start creating/updating resources is the ID of the security group we are using for the EC2 instances.

As before, we use the ec2\_seach\_string variable to find the correct group ID and set a fact called ec2\_group\_id.

Next up, we need to create or update a launch template if one already exists.

A launch template contains the basic configuration for the instances we will be launching in the auto-scaling group:

- name: "Create the launch template"

  community.aws.ec2\_launch\_template:

    region: "{{ region }}"

    state: "{{ state }}"

    name: "{{ launch\_template\_name }}"

    version\_description: "{{ our\_ami\_name }}"

    image\_id: "{{ our\_ami\_id }}"

    security\_group\_ids: ["{{ ec2\_group\_id.security\_groups[0].group\_id }}"]

    instance\_type: "{{ ec2.instance\_type }}"

    disable\_api\_termination: "{{ ec2.asg.disable\_api\_termination }}"

    tags:

      "Name": "{{ ec2\_name }}"

      "projectName": "{{ app.name }}"

      "environment": "{{ app.env }}"

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

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

      "role": "launchTemplate"

CopyExplain

With this task, we create the launch template and then publish a version called after the name of our AMI so that we can quickly identify it; we then attach the corresponding AMI ID and security group ID and set the spec of the instances we want to launch.

With the launch template in place, we need to gather a few more bits of information from the AWS API before creating the auto-scaling group.

First, we need the ID of the target group that we created in the ELB role:

- name: "Find out the target group ARN"

  community.aws.elb\_target\_group\_info:

    region: "{{ region }}"

    names:

      - "{{ elb\_target\_group\_name }}"

  register: elb\_target\_group\_output

CopyExplain

We then need the IDs of the subnets where we are going to be deploying the EC2 instances launched as part of auto-scaling group, the following task gathers information on the subnets:

- name: "Get information on the ec2 subnets"

  amazon.aws.ec2\_vpc\_subnet\_info:

    region: "{{ region }}"

    filters:

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

  register: ec2\_subnet\_output

CopyExplain

Now that we have the information on the subnets, we need to extract just the IDs of each of the subnets and create a list:

- name: "Create a list of subnet IDs"

  ansible.builtin.set\_fact:

    subnet\_ec2\_ids: "{{ subnet\_ec2\_ids | default([]) + [item.subnet\_id] }}"

  loop: "{{ ec2\_subnet\_output.subnets }}"

CopyExplain

This is the final bit of information we need, and we can now proceed with creating or updating the auto-scaling group:

- name: "Create/update the auto-scaling group using the launch template we just created"

  amazon.aws.autoscaling\_group:

    region: "{{ region }}"

    state: "{{ state }}"

    name: "{{ asg\_name }}"

    target\_group\_arns: ["{{ elb\_target\_group\_output.target\_groups[0].target\_group\_arn }}"]

    launch\_template:

      launch\_template\_name: "{{ launch\_template\_name }}"

    min\_size: "{{ ec2.asg.min\_size }}"

    max\_size: "{{ ec2.asg.max\_size }}"

    desired\_capacity: "{{ ec2.asg.desired\_capacity }}"

    health\_check\_period: "{{ ec2.asg.health\_check\_period }}"

    health\_check\_type: "{{ ec2.asg.health\_check\_type }}"

    replace\_all\_instances: "{{ ec2.asg.replace\_all\_instances }}"

    replace\_batch\_size: "{{ ec2.asg.replace\_batch\_size }}"

    vpc\_zone\_identifier: "{{ subnet\_ec2\_ids }}"

    wait\_for\_instances: "{{ ec2.asg.wait\_for\_instances }}"

    wait\_timeout: "{{ ec2.asg.wait\_timeout }}"

    tags:

      - key: "Name"

        value: "{{ ec2\_name }}"

        propagate\_at\_launch: true

      - key: "Project"

        value: "{{ app.name }}"

        propagate\_at\_launch: true

      - key: "Environment"

        value: "{{ app.env }}"

        propagate\_at\_launch: true

      - key: "Deployed\_by"

        value: "Ansible"

        propagate\_at\_launch: true

  register: ec2\_asg\_output

CopyExplain

There is quite a lot happening in this, the final resource we will be launching, so let’s go into more detail.

First, we have the basic configuration standard across most of the AWS-related modules we have called throughout this Playbook; here, we are setting the name, region, and state of the resource, which will be present for this playbook.

Next up, we must provide the Target Group **Amazon Resource Names** (**ARNs**). The target\_group\_arns key specifies the ARNs of the target groups for the load balancer, which we set to the first target group ARN from elb\_target\_group\_output and then the launch\_template key references the launch template by its name, set to the value of launch\_template\_name.

Now we have the size and capacity settings; the min\_size, max\_size, and desired\_capacity keys are set using ec2.asg.min\_size, ec2.asg.max\_size, and ec2.asg.desired\_capacity variables, which define the auto-scaling group’s minimum, maximum, and desired number of instances.

We then have the health check configuration, setting the health\_check\_period and health\_check\_type keys to control how the health of the instances in the **auto scaling group** (**ASG**) is checked.

Now we have the Instance Replacement Settings. The replace\_all\_instances and replace\_batch\_size keys instruct whether all instances should be replaced and provide the batch size for replacing instances, respectively.

Then, we have the Network Configuration, setting vpc\_zone\_identifier to use the list of subnet IDs stored in subnet\_ec2\_ids to distribute the instances in the ASG across those subnets and availability zones.

Next up are the Wait Settings, which control whether the task should wait for the instances to have a status of running and the maximum time to wait for that condition to be met.

Finally, you will have noticed that we are tagging in a pretty different way than we have been doing throughout the rest of the Playbook; the task defines several tags (Name, Project, Environment, and Deployed\_by) with respective values, all marked to propagate at launch, which means that the EC2 instances launched by the auto-scaling group will each inherit these tags when they are launched.

This concludes our walk-through of the Playbook. As you will have seen, we extended our original AWS networking Playbook from [*Chapter 10*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/10), *Building Out a Cloud Network*, to encompass more services as well as integrating our WordPress roles from the Playbook we covered in [*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress* – all that is left now is run the playbook.

Running the Playbook

Now that we have all the roles needed to deploy our resources into AWS, we can run the playbook. To start with, we need to let Ansible know our access key and secret by running the following commands with your own credentials to set the environment variables:

$ export AWS\_ACCESS\_KEY=AKIAI5KECPOTNTTVM3EDA

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

CopyExplain

With environment variables set, you kick off the Ansible run by using the following command:

$ ansible-playbook -i hosts site.yml

CopyExplain

Unlike previous chapters, where we just looked at the end of the playbook run, here we will look at some highlights of what happens when we deploy our resources.

Playbook run highlights

This is not the complete playbook output, and when running the playbook, I have not enabled debug, so all those tasks will be skipped.

We start with the VPC:

PLAY [Deploy and configure the AWS Environment] \*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [roles/vpc : Create VPC] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

We now have somewhere to put the subnets once we have gathered some information on the availability zones in our chosen region:

TASK [roles/subnets : Get some information on the available zones] \*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

CopyExplain

Once we have that information, it will loop through and include the create\_subnet.yml tasks:

TASK [roles/subnets : Create all subnets] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

included: create\_subnet.yml for localhost => (item={'name': 'ec2', 'role': 'compute'})

included: create\_subnet.yml for localhost => (item={'name': 'rds', 'role': 'database'})

included: create\_subnet.yml for localhost => (item={'name': 'efs', 'role': 'storage'})

included: create\_subnet.yml for localhost => (item={'name': 'dmz', 'role': 'public'})

CopyExplain

We then get the results of each of the four included task runs, the first of which looks like the following:

TASK [roles/subnets : Create subnet in the availability zone] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'state': 'available', 'opt\_in\_status': 'opt-in-not-required', 'messages': [], 'region\_name': 'eu-west-1', 'zone\_name': 'eu-west-1a', 'zone\_id': 'euw1-az1', 'group\_name': 'eu-west-1', 'network\_border\_group': 'eu-west-1', 'zone\_type': 'availability-zone'})

changed: [localhost] => (item={'state': 'available', 'opt\_in\_status': 'opt-in-not-required', 'messages': [], 'region\_name': 'eu-west-1', 'zone\_name': 'eu-west-1b', 'zone\_id': 'euw1-az2', 'group\_name': 'eu-west-1', 'network\_border\_group': 'eu-west-1', 'zone\_type': 'availability-zone'})

changed: [localhost] => (item={'state': 'available', 'opt\_in\_status': 'opt-in-not-required', 'messages': [], 'region\_name': 'eu-west-1', 'zone\_name': 'eu-west-1c', 'zone\_id': 'euw1-az3', 'group\_name': 'eu-west-1', 'network\_border\_group': 'eu-west-1', 'zone\_type': 'availability-zone'})

CopyExplain

As you can see, a subnet is created for each of the zones in the eu-west-1 region – this is then repeated three more times. Once the subnets have all been added, we grab more information on what has been created.

Next, the Internet Gateway role is run:

TASK [roles/gateway : Create an Internet Gateway] \*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [roles/gateway : Create a route table so the internet gateway can be used by the public subnets] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

As you may have remembered, there isn’t much happening in that role, unlike the next one, which adds the network security groups, where we start by getting your current public IP address:

TASK [roles/securitygroups : Find out your current public IP address using https://ipify.org/] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [roles/securitygroups : Set your public ip as a fact]\*

ok: [localhost]

CopyExplain

As you may recall, we create the two groups in two parts – first, we create the base groups:

TASK [roles/securitygroups : Create the base security groups] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'name': 'learnansible-elb-security-group', 'description': 'opens port 80 and 443 to the world', 'id\_var\_name': 'elb\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 80'}, {'proto': 'tcp', 'from\_port': '443', 'to\_port': '443', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 443'}]})

changed: [localhost] => (item={'name': 'learnansible-ec2-security-group', 'description': 'opens port 22 to a trusted IP and port 80 to the elb group', 'id\_var\_name': 'ec2\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '22', 'to\_port': '22', 'cidr\_ip': '86.177.22.88/32', 'rule\_desc': 'allow 86.177.22.88/32 access to port 22'}, {'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'group\_id': '', 'rule\_desc': 'allow access to port 80 from ELB'}]})

changed: [localhost] => (item={'name': 'learnansible-rds-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': '', 'rule\_desc': 'allow  access to port 3306'}]})

changed: [localhost] => (item={'name': 'learnansible-efs-security-group', 'description': 'opens port 2049 to the ec2 instances', 'id\_var\_name': 'efs\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '2049', 'to\_port': '2049', 'group\_id': '', 'rule\_desc': 'allow  access to port 2049'}]})

CopyExplain

Then we get information on the bases we have just launched and set them as facts:

TASK [roles/securitygroups : Set the fact for the security group ids] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost] => (item={'name': 'learnansible-elb-security-group', 'description': 'opens port 80 and 443 to the world', 'id\_var\_name': 'elb\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 80'}, {'proto': 'tcp', 'from\_port': '443', 'to\_port': '443', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 443'}]})

ok: [localhost] => (item={'name': 'learnansible-ec2-security-group', 'description': 'opens port 22 to a trusted IP and port 80 to the elb group', 'id\_var\_name': 'ec2\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '22', 'to\_port': '22', 'cidr\_ip': '86.177.22.88/32', 'rule\_desc': 'allow 86.177.22.88/32 access to port 22'}, {'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'group\_id': '', 'rule\_desc': 'allow access to port 80 from ELB'}]})

ok: [localhost] => (item={'name': 'learnansible-rds-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': '', 'rule\_desc': 'allow  access to port 3306'}]})

ok: [localhost] => (item={'name': 'learnansible-efs-security-group', 'description': 'opens port 2049 to the ec2 instances', 'id\_var\_name': 'efs\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '2049', 'to\_port': '2049', 'group\_id': '', 'rule\_desc': 'allow  access to port 2049'}]})

CopyExplain

Lastly, we then add the rules; you will notice from the output that we are passing in the IDs of the groups we have created so that we can use them as part of the rules:

TASK [roles/securitygroups : Provision security group rules] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'name': 'learnansible-elb-security-group', 'description': 'opens port 80 and 443 to the world', 'id\_var\_name': 'elb\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 80'}, {'proto': 'tcp', 'from\_port': '443', 'to\_port': '443', 'cidr\_ip': '0.0.0.0/0', 'rule\_desc': 'allow all on port 443'}]})

changed: [localhost] => (item={'name': 'learnansible-ec2-security-group', 'description': 'opens port 22 to a trusted IP and port 80 to the elb group', 'id\_var\_name': 'ec2\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '22', 'to\_port': '22', 'cidr\_ip': '86.177.22.88/32', 'rule\_desc': 'allow 86.177.22.88/32 access to port 22'}, {'proto': 'tcp', 'from\_port': '80', 'to\_port': '80', 'group\_id': 'sg-04f31e782e30e1f0a', 'rule\_desc': 'allow access to port 80 from ELB'}]})

changed: [localhost] => (item={'name': 'learnansible-rds-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': 'sg-05bffd3eb96602519', 'rule\_desc': 'allow sg-05bffd3eb96602519 access to port 3306'}]})

changed: [localhost] => (item={'name': 'learnansible-efs-security-group', 'description': 'opens port 2049 to the ec2 instances', 'id\_var\_name': 'efs\_group\_id', 'rules': [{'proto': 'tcp', 'from\_port': '2049', 'to\_port': '2049', 'group\_id': 'sg-05bffd3eb96602519', 'rule\_desc': 'allow sg-05bffd3eb96602519 access to port 2049'}]})

CopyExplain

Now, with the rules configured, we can start deploying some resources that use them, starting with the Target Group and ELB:

TASK [roles/elb : Provision the target group] \*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [roles/elb : Provision an application elastic load balancer] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Then EFS:

TASK [roles/efs : Generate the efs targets vars file] \*\*\*\*\*

changed: [localhost]

TASK [roles/efs : Include the efs targets vars file] \*\*\*\*\*\*

ok: [localhost]

TASK [roles/efs : Create the EFS File System] \*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Now RDS:

TASK [roles/rds : Add RDS subnet group] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [roles/rds : Create the RDS instance] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Now it is time to create the temporary EC2 instance. First, we find the AMI to use:

TASK [roles/ec2tmp : Gather information about AMIs with the specified filters] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [roles/ec2tmp : filter the list of AMIs to find the latest one] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

CopyExplain

Then, we create the SSH key pair:

TASK [roles/ec2tmp : Create an SSH Key Pair] \*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Then, we create the EC2 instance itself:

TASK [roles/ec2tmp : Create the temporary ec2 instance] \*\*\*

changed: [localhost]

CopyExplain

With the instance configured, we need to wait for it to have a status of **running**:

TASK [roles/ec2tmp : Get information about the temporary EC2 instance to see if it is running] \*\*\*

FAILED - RETRYING: [localhost]: Get information about the temporary EC2 instance to see if it is running (50 retries left).

. . . .

FAILED - RETRYING: [localhost]: Get information about the temporary EC2 instance to see if it is running (46 retries left).

ok: [localhost]

CopyExplain

Now that the instance is running, we add the newly launching EC2 instance to our host group:

TASK [roles/ec2tmp : Add the temporary EC2 instance to the vmgroup] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [roles/ec2tmp : Wait for the temporary EC2 instance to be ready to accept SSH connections] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

CopyExplain

Before we move on to connecting to the EC2 host to install and configure the software stack and WordPress, we generate the endpoints variables file:

TASK [roles/endpoints : Generate the aws endpoints file] \*\*

changed: [localhost]

CopyExplain

That concludes the first section of the site.yml file, and we can now SSH into the temporary EC2 host and install everything:

PLAY [Install and configure Wordpress] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

CopyExplain

We then progress with the installation, which, as we have already discussed, is pretty much the same set of tasks that we covered *in*[*Chapter 5*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/5), *Deploying WordPress*, and [*Chapter 9*](https://subscription.packtpub.com/book/cloud-and-networking/9781835088913/9), *Moving to the Cloud* – except for these tasks, which mount the EFS file system:

TASK [roles/stack\_config : Check that the EFS volume is ready] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

TASK [roles/stack\_config : ensure rpcbind service is running] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

TASK [roles/stack\_config : mount the EFS volume] \*\*\*\*\*\*\*\*\*\*

changed: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

CopyExplain

These tasks get the PHP version and set it as a fact:

TASK [roles/stack\_config : Get the PHP version] \*\*\*\*\*\*\*\*\*\*\*

changed: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

TASK [roles/stack\_config : Set the PHP version] \*\*\*\*\*\*\*\*\*\*\*

ok: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

CopyExplain

Once that is complete, NGINX and PHP-FPM are restarted:

RUNNING HANDLER [roles/stack\_config : restart nginx] \*\*\*\*\*\*

changed: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

RUNNING HANDLER [roles/stack\_config : restart php-fpm] \*\*\*\*

changed: [ec2-18-203-221-2.eu-west-1.compute.amazonaws.com]

CopyExplain

This concludes the tasks that bootstrap our temporary EC2 instance. We can now move back to our local machine and run the final section of the sites.yml file.

First, we create the AMI and terminate the temporary EC2 instance:

TASK [roles/ec2ami : find out some facts about the instance we have been using] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [roles/ec2ami : create the AMI] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [roles/ec2ami : remove any temporary instances which are running] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Then, we wait for two minutes:

TASK [roles/ec2ami : wait for 2 minutes before continuing]

Pausing for 120 seconds

(ctrl+C then 'C' = continue early, ctrl+C then 'A' = abort)

ok: [localhost]

CopyExplain

Now, we grab the details of the AMI we just created:

TASK [roles/autoscaling : Search for all of our AMIs] \*\*\*\*\*

ok: [localhost]

TASK [roles/autoscaling : Find the last one we built] \*\*\*\*\*

ok: [localhost]

TASK [roles/autoscaling : Grab AMI ID and name of the most recent result] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

CopyExplain

Once we have those details, we create (or if we have already run the playbook, update) the Launch Template:

TASK [roles/autoscaling : Create the launch template] \*\*\*\*\*

changed: [localhost]

CopyExplain

Now, we gather the information needed for us to create/update the Auto Scaling Group:

TASK [roles/autoscaling : find out the target group ARN] \*\*

ok: [localhost]

TASK [roles/autoscaling : get information on the ec2 subnets] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

CopyExplain

Then we create the list of subnets the Auto Scaling Group will use:

TASK [roles/autoscaling : create a list of subnet IDs] \*\*\*\*

ok: [localhost] => (item={'availability\_zone': 'eu-west-1c', 'availability\_zone\_id': 'euw1-az3', 'available\_ip\_address\_count': 27, 'cidr\_block': '10.0.0.64/27', 'default\_for\_az': False, 'map\_public\_ip\_on\_launch': False, 'map\_customer\_owned\_ip\_on\_launch': False, 'state': 'available', 'subnet\_id': 'subnet-091ea1834c5fc8e48', 'vpc\_id': 'vpc-008808ff628883751', 'owner\_id': '687011238589', 'assign\_ipv6\_address\_on\_creation': False, 'ipv6\_cidr\_block\_association\_set': [], 'tags': {'role': 'compute', 'deployedBy': 'Ansible', 'Name': 'ec2-subnet-euw1-az3', 'environment': 'prod', 'description': 'Resource managed by Ansible', 'projectName': 'learnansible'}, 'subnet\_arn': 'arn:aws:ec2:eu-west-1:687011238589:subnet/subnet-091ea1834c5fc8e48', 'enable\_dns64': False, 'ipv6\_native': False, 'private\_dns\_name\_options\_on\_launch': {'hostname\_type': 'ip-name', 'enable\_resource\_name\_dns\_a\_record': False, 'enable\_resource\_name\_dns\_aaaa\_record': False}, 'id': 'subnet-091ea1834c5fc8e48'})

CopyExplain

The preceding output is repeated twice for the other two subnets we will be using; then, we finally create/update the Auto Scaling Group:

TASK [roles/autoscaling : Create/update the auto-scaling group using the launch template we just created] \*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Now, we have come to the end of our Playbook run, and we get the recap:

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ec2-18-203-221-2.eu-west-1.compute.amazonaws.com :

ok=37   changed=28   unreachable=0    failed=0    skipped=1    rescued=0    ignored=2

localhost :

ok=56   changed=23   unreachable=0    failed=0    skipped=30   rescued=0    ignored=0

CopyExplain

When I ran the playbook, it took just over 20 minutes to complete the first time, with subsequent runs taking around 10 minutes to finish.

So, from a single command and in 20ish minutes, we have a highly available vanilla WordPress installation. If you find out the public URL of your Elastic Load Balancer from the AWS console or by checking the value of the elb key in the group\_vars/generated\_aws\_endpoints.yml file, you should be able to see your site.

Terminating all the resources

Before we complete this chapter, we need to look at terminating the resources; to do this, you can run the following:

$ ansible-playbook -i hosts destroy.yml

CopyExplain

This removes everything in the reverse order that we launched it, starting with the Auto Scaling Group:

PLAY [Destroy the AWS Environment created by the site.yml playbook] \*\*\*\*\*\*\*\*\*\*\*\*

TASK [Gathering Facts] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Delete the Auto Scaling Group] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the Launch Template] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

As there can be more than one AMI, we gather some facts and then loop through removing everything that is returned:

TASK [Get information about the AMIs] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Delete the AMI(s)] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'architecture': 'x86\_64', 'creation\_date': '2024-01-12T09:44:07.000Z', 'image\_id': 'ami-0ddfeb5a1fb64c23a', 'image\_location': '687011238589/learnansible-prod-ami-2024-01-12\_0944', 'image\_type': 'machine', 'public': False, 'tags': {'Name': 'learnansible-prod-ami-2024-01-12\_0944', 'deployedBy': 'Ansible', 'environment': 'prod', 'buildDate': '2024-01-12 09:44:06', 'description': 'Resource managed by Ansible', 'projectName': 'learnansible', 'role': 'ami'}, 'virtualization\_type': 'hvm', 'source\_instance\_id': 'i-050689909fa289998'})

CopyExplain

We then remove more one-off resources:

TASK [Create a SSH Key Pair] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the group\_vars/generated\_aws\_endpoints.yml file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the RDS database] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete RDS subnet group] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the group\_vars/generated\_rds\_passwordfile file] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the EFS File System] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the group\_vars/generated\_efs\_targets.yml file]

changed: [localhost]

TASK [Delete the application elastic load balancer]\*\*\*\*\*\*\*\*

changed: [localhost]

TASK [Delete the target group] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

As the security groups reference each other, we need to create a list of them in reverse order so we can attempt to delete a group that is referenced by the next one we are going to delete:

TASK [Create a reversed list of the security group names] \*

ok: [localhost]

TASK [Delete the security groups] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item=learnansible-efs-security-group)

changed: [localhost] => (item=learnansible-rds-security-group)

changed: [localhost] => (item=learnansible-ec2-security-group)

FAILED - RETRYING: [localhost]: Delete the security groups (50 retries left).

. . . . .

FAILED - RETRYING: [localhost]: Delete the security groups (46 retries left).

changed: [localhost] => (item=learnansible-elb-security-group)

CopyExplain

You may have noticed that it failed towards the end; that is because the AWS API is having a little trouble keeping up, and the playbook is running a little ahead of the results it is returning.

We check a few more tasks:

TASK [Get information about the VPC] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Get information about the Route Table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Delete the Route Table] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'associations': [{'main': False, 'route\_table\_association\_id': 'rtbassoc-0738bb9e5aaf44848', 'route\_table\_id': 'rtb-04bc7177949ad2c92', 'subnet\_id': 'subnet-07c28d376283741f6', 'association\_state'

TASK [Delete the Internet Gateway]\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

CopyExplain

Next, we have the subnets:

TASK [Get information on the subnets] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [localhost]

TASK [Delete the subnets] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost] => (item={'availability\_zone': 'eu-west-1c', 'availability\_zone\_id': 'euw1-az3', 'available\_ip\_address\_count': 27, 'cidr\_block': '10.0.0.64/27', 'default\_for\_az': False, 'map\_public\_ip\_on\_launch': False, 'map\_customer\_owned\_ip\_on\_launch': False, 'state': 'available', 'subnet\_id': 'subnet-091ea1834c5fc8e48', 'vpc\_id': 'vpc-008808ff628883751', 'id': 'subnet-091ea1834c5fc8e48'})

. . . . .

changed: [localhost] => (item={'availability\_zone': 'eu-west-1b', 'availability\_zone\_id': 'euw1-az2', 'available\_ip\_address\_count': 27, 'cidr\_block': '10.0.0.128/27', 'default\_for\_az': False, 'map\_public\_ip\_on\_launch': False, 'map\_customer\_owned\_ip\_on\_launch': False, 'state': 'available', 'subnet\_id': 'subnet-0fd4610392872d442', 'vpc\_id': 'vpc-008808ff628883751', 'id': 'subnet-0fd4610392872d442'})

CopyExplain

Finally, we get to the VPC and recap:

TASK [Delete the VPC] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [localhost]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

localhost :

ok=23   changed=17   unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

CopyExplain

Once the playbook has finished running, I recommend you log in to the AWS console and double-check that everything has been correctly removed, as you don’t want to incur any unexpected costs.

Summary

In this chapter, we have taken our AWS deployment to the next level by creating and launching a highly available WordPress installation. By leveraging the various services offered by AWS, we engineered out any single points of failure regarding the availability of instances and our use of availability zones.

We also built logic into our playbook to use the same command to launch a new deployment or update the operating system on an existing one with a rolling deployment of new instance AMIs that contain our updated packages, leading to zero downtime during deployment.

While the WordPress deployment is as simple as possible, deploying the production-ready images would remain similar when using a more complicated application.

In our next chapter, we will look at moving from the public to the private cloud and how Ansible interacts with VMware.